DART

Release 9.9.0

Data Assimilation Research Section

Mar 12, 2021

GETTING STARTED

The Data Assimilation Research Testbed (DART) is an open-source, freely available community facility for ensemble data assimilation (DA).[1] DART is developed and maintained by the Data Assimilation Research Section (DAReS) at the National Center for Atmospheric Research (NCAR).

[1] Anderson, J. L., T. Hoar, K. Raeder, H. Liu, N. Collins, R. Torn and A. Arellano, 2009 The Data Assimilation Research Testbed: A Community Facility. *Bulletin of the American Meteorological Society*, **90**, 1283-1296, doi:10.1175/2009BAMS2618.1

CHAPTER

ONE

ENSEMBLE DATA ASSIMILATION

Ensemble DA is a technique for combining observations with numerical models to estimate the state of a physical system.

It enables modelers, observational scientists, and geophysicists to:

- Generate initial conditions for forecasts.
- Create a retrospective estimate of the state of a system, a practice known as producing a *reanalysis*.
- Assess the relative value of specific observations on forecast skill, a practice known as conducting an *observing system experiment (OSE)*.
- Estimate the value of hypothetical observations in order to inform the design of an observing system, a practice known as conducting an *observing system simulation experiment (OSSE)*.
- Determine a model's systematic bias in estimating the state of a system, a practice known as diagnosing *model error*.

The DART software environment makes it easy to explore a variety of data assimilation methods and observations with different numerical models. It provides powerful, flexible DA tools that are easy to use and customize to support efficient and reliable DA applications. While DART is primarily oriented for DA research, it has also been used in operational settings.

DART includes:

- A comprehensive tutorial introducing the concepts of ensemble DA.
- Extensive documentation of its source code.
- Interfaces to a variety of models and observation sets that can be used to introduce new users or graduate students to ensemble DA.

DART is also designed to facilitate the combination of assimilation algorithms, models, and real or synthetic observations to allow increased understanding of all three. It provides a framework for developing, testing, and distributing advances in ensemble DA to a broad community of users by removing the implementation-specific peculiarities of one-off DA systems.

These tools are intended for use by the full range of geosciencies community: beginners and experts; students and teachers; national centers and university research labs.

CHAPTER

TWO

ORGANIZATION OF THE DOCUMENTATION

Because of DART's extensive scope, this documentation is detailed and carefully organized, enabling you to easily find the information you need. If you have any questions or suggestions for improvements, please contact DAReS staff by emailing dart@ucar.edu.

The documentation is partitioned into three parts:

- a user guide that explains how to install DART and perform data assimilation
- source code documentation that provides a detailed description of the programs and modules in the repository
- a comprehensive description of data assimilation theory

CHAPTER

THREE

MANHATTAN RELEASE

The Manhattan release is new and currently supports only a subset of the models. DAReS staff will port over any requested model. Email dart@ucar.edu if yours is not on the list.

For more information on this release, see *Manhattan*.

CHAPTER

FOUR

QUICK-START

1. fork the NCAR/DART repo
2. clone your (new) fork to your machine - this will set up a remote named 'origin'
3. create a remote to point back to the NCAR/DART repo … convention dictates that this remote should be called 'upstream'
4. check out the appropriate branch
5. Download one of the tar files (listed below) of 'large' files so you can test your DART installation.
6. If you want to issue a PR, create a feature branch and push that to your fork and issue the PR.

There are several large files that are needed to run some of the tests and examples but are not included in order to keep the repository as small as possible. If you are interested in running *bgrid_solo*, *cam-fv*, or testing the *NCEP/prep_bufr* observation converter, you will need these files. These files are available at:

Release	Size	Filename
"Manhattan"	189M	Manhattan_large_files.tar.gz
"wrf-chem.r13172"	141M	wrf-chem.r13172_large_files.tar.gz
"Lanai"	158M	Lanai_large_files.tar.gz
"Kodiak"	158M	Kodiak_large_files.tar.gz
"Jamaica"	32M	Jamaica_large_files.tar.gz
"Hawaii"	32M	Hawaii_large_files.tar.gz

Download the appropriate tar file and untar it into your DART repository. Ignore any warnings about `tar: Ignoring unknown extended header keyword`.

Go into the `build_templates` directory and copy over the closest `mkmf.template`._compiler.system_ file into `mkmf.template`.

Edit it to set the NETCDF directory location if not in `/usr/local` or comment it out and set $NETCDF in your environment. *This NetCDF library must have been compiled with the same compiler that you use to compile DART and must include the F90 interfaces.*

Go into `models/lorenz_63/work` and run *quickbuild.csh*.

```
$ cd models/lorenz_63/work
$ ./quickbuild.csh
```

If it compiles, run this series of commands to do a very basic test:

```
$ ./perfect_model_obs
$ ./filter
```

If that runs and you have Matlab installed on your system add `DART/diagnostics/matlab` to your matlab search path and run the `plot_total_err` diagnostic script while in the `models/lorenz_63/work` directory. If the output plots and looks reasonable (error level stays around 2 and doesn't grow unbounded) you have successfully installed DART and completed your first assimilation with it.

If you are planning to run one of the larger models and want to use the Lorenz 63 model as a test, run `./quickbuild.csh -mpi`. It will build filter and any other MPI-capable executables with MPI.

Important: The `mpif90` command you use must have been built with the same version of the compiler as you are using.

If any of these steps fail or you don't know how to do them, go to the DART project web page listed above for very detailed instructions that should get you over any bumps in the process.

CHAPTER FIVE

CITING DART

Cite DART using the following text:

> The Data Assimilation Research Testbed (Version X.Y.Z) [Software]. (2019). Boulder, Colorado: UCAR/NCAR/CISL/DAReS. http://doi.org/10.5065/D6WQ0202

Update the DART version and year as appropriate.

CHAPTER

SIX

REFERENCES

6.1 System requirements

The DART software is intended to compile and run on many different Unix/Linux operating systems with little to no change. At this point we have no plans to port DART to Windows machines, although Windows 10 users may be interested in the free Windows Subsystem For Linux which allows developers to "run a GNU/Linux environment &mdash; including most command-line tools, utilities, and applications &mdash; directly on Windows, unmodified, without the overhead of a virtual machine" (see https://docs.microsoft.com/en-us/windows/wsl/about for more details)

Note: We have tried to make the DART code as portable as possible, but we do not have access to all compilers on all platforms, so unfortunately we cannot guarantee that the code will work correctly on your particular system.

We are genuinely interested in your experience building the system, so we welcome you to send us an email with your experiences to dart@ucar.edu.

We will endeavor to incorporate your suggestions into future versions of this guide.

Minimally, you will need:

1. a Fortran90 compiler,
2. the netCDF libraries built with the F90 interface,
3. *perl* (just about any version),
4. an environment that understands *csh*, *tcsh*, *sh*, and *ksh*
5. the long-lived Unix build tool *make*
6. and up to 1 Gb of disk space for the DART distribution.

History has shown that it is a very good idea to remove the stack and heap limits in your run-time environment with the following terminal commands:

```
> limit stacksize unlimited
> limit datasize unlimited
```

Additionally, the following tools have proven to be *nice* (but are not required to run DART):

1. ncview: a great visual browser for netCDF files.
2. the netCDF Operators (NCO): tools to perform operations on netCDF files like concatenating, slicing, and dicing
3. Some sort of MPI environment. In other words, DART does not come with *MPICH*, *LAM-MPI*, or *OpenMPI*, but many users of DART rely on these MPI distributions to run DART in a distributed-memory parallel setting. In order to use MPI with DART, please refer to the DART MPI introduction.

4. If you want to use the DART diagnostic scripts, you will need a basic MATLAB® installation. No additional toolboxes are required, and no third-party toolboxes are required.

6.2 Fortran90 compiler

The DART software is written in standard Fortran 90, with no compiler-specific extensions. It has been compiled and run with several versions of each of the following:

- GNU Fortran Compiler (known as "gfortran") (free)
- Intel Fortran Compiler for Linux and OSX
- IBM XL Fortran Compiler
- Portland Group Fortran Compiler
- Lahey Fortran Compiler
- NAG Fortran compiler
- PathScale Fortran compiler

Since recompiling the code is a necessity to experiment with different models, there are no DART binaries to distribute. If you are unfamiliar with Fortran and/or wonder why we would choose this language, see Why Fortran? for more information.

6.3 Locating netCDF library

DART uses the netCDF self-describing data format for storing the results of assimilation experiments. These files have the extension *.nc* and can be read by a number of standard data analysis tools. In particular, DART also makes use of the F90 netCDF interface which is available through the `netcdf.mod` and `typesizes.mod` modules and the `libnetcdf` library. Depending on the version, the `libnetcdff` library is also often required.

If the netCDF library does not exist on your system, you must build it (as well as the F90 interface modules).

Warning: You must build netCDF with the same compiler (including version) you plan to use for compiling DART. In practice this means that even if you have a netCDF distribution on your system, you may need to recompile netCDF in a separate location to match the compiler you will use for DART. The library and instructions for building the library or installing from a package manager may be found at the netCDF home page.

Important: The normal location for the netCDF Fortran modules and libraries would be in the `include` and `lib` subdirectories of the netCDF installation. However, different compilers or package managers sometimes place the modules and/or libraries into non-standard locations. It is required that both modules and the libraries be present.

Note: The location of the netCDF library, `libnetcdf.a`, and the locations of both `netcdf.mod` and `typesizes.mod` will be needed later. Depending on the version of netCDF and the build options selected, the Fortran interface routines may be in a separate library named `libnetcdff.a` (note the two F's). In this case both libraries are required to build executables.

6.4 Downloading DART

The DART source code is distributed on the GitHub repository NCAR/DART with the documentation served through GitHub Pages at http://dart.ucar.edu.

Go to https://github.com/NCAR/DART and clone the repository or get the ZIP file according to your preference. See the github help page on cloning for more information on how to clone a repository. Take note of the directory you installed into, which is referred to as *DARTHOME* throughout this documentation.

Note: If you are interested in contributing to DART, see the *Contributors' guide* for more information. In short, you will need to be familiar with the GitHub workflow.

Unzip or clone the distribution in your desired directory, which we refer to as `DART` in this document. Compiling the code in this tree (as is usually the case) may require a large amount of additional disk space (up to the 1 Gb required for DART), so be aware of any disk quota restrictions before continuing.

6.4.1 Organization of the repository

The top level DART source code tree contains the following directories and files:

Directory	Purpose
`assimilation_code/`	assimilation tools and programs
`build_templates/`	Configuration files for installation
`developer_tests/`	regression testing
`diagnostics/`	routines to diagnose assimilation performance
`guide/`	General documentation and DART_LAB tutorials
`models/`	the interface routines for the models
`observations/`	routines for converting observations and forward operators
`theory/`	pedagogical material discussing data assimilation theory
Files	**Purpose**
`changelog.rst`	Brief summary of recent changes
`copyright.rst`	terms of use and copyright information
`readme.rst`	Basic Information about DART

6.5 Compiling DART

Now that the DART code has been downloaded and the prerequisites have been verified, you can now begin building and verifying the DART installation.

6.5.1 Customizing the build scripts — overview

DART executable programs are constructed using two tools: *mkmf*, and *make*. The *make* utility is a very commonly used tool that requires a user-defined input file (a `Makefile`) that records dependencies between different source files. *make* then performs actions to the source hierarchy, in order of dependence, when one or more of the source files is modified. *mkmf* is a *perl* script that generates a *make* input file (named *Makefile*) and an example namelist `input.nml.<program>_default` with default values.

mkmf (think *"make makefile"*) requires two separate input files. The first is a template file which specifies the commands required for a specific Fortran90 compiler and may also contain pointers to directories containing pre- compiled utilities required by the DART system. **This template file will need to be modified to reflect your system as detailed in the next section**.

The second input file is a `path_names` file which is supplied by DART and can be used without modification. An *mkmf* command is executed which uses the `path_names` file and the mkmf template file to produce a `Makefile` which is subsequently used by the standard *make* utility.

Shell scripts that execute the *mkmf* command for all standard DART executables are provided with the standard DART distribution. For more information on the mkmf tool please see the mkmf documentation.

6.5.2 Building and Customizing the 'mkmf.template' file

A series of templates for different compilers/architectures can be found in the `DARTHOME/build_templates` directory and have names with extensions that identify the compiler, the architecture, or both. This is how you inform the build process of the specifics of your system. **Our intent is that you copy one that is similar to your system into** `DARTHOME/build_templates/mkmf.template` **and customize it.**

For the discussion that follows, knowledge of the contents of one of these templates (e.g. `DARTHOME/build_templates/mkmf.template.intel.linux`) is needed. Note that only the LAST lines of the file are shown here. The first portion of the file is a large comment block that provides valuable advice on how to customize the *mkmf* template file if needed.

```
MPIFC = mpif90
MPILD = mpif90
FC = ifort
LD = ifort
NETCDF = /usr/local
INCS = -I$(NETCDF)/include
LIBS = -L$(NETCDF)/lib -lnetcdf -lnetcdff
FFLAGS = -O2 $(INCS)
LDFLAGS = $(FFLAGS) $(LIBS)
```

variable	value
FC	the Fortran compiler
LD	the name of the loader; typically, the same as the Fortran compiler
MPIFC	the MPI Fortran compiler; see the DART MPI intro duction for more info
MPILD	the MPI loader; see the DART MPI intro duction for more info
NETCDF	the location of your root netCDF installation, which is assumed to contain `netcdf.mod` and `typesizes.mod` in the include subdirectory. Note that the value of the NETCDF variable will be used by the "INCS" and "LIBS" variables.
INCS	the includes passed to the compiler during compilation. Note you may need to change this if your netCDF includes `netcdf.mod` and `typesizes.mod` are not in the standard location under the `include` subdirectory of NETCDF.
LIBS	the libraries passed to "FC" (or "MPIFC") during compilation. Note you may need to change this if the netCDF libraries `libnetcdf` and `libnetcdff` are not in the standard location under the "lib" subdirectory of NETCDF.
FFLAGS	the Fortran flags passed to "FC" (or "MPIFC") during compilation. There are often flags used for optimized code versus debugging code. See your particular compiler's documentation for more information.
LDFLAGS	the linker flags passed to *LD* during compilation. See your particular linker's documentation for more information.

6.5.3 Customizing the path names files

Several `path_names_*` files are provided in the "work" directory for each specific model. In this case, the directory of interest is `DARTHOME/models/lorenz_63/work` (see the next section). Since each model comes with its own set of files, the `path_names_*` files typically need no customization. However, modifying these files will be required if you wish to add your model to DART. See *How do I run DART with my model?* for more information.

6.5.4 Building the Lorenz_63 DART project

In order to get started with DART, here we use the Lorenz 63 model, which is a simple ODE model with only three variables. DART supports models with many orders of magnitude more variables than three, but if you can compile and run the DART code for any ONE of the models, you should be able to compile and run DART for ANY of the models. For time-dependent filtering known as **cycling**, where observations are iteratively assimilated at multiple time steps, DART requires the ability to move the model state forward in time. For low-order models, this may be possible with a Fortran function call, but for higher-order models, this is typically done outside of DART's execution control. However, the assimilation itself is conducted the same way for **all** models. For this reason, here we focus solely on the Lorenz 63 model. If so desired, see *The Lorenz 63 model: what is it and why should we care?* for more information on this simple yet surprisingly relevant model. See *A high-level workflow of DA in DART* for further information regarding the DART workflow if you prefer to do so before building the code.

There are seven separate, stand-alone programs that are typically necessary for the end-to-end execution of a DART experiment; see below or the *What is DART?* section for more information on these programs and their interactions. All DART programs are compiled the same way, and each model directory has a directory called `work` that has the components necessary to build the executables.

Note: some higher-order models have many more than seven programs; for example, the Weather Research and Forecasting (WRF) model, which is run operationally around the world to predict regional weather, has 28 separate programs. Nonetheless, each of these programs are built the same way.

The `quickbuild.csh` in each directory builds all seven programs necessary for Lorenz 63. Describing what the `quickbuild.csh` script does is useful for understanding how to get started with DART.

The following shell commands show how to build two of these seven programs for the lorenz_63 model: *preprocess* and *obs_diag*. *preprocess* is a special program that needs to be built and run to automatically generate Fortran code that is used by DART to support a subset of observations - which are (potentially) different for every model. Once *preprocess* has been run and the required Fortran code has been generated, any of the other DART programs may be built in the same way as *obs_diag* in this example. Thus, the following runs *mkmf* to make a `Makefile` for *preprocess*, makes the *preprocess* program, runs *preprocess* to generate the Fortran observation code, runs *mkmf* to make a `Makefile` for *obs_diag*, then makes the *obs_diag* program:

```
$ cd DARTHOME/models/lorenz_63/work
$ ./mkmf_preprocess
$ make
$ ./preprocess
$ ./mkmf_obs_diag
$ make
```

The remaining executables are built in the same fashion as *obs_diag*: run the particular *mkmf* script to generate a Makefile, then execute *make* to build the corresponding program.

Currently, DART executables are built in a `work` subdirectory under the directory containing code for the given model. The Lorenz_63 model has seven `mkmf_xxxxxx` files for the following programs:

Program	Purpose
preprocess	creates custom source code for just the observations of interest
create_obs_sequence	specify a (set) of observation characteristics taken by a particular (set of) instruments
create_fixed_network_seq	specify the temporal attributes of the observation sets
perfect_model_obs	spinup and generate "true state" for synthetic observation experiments
filter	perform data assimilation analysis
obs_diag	creates observation-space diagnostic files in netCDF format to support visualization and quantification.
obs_sequence_tool	manipulates observation sequence files. This tool is not generally required (particularly for low-order models) but can be used to combine observation sequences or convert from ASCII to binary or vice-versa. Since this is a rather specialized routine, we will not cover its use further in this document.

As mentioned above, `quickbuild.csh` is a script that will build every executable in the directory. There is an optional argument that will additionally build the MPI-enabled versions which will not be covered in this set of instructions. See The DART MPI introduction page for more information on using DART with MPI.

Running `quickbuild.csh` will compile all the executables mentioned above for the lorenz_63 model:

```
$ cd DARTHOME/models/lorenz_63/work
$ ./quickbuild.csh
```

The result (hopefully) is that seven executables now reside in your work directory.

Note: The most common problem is that the netCDF libraries and/or include files were not found in the specified location(s). The second most common problem is that the netCDF libraries were built with a different compiler than the one used for DART. Find (or compile) a compatible netCDF library, edit the `DARTHOME/build_templates/mkmf.template` to point to the correct locations of the includes and library files, recreate the `Makefiles`, and try again.

6.6 Verifying installation

Note: These verification steps require MATLAB®. UCAR Member Institutions have access to institutional licenses for MATLAB, thus we have created verification tools using it.

The Lorenz model is notoriously sensitive to very small changes; in fact, the story of Lorenz discovering this sensitivity is a classic in the annals of the study of chaos, which in turn was instrumental in the development of data assimilation as a field of study. See *The Lorenz 63 model: what is it and why should we care?* or *What is data assimilation?* for more information.

This sensitivity is of practical interest for verifying these results. The initial conditions files and observations sequences are provided in ASCII, which is portable across systems, but there may be some machine-specific round-off error in the conversion from ASCII to machine binary. As Lorenz 63 is such a nonlinear model, extremely small differences in the initial conditions may eventually result in noticeably different model trajectories. Even different compiler flags may cause tiny differences that ultimately result in large differences. Your results should start out looking VERY SIMILAR and may diverge with time.

The simplest way to determine if the installation is successful is to run some of the functions available in `DARTHOME/diagnostics/matlab/`. Usually, we launch MATLAB from the `DARTHOME/models/lorenz_63/work` directory and use the MATLAB *addpath* command to make the `DARTHOME/matlab/` functions available for execution in any working directory.

In the case of this Lorenz model, we know the "true" (by definition) state of the model that is consistent with the observations, which was generated by the *perfect_model_obs* program as described in *Checking the build — running something*. The following MATLAB scripts compare the ensemble members with the truth and can calculate the error in the assimilation:

```
$ cd DARTHOME/models/lorenz_63/work
$  matlab -nodesktop
(Skipping startup messages)

    [matlab_prompt] addpath ../../../diagnostics/matlab
    [matlab_prompt] plot_total_err
    Input name of true model trajectory file;
    (cr) for perfect_output.nc
    perfect_output.nc
    Input name of ensemble trajectory file;
    (cr) for preassim.nc
    preassim.nc
    Comparing true_state.nc and
              preassim.nc
    [matlab_prompt] plot_ens_time_series
    Input name of ensemble trajectory file;
    (cr) for preassim.nc

    Comparing true_state.nc and
              preassim.nc
    Using Variable state IDs 1  2  3
```

(continues on next page)

(continued from previous page)

```
    pinfo =

      struct with fields:

                     model: 'Lorenz_63'
                   def_var: 'state'
            num_state_vars: 1
                num_copies: 20
           num_ens_members: 20
          ensemble_indices: [1 2 3 ... 18 19 20]
             min_state_var: 1
             max_state_var: 3
            def_state_vars: [1 2 3]
                     fname: 'preassim.nc'
                truth_file: 'true_state.nc'
                diagn_file: 'preassim.nc'
                truth_time: [1 200]
                diagn_time: [1 200]
                      vars: {'state'}
                      time: [200x1 double]
        time_series_length: 200
                       var: 'state'
                  var_inds: [1 2 3]
```

From the above `plot_ens_time_series` graphic, you can see the individual green ensemble members becoming more constrained with less spread as time evolves. If your figures look similar to these, you should feel confident that everything is working as intended. Don't miss the opportunity to rotate the "butterfly" plot for that classic chaos theory experience (perhaps while saying, "life, uh, finds a way").

Congratulations! You have now successfully configured DART and are ready to begin the next phase of your interaction with DART. You may wish to learn more about:

- *What is data assimilation?* — a brief introduction to ensemble data assimilation. This section includes more information about the Lorenz 63 model and how to configure the `input.nml` file to play with DA experiments in DART using the Lorenz 63 model.
- *What is DART?* — This section includes more information about DART and a basic flow chart of the overall DART workflow.
- *How do I run DART with my model?*
- *How do I add my observations to DART?*
- *How would I use DART for teaching students and/or myself?*
- *How can I contribute to DART?*

Note: In the case that the above instructions had one or more issues that either did not work for you as intended or were confusing, please contact the DART software development team at dart@ucar.edu. We value your input to make getting started as smooth as possible for new DART users!

6.7 Assimilation in a complex model

Running a successful assimilation takes careful diagnostic work and experiment iterations to find the best settings for your specific case.

The basic Kalman filter can be coded in only a handful of lines. The difficulty in getting an assimilation system working properly involves making the right choices to compensate for sampling errors, model bias, observation error, lack of model forecast divergence, variations in observation density in space and time, random correlations, etc. There are tools built into DART to deal with most of these problems but it takes careful work to apply them correctly.

If you are adding a new model or a new observation type, you should assimilate exactly one observation, with no model advance, with inflation turned off, with a large cutoff, and with the outlier threshold off (see below for how to set these namelist items).

Run an assimilation. Look at the `obs_seq.final` file to see what the forward operator computed. Use ncdiff to difference the `preassim_mean.nc` and `postassim_mean.nc` (or `output_mean.nc`) diagnostic NetCDF files and look at the changes (the "innovations") in the various model fields. Is it in the right location for that observation? Does it have a reasonable value?

Then assimilate a group of observations and check the results carefully. Run the observation diagnostics and look at the total error and spread. Look carefully at the number of observations being assimilated compared to how many are available.

Assimilations that are not working can give good looking statistics if they reject all but the few observations that happen to match the current state. The errors should grow as the model advances and then shrink when new observations are assimilated, so a timeseries plot of the RMSE should show a sawtooth pattern. The initial error entirely depends on the match between the initial ensemble and the observations and may be large but it should decrease and then reach a roughly stable level. The ensemble spread should ultimately remain relatively steady, at a value around the expected observation error level. Once you believe you have a working assimilation, this will be your baseline case.

If the ensemble spread is too small, several of the DART facilities described below are intended to compensate for ensemble members getting too close to each other. Then one by one enable or tune each of the items below, checking each time to see what is the effect on the results.

Suggestions for the most common namelist settings and features built into DART for running a successful assimilation include:

6.7.1 Ensemble size

In practice, ensemble sizes between 20 and 100 seem to work best. Fewer than 20-30 members leads to statistical errors which are too large. More than 100 members takes longer to run with very little benefit, and eventually the results get worse again. Often the limit on the number of members is based on the size of the model since you have to run N copies of the model each time you move forward in time. If you can, start with 50-60 members and then experiment with fewer or more once you have a set of baseline results to compare it with. The namelist setting for ensemble size is `&filter_nml :: ens_size`

6.7.2 Localization

There are two main advantages to using localization. One is it avoids an observation impacting unrelated state variables because of spurious correlations. The other is that, especially for large models, it improves run-time performance because only points within the localization radius need to be considered. Because of the way the parallelization was implemented in DART, localization was easy to add and using it usually results in a very large performance gain. See here for a discussion of localization-related namelist items.

6.7.3 Inflation

Since the filter is run with a number of members which is usually small compared to the number of degrees of freedom of the model (i.e. the size of the state vector or the number of EOFs needed to characterize the variability), the model uncertainty is under-represented. Other sources of error and uncertainty are not represented at all. These factors lead to the ensemble being 'over-confident', or having too little spread. More observations leads to more over-confidence. This characteristic can worsen with time, leading to ensemble collapse to a single solution. Inflation increases the spread of the members in a systematic way to overcome this problem. There are several sophisticated options on inflation, including spatial and temporal adaptive and damping options, which help deal with observations which vary in density over time and location. See here for a discussion of inflation-related namelist items.

6.7.4 Outlier rejection

Outlier rejection can be used to avoid bad observations (ones where the value was recorded in error or the processing has an error and a non-physical value was generated). It also avoids observations which have accurate values but the mean of the ensemble members is so far from the observation value that assimilating it would result in unacceptably large increments that might destablize the model run. If the difference between the observation and the prior ensemble mean is more than N standard deviations from the square root of the sum of the prior ensemble and observation error variance, the observation will be rejected. The namelist setting for the number of standard deviations to include is `&filter_nml :: outlier_threshold` and we typically suggest starting with a value of 3.0.

6.7.5 Sampling error

For small ensemble sizes a table of expected statistical error distributions can be generated before running DART. Corrections accounting for these errors are applied during the assimilation to increase the ensemble spread which can improve the assimilation results. The namelist item to enable this option is `&assim_tools_nml :: sampling_error_correction`. Additionally you will need to have the precomputed correction file `sampling_error_correction_table.nc`, in the run directory. See the description of the namelist item in the &assim_tools_nml namelist, and *system simulation programs* for instructions on where to find (or how to generate) the auxiliary file needed by this code. See Anderson (2011).

Free run/forecast after assimilation

Separate scripting can be done to support forecasts starting from the analyzed model states. After filter exits, the models can be run freely (with no assimilated data) further forward in time using one or more of the last updated model states from filter. Since all ensemble members are equally likely a member can be selected at random, or a member close to the mean can be chosen. See the *PROGRAM closest_member_tool* for one way to select a "close" member. The ensemble mean is available to be used, but since it is a combination of all the member states it may not have self-consistent features, so using a single member is usually preferred.

Evaluating observations without assimilation

Filter can be used to evaluate the accuracy of a single model state based on a set of available observations. Either copy or link the model state file so there appear to be 2 separate ensemble members (which are identical). Set the filter namelist ensemble size to 2 by setting `ens_size` to 2 in the &filter_nml namelist. Turn off the outlier threshold and both Prior and Posterior inflation by setting `outlier_threshold` to -1, and both the `inf_flavor` values to 0 in the same &filter_nml namelist. Set all observation types to be 'evaluate-only' and have no types in the 'assimilate' list by listing all types in the `evaluate_these_obs_types` list in the `&obs_kind_nml` section of the namelist, and none in the assimilation list. Run filter as usual, including model advances if needed. Run observation diagnostics on the resulting `obs_seq.final` file to compute the difference between the observed values and the predicted values from this model state.

Verification/comparison with and without assimilation

To compare results of an experiment with and without assimilating data, do one run assimilating the observations. Then do a second run where all the observation types are moved to the `evaluate_these_obs_types` list in the `&obs_kind_nml` section of the namelist. Also turn inflation off by setting both `inf_flavor` values to 0 in the &filter_nml namelist. The forward operators will still be called, but they will have no impact on the model state. Then the two sets of diagnostic state space netcdf files can be compared to evaluate the impact of assimilating the observations, and the observation diagnostic files can also be compared.

DART quality control flag added to output observation sequence file

The filter adds a quality control field with metadata 'DART quality control' to the `obs_seq.final` file. At present, this field can have the following values:

0:	Observation was assimilated successfully
1:	Observation was evaluated only but not used in the assimilation
2:	The observation was used but one or more of the posterior forward observation operators failed
3:	The observation was evaluated only but not used AND one or more of the posterior forward observation operators failed
4:	One or more prior forward observation operators failed so the observation was not used
5:	The observation was not used because it was not selected in the namelist to be assimilated or evaluated
6:	The prior quality control value was too high so the observation was not used.
7:	Outlier test failed (see below)

The outlier test computes the difference between the observation value and the prior ensemble mean. It then computes a standard deviation by taking the square root of the sum of the observation error variance and the prior ensemble variance for the observation. If the difference between the ensemble mean and the observation value is more than the specified number of standard deviations, then the observation is not used and the DART quality control field is set to 7.

6.8 Message Passing Interface

6.8.1 Introduction

DART programs can be compiled using the *Message Passing Interface (MPI)*. MPI is both a library and run-time system that enables multiple copies of a single program to run in parallel, exchange data, and combine to solve a problem more quickly.

DART does **NOT** require MPI to run; the default build scripts do not need nor use MPI in any way. However, for larger models with large state vectors and large numbers of observations, the data assimilation step will run much faster in parallel, which requires MPI to be installed and used. However, if multiple ensembles of your model fit comfortably (in time and memory space) on a single processor, you need read no further about MPI.

MPI is an open-source standard; there are many implementations of it. If you have a large single-vendor system it probably comes with an MPI library by default. For a Linux cluster there are generally more variations in what might be installed; most systems use a version of MPI called MPICH. In smaller clusters or dual-processor workstations a version of MPI called either LAM-MPI or OpenMPI might be installed, or can be downloaded and installed by the end user.

Note: OpenMP is a different parallel system; OpenMPI is a recent effort with a confusingly similar name.

An "MPI program" makes calls to an MPI library, and needs to be compiled with MPI include files and libraries. Generally the MPI installation includes a shell script called `mpif90` which adds the flags and libraries appropriate for each type of fortran compiler. So compiling an MPI program usually means simply changing the fortran compiler name to the MPI script name.

These MPI scripts are built during the MPI install process and are specific to a particular compiler; if your system has multiple fortran compilers installed then either there will be multiple MPI scripts built, one for each compiler type, or there will be an environment variable or flag to the MPI script to select which compiler to invoke. See your system documentation or find an example of a successful MPI program compile command and copy it.

DART use of MPI

To run in parallel, only the DART 'filter' program (and possibly the companion 'wakeup_filter' program) need be compiled with the MPI scripts. All other DART executables should be compiled with a standard F90 compiler and are not MPI enabled. (And note again that 'filter' can still be built as a single executable like previous releases of DART; using MPI and running in parallel is simply an additional option.) To build a parallel version of the 'filter' program, the 'mkmf_filter' command needs to be called with the '-mpi' option to generate a Makefile which compiles with the MPI scripts instead of the Fortran compiler.

See the `quickbuild.csh` script in each `$DART/models/*/work` directory for the commands that need to be edited to enable the MPI utilities. You will also need to edit the `$DART/mkmf/mkmf.template` file to call the proper version of the MPI compile script if it does not have the default name, is not in a standard location on the system, or needs additional options set to select between multiple Fortran compilers.

MPI programs generally need to be started with a shell script called 'mpirun' or 'mpiexec', but they also interact with any batch control system that might be installed on the cluster or parallel system. Parallel systems with multiple users generally run some sort of batch system (e.g. LSF, PBS, POE, LoadLeveler, etc). You submit a job request to this system and it schedules which nodes are assigned to which jobs. Unfortunately the details of this vary widely from system to system; consult your local web pages or knowledgeable system admin for help here. Generally the run scripts supplied with DART have generic sections to deal with LSF, PBS, no batch system at all, and sequential execution, but the details (e.g. the specific queue names, accounting charge codes) will almost certainly have to be adjusted.

The data assimilation process involves running multiple copies (ensembles) of a user model, with an assimilation computation interspersed between calls to the model. There are many possible execution combinations, including:

- Compiling the assimilation program 'filter' with the model, resulting in a single executable. This can be either a sequential or parallel program.
- Compiling 'filter' separately from the model, and having 2 separate executables. Either or both can be sequential or parallel.

The choice of how to combine the 'filter' program and the model has 2 parts: building the executables and then running them. At build time, the choice of using MPI or not must be made. At execution time, the setting of the 'async' namelist value in the filter_nml section controls how the 'filter' program interacts with the model.
Choices include:

- async = 0 The model and filter programs are compiled into a single executable, and when the model needs to advance, the filter program calls a subroutine. See a diagram which illustrates this option.
- async = 2 The model is compiled into a sequential (single task) program. If 'filter' is running in parallel, each filter task will execute the model independently to advance the group of ensembles. See a diagram which illustrates this option.
- async = 4 The model is compiled into an MPI program (parallel) and only 'filter' task 0 tells the startup script when it is time to advance the model. Each ensemble is advanced one by one, with the model using all the processors to run in parallel. See a diagram which illustrates this option.
- async ignored (sometimes referred to as 'async 5', but not a setting in the namelist) This is the way most large models run now. There is a separate script, outside of filter, which runs the N copies of the model to do the advance. Then filter is run, as an MPI program, and it only assimilates for a single time and then exits. The external script manages the file motion between steps, and calls both the models and filter in turn.

This release of DART has the restriction that if the model and the 'filter' program are both compiled with MPI and are run in 'async=4' mode, that they both run on the same number of processors; e.g. if 'filter' is run on 16 processors, the model must be started on 16 processors as well. Alternatively, if the user model is compiled as a single executable (async=2), 'filter' can run in parallel on any number of processors and each model advance can be executed independently without the model having to know about MPI or parallelism.
Compiling and running an MPI application can be substantially more complicated than running a single executable. There are a suite of small test programs to help diagnose any problems encountered in trying to run the new version of DART. Look in developer_tests/mpi_utilities/tests/README for instructions and a set of tests to narrow down any difficulties.

Performance issues and timing results

Getting good performance from a parallel program is frequently difficult. Here are a few of reasons why:

- Amdahl's law You can look up the actual formula for this "law" in the Wikipedia, but the gist is that the amount of serial code in your program limits how much faster your program runs on a parallel machine, and at some point (often much sooner than you'd expect) you stop getting any speedup when adding more processors.
- Surface area to volume ratio Many scientific problems involve breaking up a large grid or array of data and distributing the smaller chunks across the multiple processors. Each processor computes values for the data on the interior of the chunk they are given, but frequently the data along the edges of each chunk must be communicated to the processors which hold the neighboring chunks of the grid. As you increase the number of processors (and keep the problem size the same) the chunk size becomes smaller. As this happens, the 'surface

area' around the edges decreases slower than the 'volume' inside that one processor can compute independently of other processors. At some point the communication overhead of exchanging edge data limits your speedup.

- Hardware architecture system balance Raw CPU speeds have increased faster than memory access times, which have increased faster than access to secondary storage (e.g. I/O to disk). Computations which need to read input data and write result files typically create I/O bottlenecks. There are machines with parallel filesystems, but many programs are written to have a single processor read in the data and broadcast it to all the other processors, and collect the data on a single node before writing. As the number of processors increases the amount of time spent waiting for I/O and communication to and from the I/O node increases. There are also capacity issues; for example the amount of memory available on the I/O node to hold the entire dataset can be insufficient.
- NUMA memory Many machines today have multiple levels of memory: on-chip private cache, on-chip shared cache, local shared memory, and remote shared memory. The approach is referred as Non-Uniform Memory Access (NUMA) because each level of memory has different access times. While in general having faster memory improves performance, it also makes the performance very difficult to predict since it depends not just on the algorithms in the code, but is very strongly a function of working-set size and memory access patterns. Beyond shared memory there is distributed memory, meaning multiple CPUs are closely connected but cannot directly address the other memory. The communication time between nodes then depends on a hardware switch or network card, which is much slower than local access to memory. The performance results can be heavily influenced in this case by problem size and amount of communication between processes.

Parallel performance can be measured and expressed in several different ways. A few of the relevant definitions are:

- Speedup Generally defined as the wall-clock time for a single processor divided by the wall-clock time for N processors.
- Efficiency The speedup number divided by N, which for perfect scalability will remain at 1.0 as N increases.
- Strong scaling The problem size is held constant and the number of processors is increased.
- Weak scaling The problem size grows as the number of processors increases so the amount of work per processor is held constant.

We measured the strong scaling efficiency of the DART 'filter' program on a variety of platforms and problem sizes. The scaling looks very good up to the numbers of processors available to us to test on. It is assumed that for MPP (Massively-Parallel Processing) machines with 10,000s of processors that some algorithmic changes will be required. These are described in this paper.

User considerations for their own configurations

Many parallel machines today are a hybrid of shared and distributed memory processors; meaning that some small number (e.g. 2-32) of CPUs share some amount of physical memory and can transfer data quickly between them, while communicating data to other CPUs involves slower communication across either some kind of hardware switch or fabric, or a network communication card like high speed ethernet.

Running as many tasks per node as CPUs per shared-memory node is in general good, unless the total amount of virtual memory used by the program exceeds the physical memory. Factors to consider here include whether each task is limited by the operating system to 1/Nth of the physical memory, or whether one task is free to consume more than its share. If the node starts paging memory to disk, performance takes a huge nosedive.

Some models have large memory footprints, and it may be necessary to run in MPI mode not necessarily because the computation is faster in parallel, but because the dataset size is larger than the physical memory on a node and must be divided and spread across multiple nodes to avoid paging to disk.

6.9 Filters

The different types of assimilation algorithms (EAKF, ENKF, Kernel filter, Particle filter, etc.) are determined by the `&assim_tools_nml:filter_kind` entry, described in *MODULE assim_tools_mod*. Despite having 'filter' in the name, they are assimilation algorithms and so are implemented in `assim_tools_mod.f90`.

6.10 Inflation

In pre-Manhattan DART, there were two choices for the basic type of inflation: observation-space or state-space. Observation-space inflation is no longer supported. (If you are interested in observation-space inflation, talk to Jeff first.) The rest of this discussion applies to state-space inflation.

State-space inflation changes the spread of an ensemble without changing the ensemble mean. The algorithm computes the ensemble mean and standard deviation for each variable in the state vector in turn, and then moves the member's values away from the mean in such a way that the mean remains unchanged. The resulting standard deviation is larger than before. It can be applied to the Prior state, before observations are assimilated (the most frequently used case), or it can be applied to the Posterior state, after assimilation. See Anderson (2007), Anderson (2009).

Inflation values can vary in space and time, depending on the specified namelist values. Even though we talk about a single inflation value, the inflation has a gaussian distribution with a mean and standard deviation. We use the mean value when we inflate, and the standard deviation indicates how sure of the value we are. Larger standard deviation values mean "less sure" and the inflation value can increase more quickly with time. Smaller values mean "more sure" and the time evolution will be slower since we are more confident that the mean (inflation value) is correct.

The standard deviation of inflation allows inflation values to increase with time, if required by increasing density or frequency of observations, but it does not provide a mechanism to reduce the inflation when the frequency or density of observations declines. So there is also an option to damp inflation through time. In practice with large geophysical models using damped inflation has been a successful strategy.

The following namelist items which control inflation are found in the `input.nml` file, in the &filter_nml namelist. The detailed descriptions are in the namelist page. Here we try to give some basic advice about commonly used values and suggestions for where to start. Spatial variation is controlled by `inf_flavor`, which also controls whether there's any inflation, `inf_initial_from_restart`, and `inf_initial`, as described below. Time variation is controlled by `inf_sd_initial_from_restart`, `inf_sd_initial`, `inf_sd_lower_bound`, `inf_damping`, `inf_lower_bound` and `inf_upper_bound`.

In the namelist each entry has two values. The first is for Prior inflation and the second is for Posterior inflation.

`&filter_nml :: inf_flavor` *valid values:*0, 2, 3, 4, 5

Set the type of Prior and Posterior inflation applied to the state vector. Values mean:

- **0:** No inflation (Prior and/or Posterior) and all other inflation variables are ignored
- [**1:** Deprecated: Observation space inflation]
- **2:** Spatially-varying state space inflation (gaussian)
- **3:** Spatially-uniform state space inflation (gaussian)
- **4:** Relaxation To Prior Spread (Posterior inflation only)
- **5:** Enhanced Spatially-varying state space inflation (inverse gamma)

Spatially-varying state space inflation stores an array of inflation values, one for each item in the state vector. If time-evolution is enabled each value can evolve independently. Spatially-uniform state space inflation uses a single inflation value for all items in the state vector. If time-evolution is enabled that single value can evolve. See `inf_sd_*` below for control of the time-evolution behavior. Enhanced spatially-varying inflation uses an inverse-gamma distribution which allows the standard deviation of the inflation to increase or decrease through time and may produce better results. In practice we recommend starting with no inflation (both values 0). Then try inflation type 2 or 5 prior inflation and no inflation (0) for posterior. WARNING: even if inf_flavor is not 0, inflation will be turned off if `inf_damping` is set to 0.

`&filter_nml :: inf_initial_from_restart` *valid values:* .true. or .false.

If true, read the inflation values from an inflation restart file named `input_{prior,post}inf_mean.nc.` An initial run could be done to let spatially-varying inflation values evolve in a spinup phase, and then the saved values can be read back in and used as fixed values in further runs. Or if time-varying inflation is used, then the restart file from the previous job step must be supplied as an input file for the next step.

`&filter_nml :: inf_initial` *valid values:* real numbers, usually 1.0 or slightly larger If not reading in inflation values from a restart file, the initial value to set for the inflation. Generally we recommend starting with just slightly above 1.0, maybe 1.02, for a slight amount of initial inflation.

`&filter_nml :: inf_lower_bound` *valid values:* real numbers, usually 1.0 or slightly larger

If inflation is time-evolving (see `inf_sd_*` below), then this sets the lowest value the inflation can evolve to. Setting a number less than one allows for deflation but generally in a well-observed system the ensemble needs more spread and not less. We recommend a setting of 1.0.

`&filter_nml :: inf_upper_bound` *valid values:* real numbers, larger than 1.0

If inflation is time-evolving (see `inf_sd_*` below), then this sets the largest value the inflation can evolve to. We recommend a setting of 100.0, although if the inflation values reach those levels there is probably a problem with the assimilation.

`&filter_nml :: inf_damping` *valid values:* 0.0 to 1.0

Applies to all state-space inflation types, but most frequently used with time-adaptive inflation variants. The difference between the current inflation value and 1.0 is multiplied by this factor before the next assimilation cycle. So the inflation values are pushed towards 1.0, from above or below (if inf_lower_bound allows inflation values less than 1.0). A value of 0.0 turns all inflation off by forcing the inflation value to 1.0. A value of 1.0 turns damping off by leaving the original inflation value unchanged. We have had good results in large geophysical models using time- and space-adaptive state-space inflation and setting the damping to a value of 0.9, which damps slowly.

`&filter_nml :: inf_sd_initial_from_restart` *valid values:* .true. or .false.

If true, read the inflation standard deviation values from an restart file named `input_{prior, post}inf_sd.nc.` See the comments above about `inflation_initial_from_restart`.

`&filter_nml :: inf_sd_initial` *valid values:* 0.0 to disable evolution of inflation, > 0.0 otherwise

The initial value to set for the inflation standard deviation, IF not reading in inflation standard deviation values from a file. This value (or these values) control whether the inflation values evolve with time or not. A negative value or 0.0 prevents the inflation values from being updated, so they are constant throughout the run. If positive, the inflation values evolve through time. We have had good results setting this and `inf_sd_lower_bound` to 0.6 for large geophysical models.

`&filter_nml :: inf_sd_lower_bound` *valid values:* 0.0 to disable evolution of inflation, > 0.0 otherwise

If the setting of `inf_sd_initial` is 0 (to disable time evolution of inflation) then set this to the same value.

Otherwise, the standard deviation of the inflation cannot fall below this value. Smaller values will restrict the inflation to vary more slowly with time; larger values will allow the inflation to adapt more quickly. We have had good results setting this and `inf_sd_initial` to 0.6 for large geophysical models. Since the inf_sd_lower_bound is a scalar, it is not possible to set different lower bounds for different parts of the state vector.

Time-varying inflation with flavor 2 generally results in the inflation standard deviation for all state variables shrinking to the lower bound and staying there. For flavor 5, the inflation standard deviation value is allowed to increase and decrease.

&filter_nml :: inf_sd_max_change *valid values:* 1.0 to 2.0

Used only with the Enhanced inflation (flavor 5). The Enhanced inflation algorithm allows the standard deviation to increase as well as decrease. The inf_sd_max_change controls the maximum increase of the standard deviation in an assimilation cycle. A value of 1.0 means it will not increase, a value of 2.0 means it can double; a value inbetween sets the percentage it can increase, e.g. 1.05 is a limit of 5%. Suggested value is 1.05 (max increase of 5% per cycle).

Because the standard deviation for original flavor 2 could never increase, setting the inf_sd_initial value equal to the inf_sd_lower_bound value effectively fixed the standard deviation at a constant value. To match the same behavior, if they are equal and Enhanced inflation (flavor 5) is used it will also use that fixed value for the standard deviation of the inflation. Otherwise the standard deviation will adapt as needed during each assimilation cycle.

&filter_nml :: inf_deterministic *valid values:* .true. or .false.

Recommend always using .true..

6.10.1 Guidance regarding inflation

The suggested procedure for testing inflation options is to start without any (both inf_flavor values set to 0 and inf_damping > 0.). Then enable Prior state space, spatially-varying inflation, with no Posterior inflation (set inf_flavor to [2, 0]). Then try damped inflation (set inf_damping to 0.9 and set inf_sd_initial and inf_sd_lower_bound to 0.6). The inflation values and standard deviation are written out to files with _{prior, post}inf_{mean, sd} in their names. These NetCDF files can be viewed with common tools (we often use ncview). Expected inflation values are generally in the 1 to 30 range; if values grow much larger than this it usually indicates a problem with the assimilation.

It is possible to set inflation values in an existing netCDF file by using one of the standard NCO utilities like "ncap2" on a copy of a restart file. Inflation mean and sd values look exactly like restart values, arranged by variable type like T, U, V, etc.

Here's an example of using ncap2 to set the T,U and V inf values:

```
ncap2 -s 'T=1.0;U=1.0;V=1.0' wrfinput_d01 input_priorinf_mean.nc
ncap2 -s 'T=0.6;U=0.6;V=0.6' wrfinput_d01 input_priorinf_sd.nc
-or-
ncap2 -s 'T(:,:,:)=1.0;U(:,:,:)=1.0;V(:,:,:)=1.0' wrfinput_d01 input_priorinf_mean.nc
ncap2 -s 'T(:,:,:)=0.6;U(:,:,:)=0.6;V(:,:,:)=0.6' wrfinput_d01 input_priorinf_sd.nc
```

Some versions of the NCO utilities change the full 3D arrays into a single scalar. If that's your result (check your output with ncdump -h) use the alternate syntax or a more recent version of the NCO tools.

6.11 Increments: How does output differ from input?

The innovations to the model state are easy to derive. Use the NCO Operator *ncdiff* to difference the two DART diagnostic netCDF files to create the innovations. Be sure to check the *CopyMetaData* variable to figure out what *copy* is of interest. Then, use *ncview* to explore the innovations or the inflation values or . . .

If the assimilation used state-space inflation, the inflation fields will be added as additional 'copies'. A sure sign of trouble is if the inflation fields grow without bound. As the observation network changes, expect the inflation values to change.

The only other thing I look for in state-space is that the increments are 'reasonable'. As the assimilation 'burns in', the increments are generally larger than increments from an assimilation that has been cycling for a long time. If the increments keep getting bigger, the ensemble is continually drifting away from the observation. Not good. In *ncview*, it is useful to navigate to the copy/level of interest and re-range the data to values appropriate to the current data and then hit the '>>' button to animate the image. It should be possible to get a sense of the magnitude of the innovations as a function of time.

6.11.1 Example from a model of intermediate complexity: the bgrid model

I ran a perfect model experiment with the bgrid model in the DART-default configuration and turned on some adaptive inflation for this example. To fully demonstrate the adaptive inflation, it is useful to have an observation network that changes through time. I created two observation sequence files: one that had a single 'RADIOSONDE_TEMPERATURE' observation at the surface with an observation error variance of 1.5 degrees Kelvin - repeated every 6 hours for 6 days (24 timesteps); and one that had 9 observations locations clustered in about the same location that repeated every 6 hours for 1.5 days (6 timesteps). I merged the two observation sequences into one using `obs_sequence_tool` and ran them through `perfect_model_obs` to derive the observation values and create an `obs_seq.out` file to run through `filter`.

Note: Other models may have their ensemble means and spreads and inflation values in separate files. *See the table of possible filenames.*

```
$ cd ${DARTROOT}/models/bgrid_solo/work
$ ncdiff analysis.nc preassim.nc Innov.nc
$ ncview preassim.nc &
$ ncview Innov.nc &
$ ncdump -v MemberMetadata preassim.nc
netcdf preassim {
dimensions:
        metadatalength = 64 ;
        member = 20 ;
        time = UNLIMITED ; // (24 currently)
        NMLlinelen = 129 ;
        NMLnlines = 303 ;
        StateVariable = 28200 ;
        TmpI = 60 ;
        TmpJ = 30 ;
        lev = 5 ;
        VelI = 60 ;
        VelJ = 29 ;
variables:
        char MemberMetadata(member, metadatalength) ;
                MemberMetadata:long_name = "Metadata for each copy/member" ;
        ...
```

(continues on next page)

(continued from previous page)

```
        double ps(time, member, TmpJ, TmpI) ;
                ps:long_name = "surface pressure" ;
                ps:units = "Pa" ;
                ps:units_long_name = "pascals" ;
        double t(time, member, lev, TmpJ, TmpI) ;
                t:long_name = "temperature" ;
                t:units = "degrees Kelvin" ;
        double u(time, member, lev, VelJ, VelI) ;
                u:long_name = "zonal wind component" ;
                u:units = "m/s" ;
        double v(time, member, lev, VelJ, VelI) ;
                v:long_name = "meridional wind component" ;
                v:units = "m/s" ;
        double ps_mean(time, TmpJ, TmpI) ;       The ensemble mean   is now a␣
↪separate variable.
        double t_mean(time, lev, TmpJ, TmpI) ;    The ensemble spread is now a␣
↪separate variable.
        double u_mean(time, lev, VelJ, VelI) ;    If I was using inflation, they␣
↪would also be separate variables.
        double v_mean(time, lev, VelJ, VelI) ;
        double ps_sd(time, TmpJ, TmpI) ;
        double t_sd(time, lev, TmpJ, TmpI) ;
        double u_sd(time, lev, VelJ, VelI) ;
        double v_sd(time, lev, VelJ, VelI) ;

data:
  MemberMetadata =
  "ensemble member      1 ",
  "ensemble member      2 ",
  "ensemble member      3 ",
  "ensemble member      4 ",
  "ensemble member      5 ",
  "ensemble member      6 ",
  "ensemble member      7 ",
  "ensemble member      8 ",
  "ensemble member      9 ",
  "ensemble member     10 ",
  "ensemble member     11 ",
  "ensemble member     12 ",
  "ensemble member     13 ",
  "ensemble member     14 ",
  "ensemble member     15 ",
  "ensemble member     16 ",
  "ensemble member     17 ",
  "ensemble member     18 ",
  "ensemble member     19 ",
  "ensemble member     20 " ;
}
```

This is an exploration of the `preassim.nc` file. Note that I selected the '**t**' field, turned the coastlines 'off' under the 'Opts' button, used the 'Repl' instead of 'Bi-lin' (to more faithfully represent the model resolution), *navigated to copy 23 of 24 (in this case, the***inflation mean**) select the **inflation mean variable of your choice** and advanced to the last timestep. The image plot is pretty boring, but does indicate that the inflation values are restricted to where I put the observations. Right-clicking on the 'Range' button automatically re-ranges the colorbar to the min/max of the current data. Clicking on any location generates a time series figure.

This is an exploration of the `Innov.nc` file as created by *ncdiff*. Note that the titles are somewhat misleading because they reflect information from the first file given to *ncdiff*. This time I left the rendering as 'Bi-lin' (which obfuscates the model resolution), *navigated to copy 1 of 24 (in this case, the***ensemble mean***)* selected the **t_mean** variable and advanced to the 6th timestep. Right-click on the 'Range' button to reset the colorbar. The image plot confirms that the innovations are restricted to a local region. Clicking on any location generates a time series.

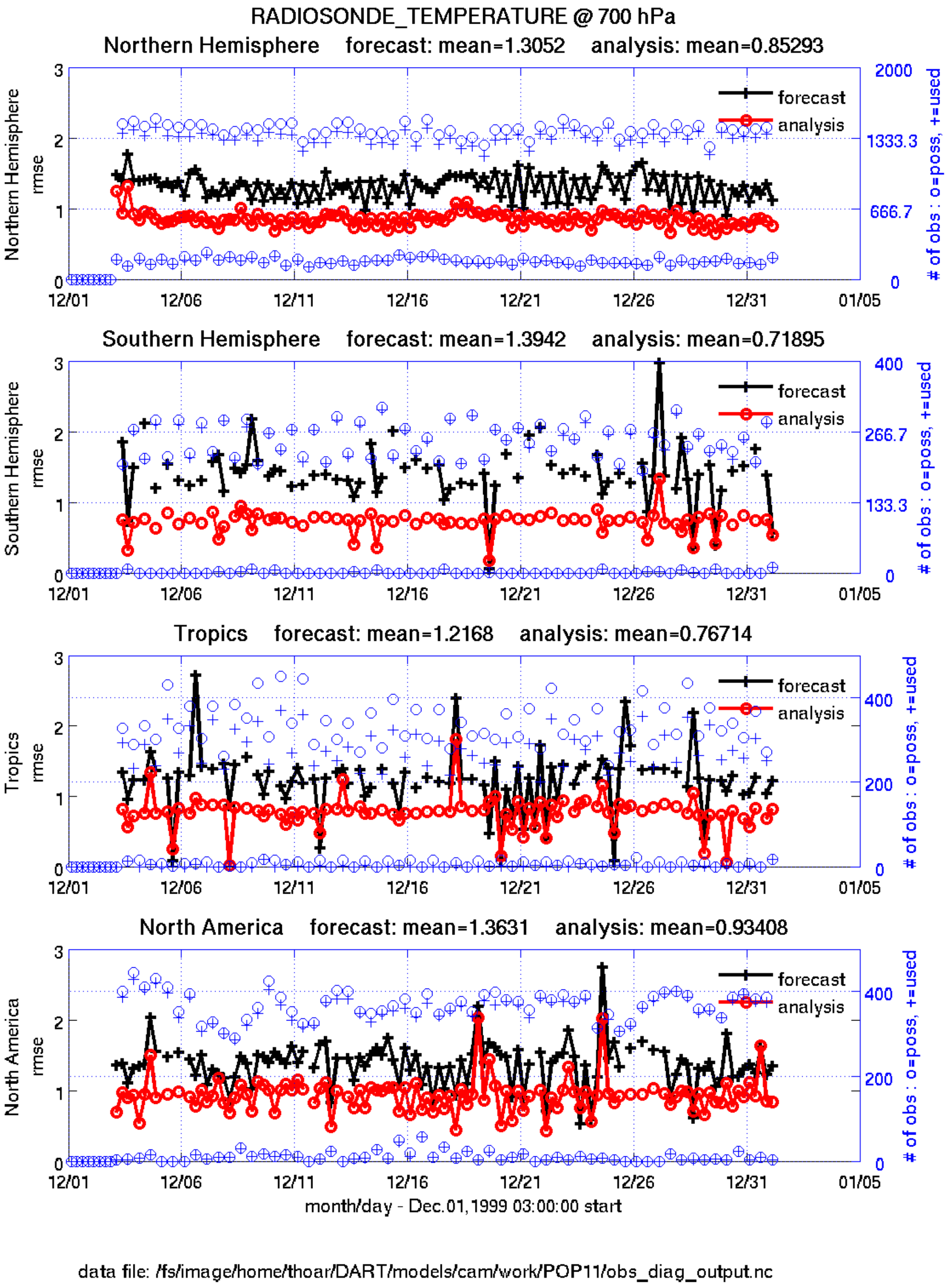
This is fundamentally the same as the previous panel except that I have now selected the '**u**' **u_mean** variable. Despite the fact the observations were only of '**t**', the assimilation has generated (rightly so) increments to the '**u**' state variable.

6.12 MATLAB® observation space diagnostics

The observation-space functions are in the `$DARTROOT/diagnostics/matlab` directory. Once you have processed the `obs_seq.final` files into a single `obs_diag_output.nc`, you can use that as input to your own plotting routines or use the following DART MATLAB® routines:

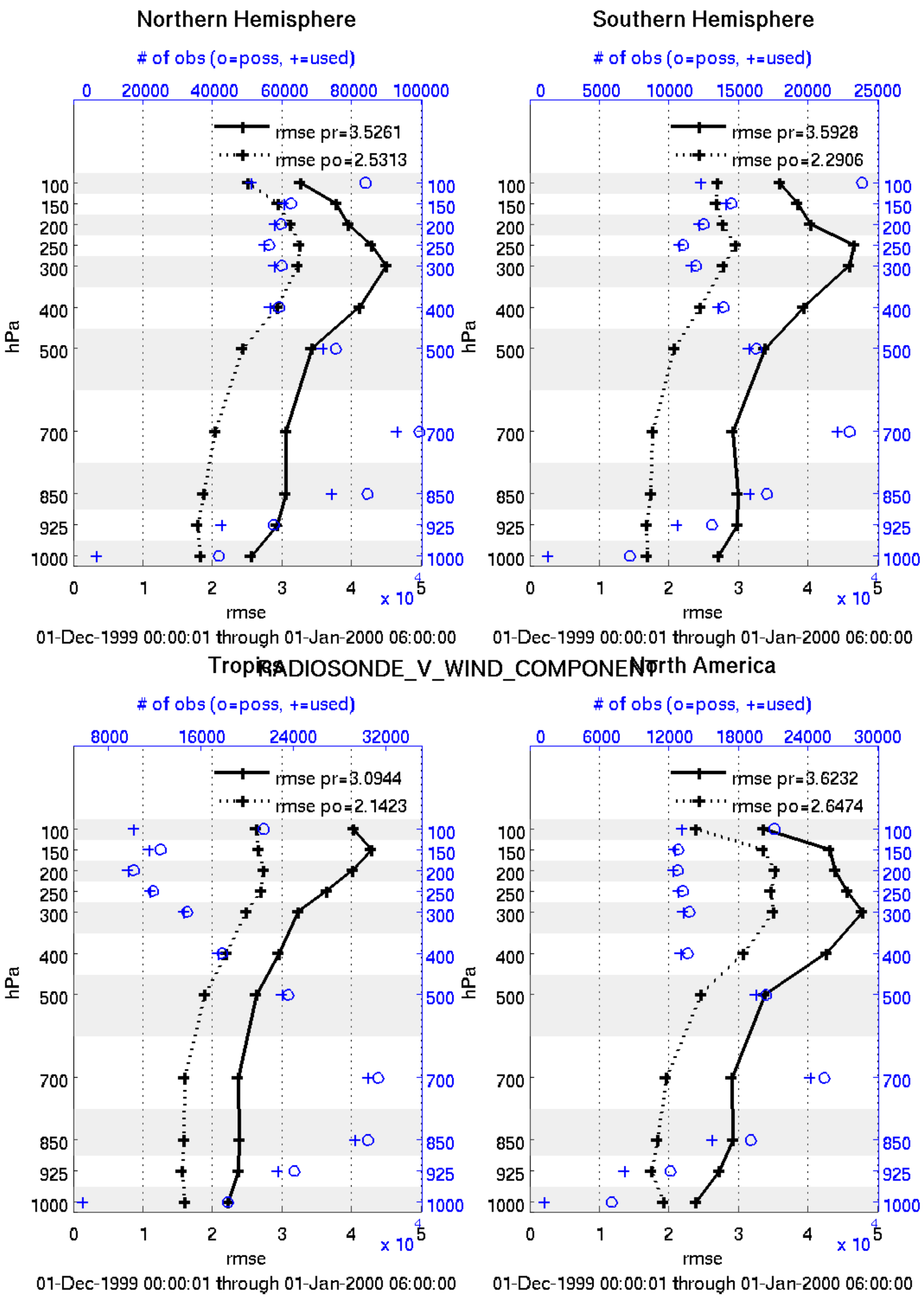
plot_evolution.m plots the temporal evolution of any of the quantities above for each variable for specified levels. The number of observations possible and used are plotted on the same axis.

```
fname      = 'POP11/obs_diag_output.nc';          % netcdf file produced by 'obs_diag'
copystring = 'rmse';                              % 'copy' string == quantity of␣
↪interest
plotdat    = plot_evolution(fname,copystring);   % -- OR --
plotdat    = plot_evolution(fname,copystring,'obsname','RADIOSONDE_TEMPERATURE');
```



plot_profile.m plots the spatial and temporal average of any specified quantity as a function of height. The number of observations possible and used are plotted on the same axis.

```
fname       = 'POP11/obs_diag_output.nc';         % netcdf file produced by 'obs_diag'
copystring = 'rmse';                              % 'copy' string == quantity of␣
 ↪interest
plotdat     = plot_profile(fname,copystring);
```



data file: /fs/image/home/thoar/DART/models/cam/work/POP11/obs_diag_output.nc

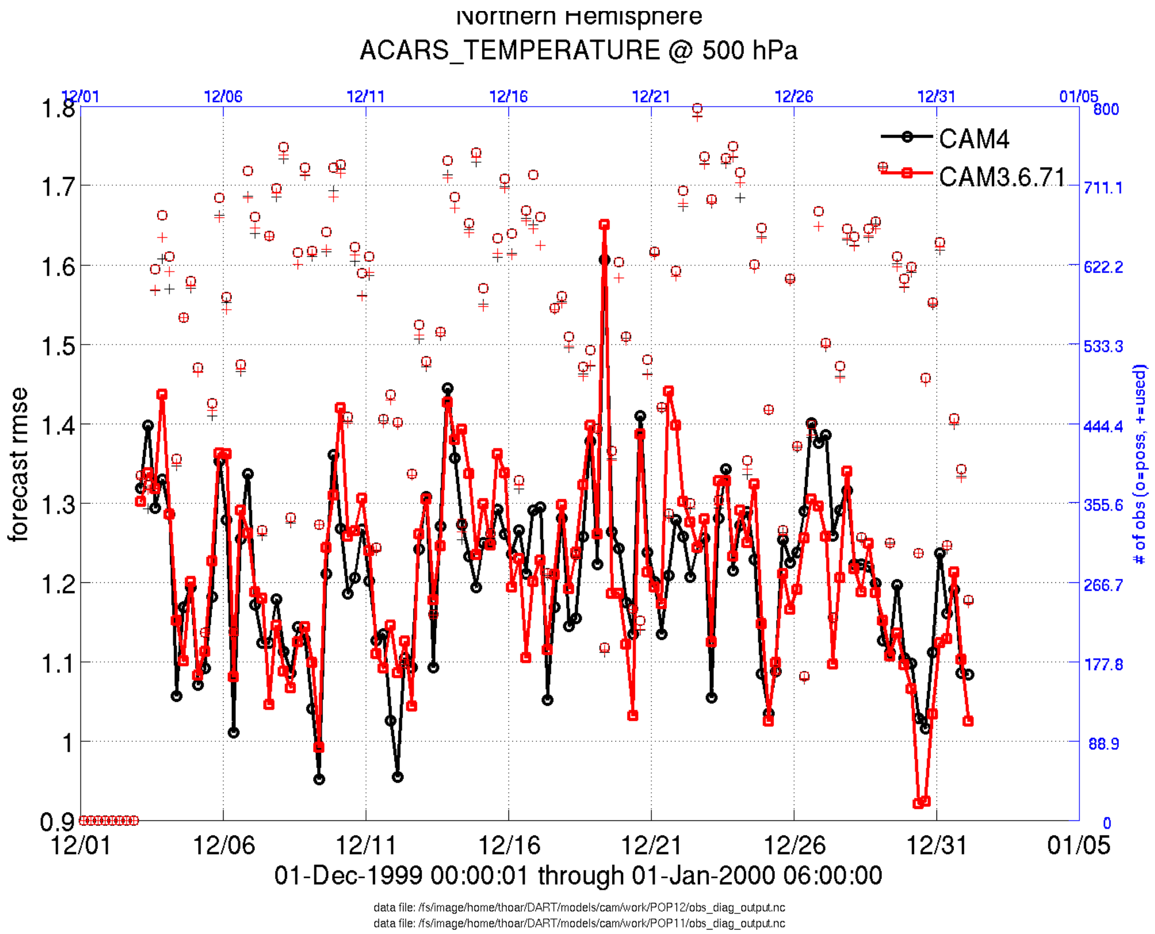
plot_rmse_xxx_evolution.m same as `plot_evolution.m` but will overlay **rmse** on the same axis.

plot_rmse_xxx_profile.m same as `plot_profile.m` with an overlay of **rmse**.

plot_bias_xxx_profile.m same as `plot_profile.m` with an overlay of **bias**.

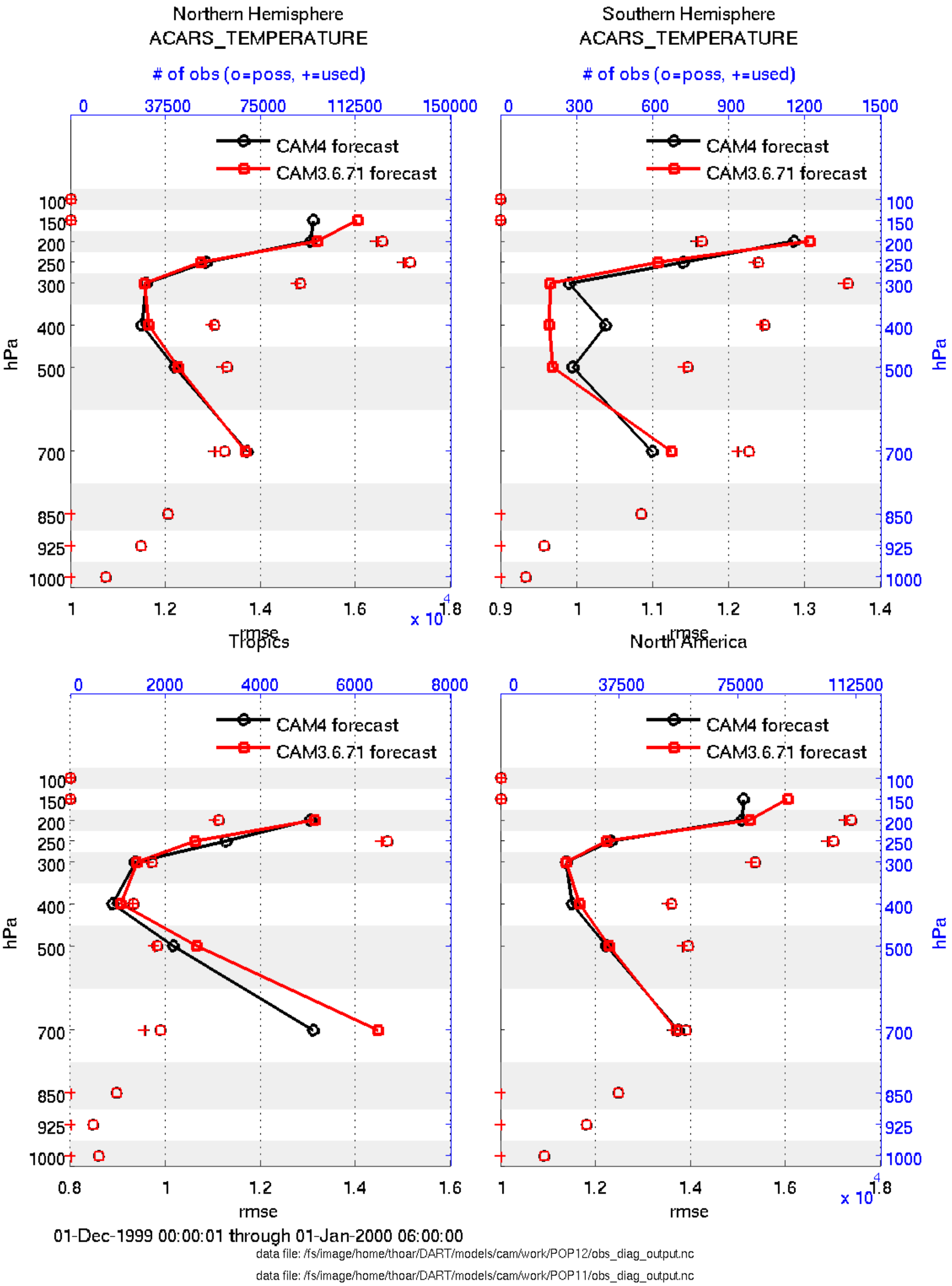
two_experiments_evolution.m same as `plot_evolution.m` but will overlay multiple (more than two, actually) experiments (i.e. multiple `obs_diag_output.nc` files) on the same axis. A separate figure is created for each region in the `obs_diag_output.nc` file.

```
files    = {'POP12/obs_diag_output.nc','POP11/obs_diag_output.nc'};
titles   = {'CAM4','CAM3.6.71'};
varnames = {'ACARS_TEMPERATURE'};
qtty     = 'rmse';
prpo     = 'prior';
levelind = 5;
two_experiments_evolution(files, titles,{'ACARS_TEMPERATURE'}, qtty, prpo, levelind)
```



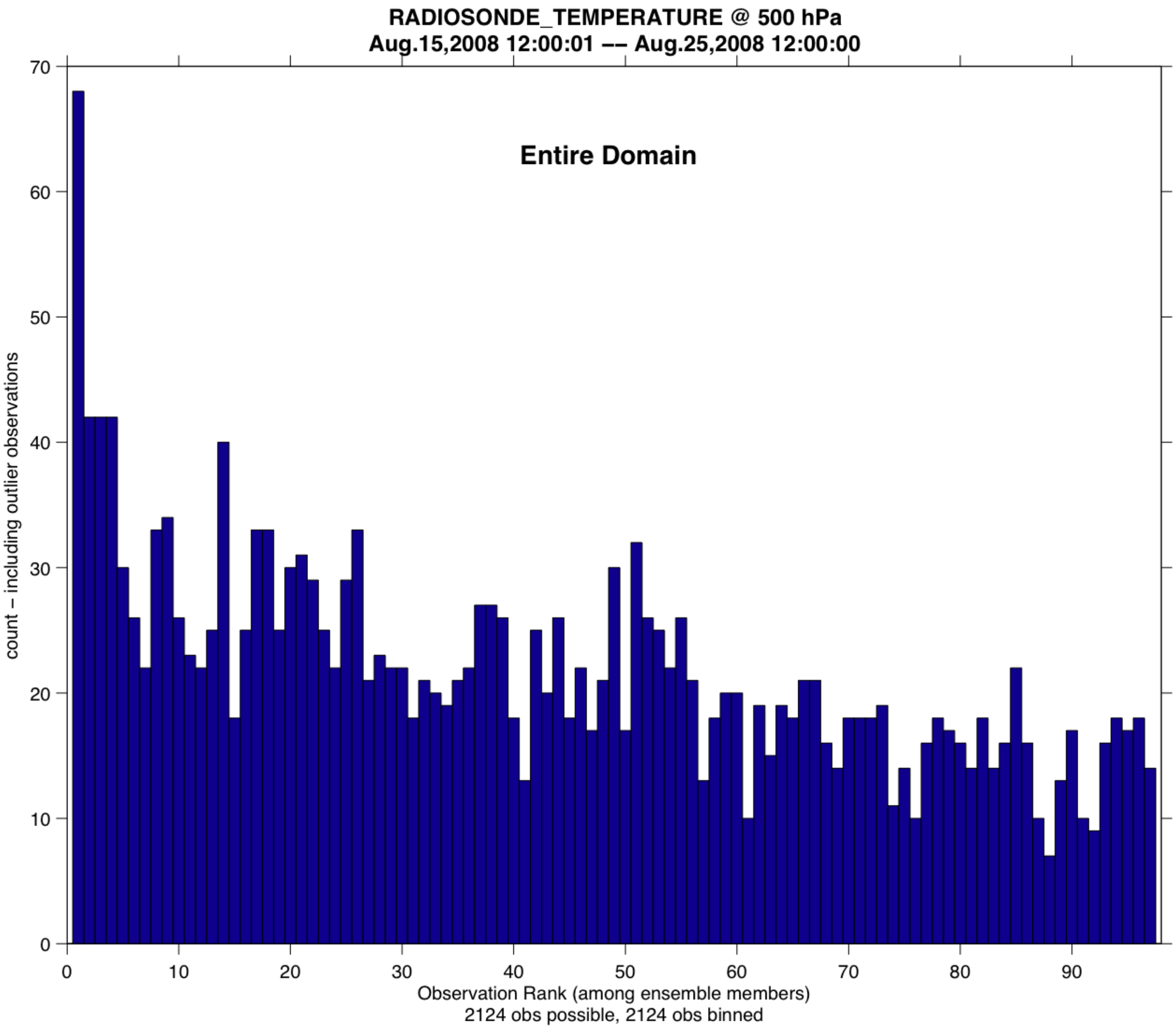
two_experiments_profile.m same as `plot_profile.m` but will overlay multiple (more than two, actually) experiments (i.e. multiple `obs_diag_output.nc` files) on the same axis. If the `obs_diag_output.nc` file was created with multiple regions, there are multiple axes on a single figure.

```
files    = {'POP12/obs_diag_output.nc','POP11/obs_diag_output.nc'};
titles   = {'CAM4','CAM3.6.71'};
varnames = {'ACARS_TEMPERATURE'};
qtty     = 'rmse';
prpo     = 'prior';
two_experiments_profile(files, titles, varnames, qtty, prpo)
```



plot_rank_histogram.m will create rank histograms for any variable that has that information present in `obs_diag_output.nc`.

```
fname     = 'obs_diag_output.nc'; % netcdf file produced by 'obs_diag'
timeindex = 3;                    % plot the histogram for the third timestep
plotdat   = plot_rank_histogram(fname, timeindex, 'RADIOSONDE_TEMPERATURE');
```



data file: /fs/image/home/thoar/DART/models/wrf/work/obs_diag_output.nc

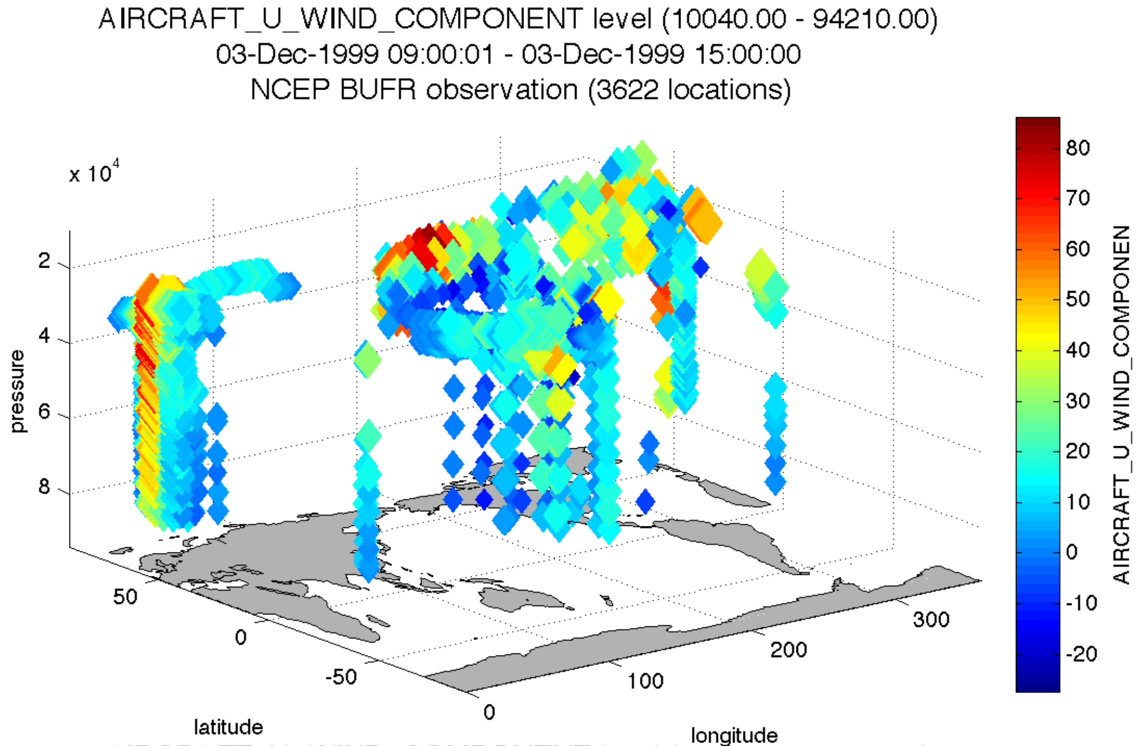
You may also convert observation sequence files to netCDF by using *PROGRAM obs_seq_to_netcdf*. All of the following routines will work on observation sequences files AFTER an assimilation (i.e. `obs_seq.final` files that have been converted to netCDF), and some of them will work on `obs_seq.out`-type files that have been converted.

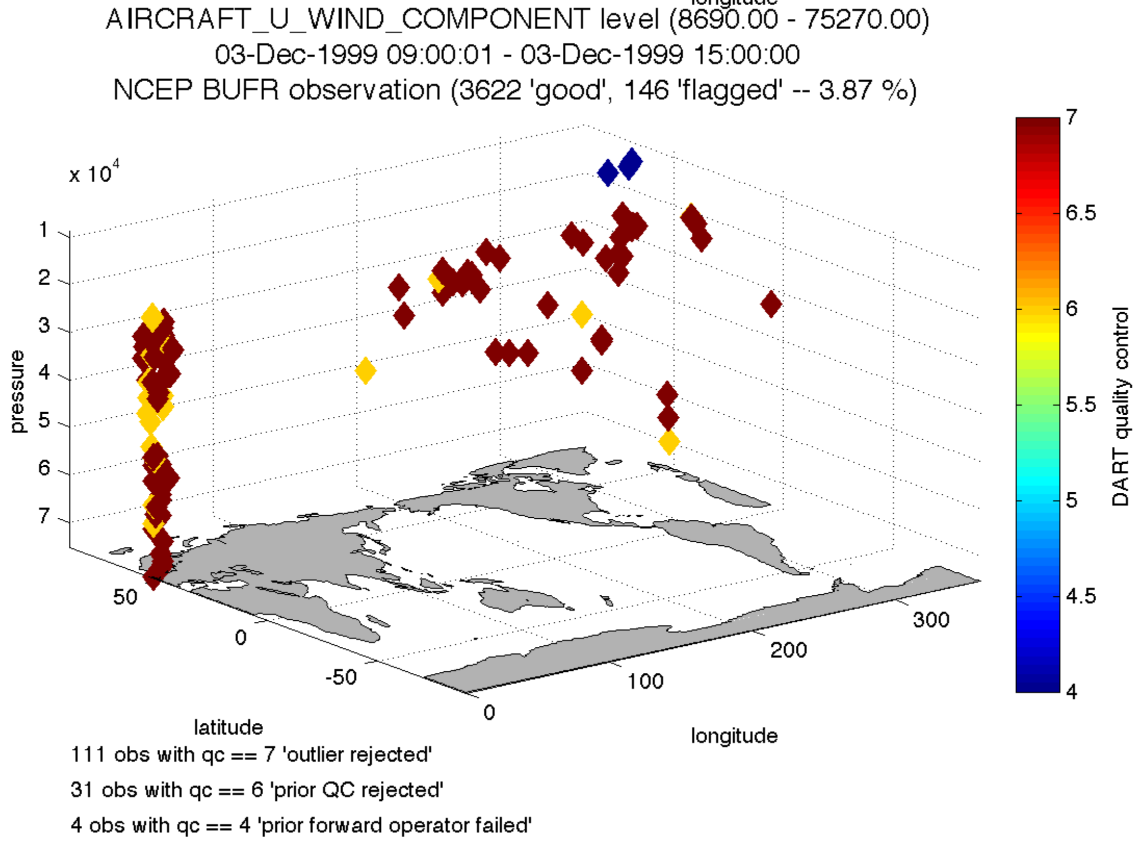
read_obs_netcdf.m reads a particular variable and copy from a netCDF-format observation sequence file and returns a single structure with useful bits for plotting/exploring. This routine is the back-end for `plot_obs_netcdf.m`.

```
fname         = 'obs_sequence_001.nc';
ObsTypeString = 'RADIOSONDE_U_WIND_COMPONENT';   % or 'ALL' ...
region        = [0 360 -90 90 -Inf Inf];
CopyString    = 'NCEP BUFR observation';
QCString      = 'DART quality control';
verbose       = 1;   % anything > 0 == 'true'
obs = read_obs_netcdf(fname, ObsTypeString, region, CopyString, QCString, verbose);
```

plot_obs_netcdf.m creates a 3D scatterplot of the observation locations, color-coded to the observation values. A second axis will also plot the QC values if desired.

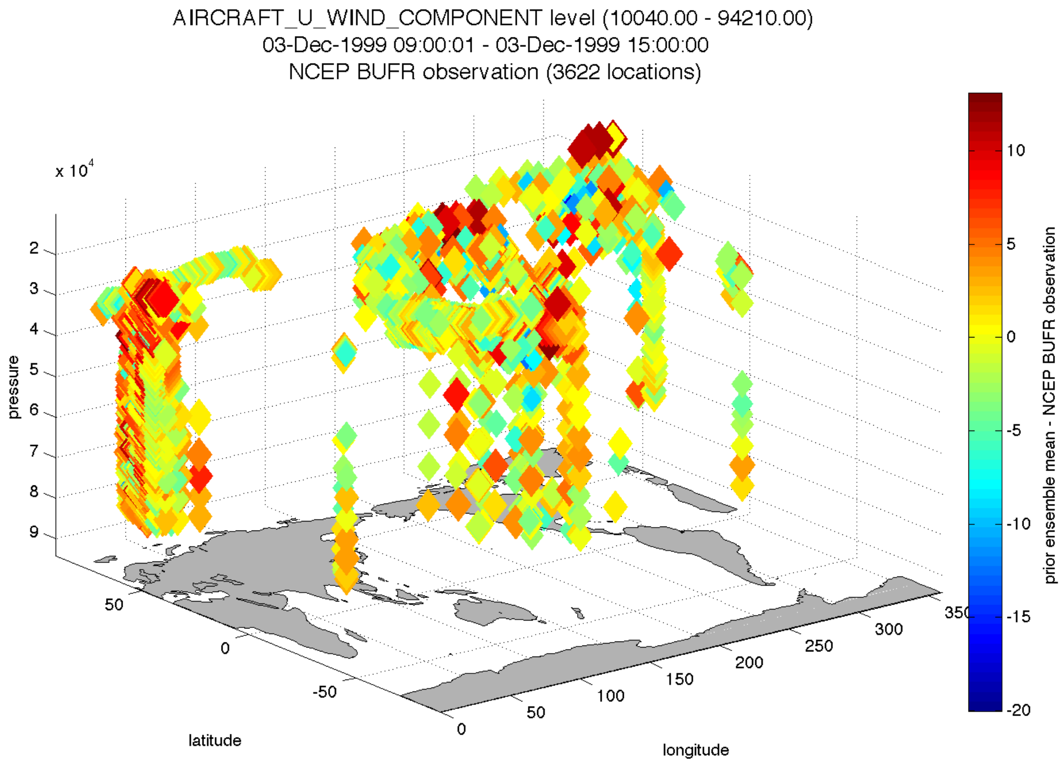
```
fname         = 'POP11/obs_epoch_011.nc';
region        = [0 360 -90 90 -Inf Inf];
ObsTypeString = 'AIRCRAFT_U_WIND_COMPONENT';
CopyString    = 'NCEP BUFR observation';
QCString      = 'DART quality control';
maxgoodQC     = 2;
verbose       = 1;   % > 0 means 'print summary to command window'
twoup         = 1;   % > 0 means 'use same Figure for QC plot'
bob = plot_obs_netcdf(fname, ObsTypeString, region, CopyString, ...
                  QCString, maxgoodQC, verbose, twoup);
```





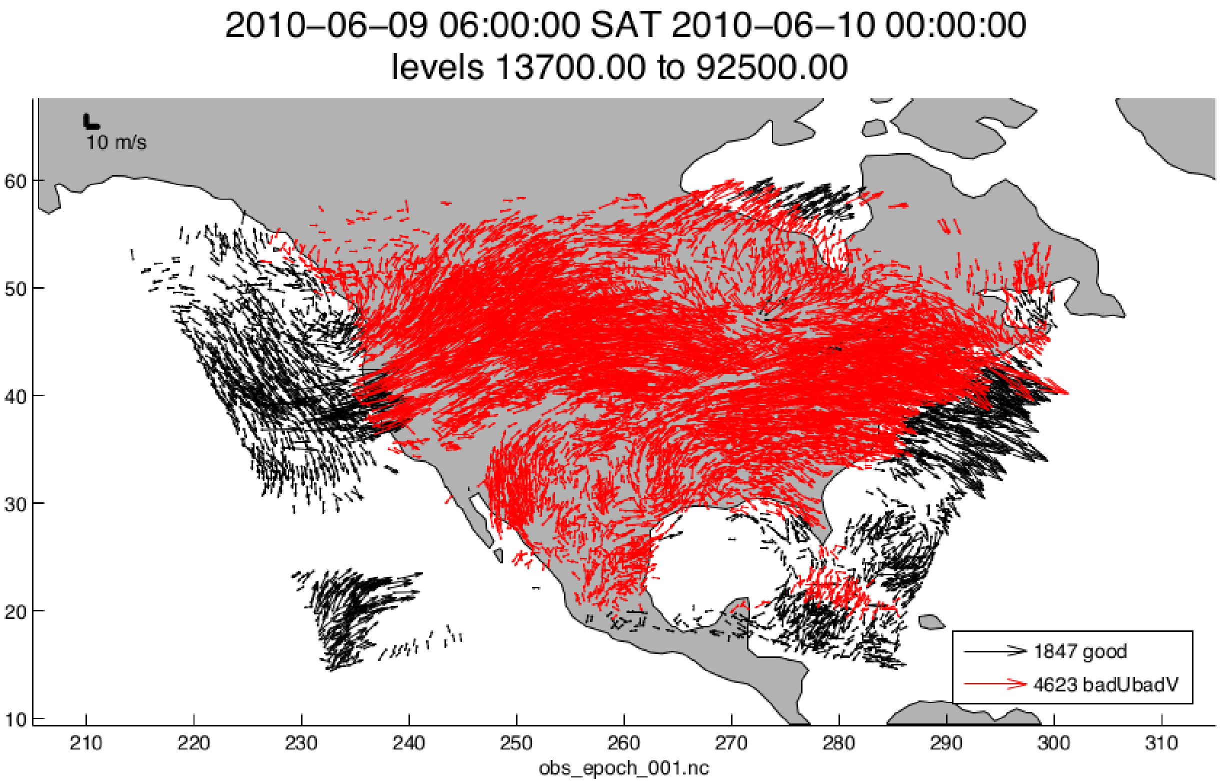
plot_obs_netcdf_diffs.m creates a 3D scatterplot of the difference between two 'copies' of an observation.

```
fname          = 'POP11/obs_epoch_011.nc';
region         = [0 360 -90 90 -Inf Inf];
ObsTypeString = 'AIRCRAFT_U_WIND_COMPONENT';
CopyString1    = 'NCEP BUFR observation';
CopyString2    = 'prior ensemble mean';
QCString       = 'DART quality control';
maxQC          = 2;
verbose        = 1;    % > 0 means 'print summary to command window'
twoup          = 0;    % > 0 means 'use same Figure for QC plot'
bob = plot_obs_netcdf_diffs(fname, ObsTypeString, region, CopyString1, CopyString2, ..
↪.
                            QCString, maxQC, verbose, twoup);
```

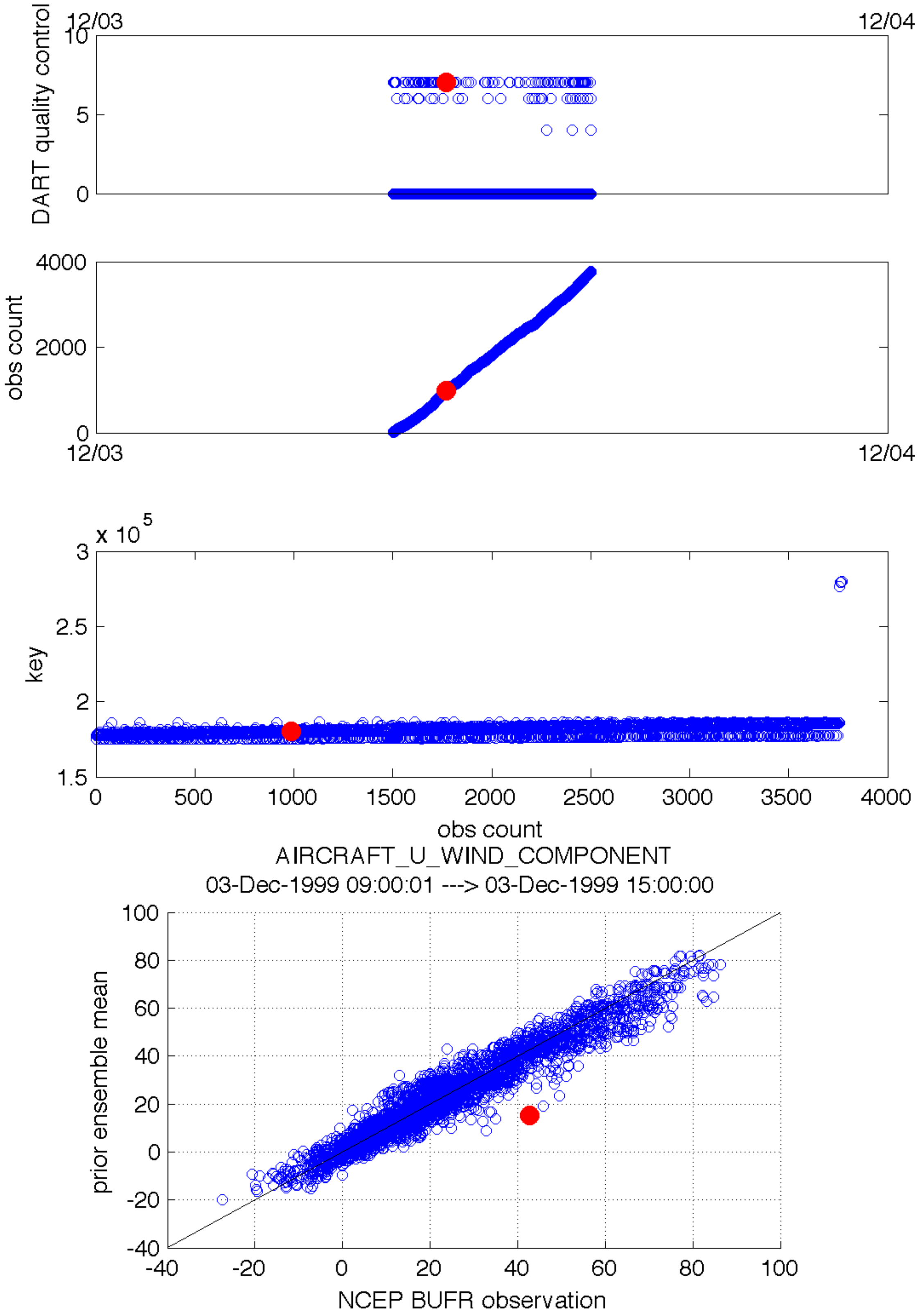


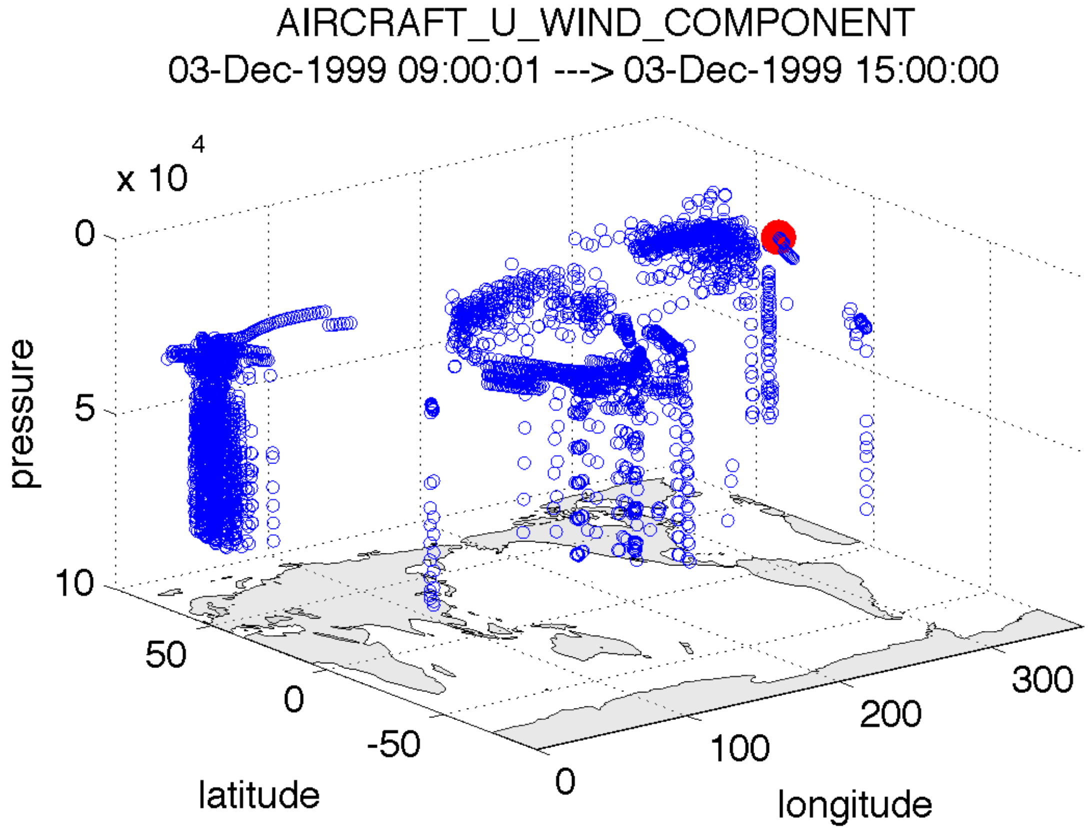
plot_wind_vectors.m creates a 2D 'quiver' plot of a wind field. This function is in the `matlab/private` directory - but if you want to use it, you can move it out. I find it has very little practical value.

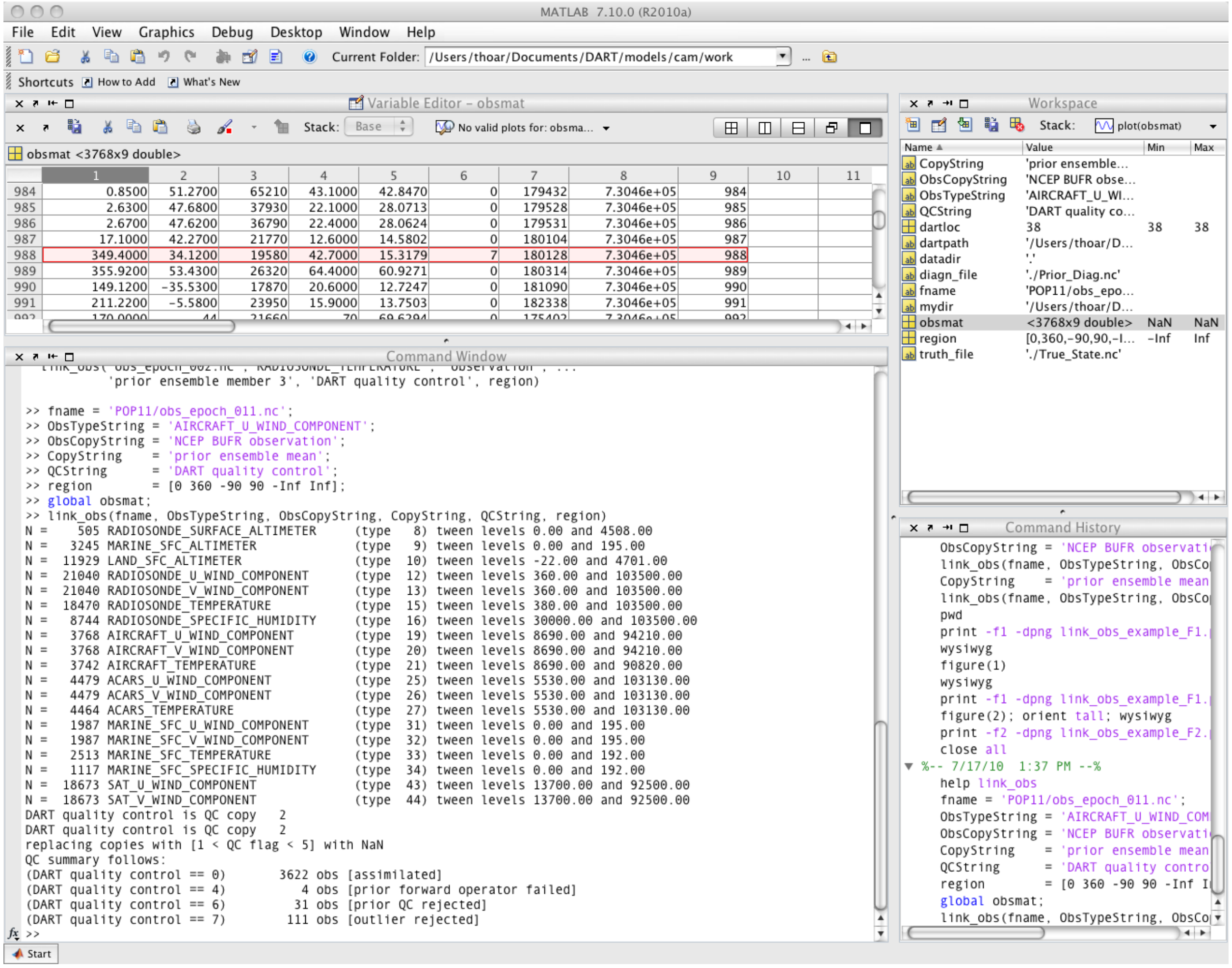
```
fname        = 'obs_epoch_001.nc';
platform     = 'SAT';     % usually 'RADIOSONDE', 'SAT', 'METAR', ...
CopyString   = 'NCEP BUFR observation';
QCString     = 'DART quality control';
region       = [210 310 12 65 -Inf Inf];
scalefactor = 5;      % reference arrow magnitude
bob = plot_wind_vectors(fname, platform, CopyString, QCString, ...
                        'region', region, 'scalefactor', scalefactor);
```



link_obs.m creates multiple figures that have linked attributes. This is my favorite function. Click on the little paintbrush icon in any of the figure frames and select all the observations with DART `QC == 4` in one window, and those same observations are highlighted in all the other windows (for example). The 3D scatterplot can be rotated around with the mouse to really pinpoint exactly where the observations are getting rejected, for example. All the images are links to larger versions - the image on the right has the MATLAB® call. If the data browser (the spreadsheet-like panel) is open, the selected observations get highlighted there too.


12/03
12/04
10
5
0
DART quality control
4000
2000
0
obs count
12/03
12/04
x 10^5
3
2.5
2
1.5
key
0
500
1000
1500
2000
2500
3000
3500
4000
obs count
AIRCRAFT_U_WIND_COMPONENT
03-Dec-1999 09:00:01 ---> 03-Dec-1999 15:00:00
100
80
60
40
20
0
-20
-40
prior ensemble mean
-40
-20
0
20
40
60
80
100
NCEP BUFR observation


AIRCRAFT_U_WIND_COMPONENT
03-Dec-1999 09:00:01 ---> 03-Dec-1999 15:00:00
x 10^4
0
5
10
pressure
50
0
-50
latitude
0
100
200
300
longitude


MATLAB 7.10.0 (R2010a)
File Edit View Graphics Debug Desktop Window Help
Current Folder: /Users/thoar/Documents/DART/models/cam/work
Shortcuts How to Add What's New
Variable Editor – obsmat
Stack: Base
No valid plots for: obsma...
Workspace
Stack: plot(obsmat)
Command Window
Command History
Start

obsmat <3768x9 double>

	1	2	3	4	5	6	7	8	9	10	11
984	0.8500	51.2700	65210	43.1000	42.8470	0	179432	7.3046e+05	984		
985	2.6300	47.6800	37930	22.1000	28.0713	0	179528	7.3046e+05	985		
986	2.6700	47.6200	36790	22.4000	28.0624	0	179531	7.3046e+05	986		
987	17.1000	42.2700	21770	12.6000	14.5802	0	180104	7.3046e+05	987		
988	349.4000	34.1200	19580	42.7000	15.3179	7	180128	7.3046e+05	988		
989	355.9200	53.4300	26320	64.4000	60.9271	0	180314	7.3046e+05	989		
990	149.1200	-35.5300	17870	20.6000	12.7247	0	181090	7.3046e+05	990		
991	211.2200	-5.5800	23950	15.9000	13.7503	0	182338	7.3046e+05	991		
992	170.0000	44	21660	70	69.6294	0	175402	7.3046e+05	992		

Name ▲	Value	Min	Max
CopyString	'prior ensemble...		
ObsCopyString	'NCEP BUFR obse...		
ObsTypeString	'AIRCRAFT_U_WI...		
QCString	'DART quality co...		
dartloc	38	38	38
dartpath	'/Users/thoar/D...		
datadir	'.'		
diagn_file	'./Prior_Diag.nc'		
fname	'POP11/obs_epo...		
mydir	'/Users/thoar/D...		
obsmat	<3768x9 double>	NaN	NaN
region	[0,360,-90,90,-I...	-Inf	Inf
truth_file	'./True_State.nc'		

```
link_obs('obs_epoch_002.nc', 'RADIOSONDE_TEMPERATURE', 'observation', ...
        'prior ensemble member 3', 'DART quality control', region)

>> fname = 'POP11/obs_epoch_011.nc';
>> ObsTypeString = 'AIRCRAFT_U_WIND_COMPONENT';
>> ObsCopyString = 'NCEP BUFR observation';
>> CopyString    = 'prior ensemble mean';
>> QCString      = 'DART quality control';
>> region        = [0 360 -90 90 -Inf Inf];
>> global obsmat;
>> link_obs(fname, ObsTypeString, ObsCopyString, CopyString, QCString, region)
N =    505 RADIOSONDE_SURFACE_ALTIMETER     (type   8) tween levels 0.00 and 4508.00
N =   3245 MARINE_SFC_ALTIMETER             (type   9) tween levels 0.00 and 195.00
N =  11929 LAND_SFC_ALTIMETER               (type  10) tween levels -22.00 and 4701.00
N =  21040 RADIOSONDE_U_WIND_COMPONENT      (type  12) tween levels 360.00 and 103500.00
N =  21040 RADIOSONDE_V_WIND_COMPONENT      (type  13) tween levels 360.00 and 103500.00
N =  18470 RADIOSONDE_TEMPERATURE           (type  15) tween levels 380.00 and 103500.00
N =   8744 RADIOSONDE_SPECIFIC_HUMIDITY     (type  16) tween levels 30000.00 and 103500.00
N =   3768 AIRCRAFT_U_WIND_COMPONENT        (type  19) tween levels 8690.00 and 94210.00
N =   3768 AIRCRAFT_V_WIND_COMPONENT        (type  20) tween levels 8690.00 and 94210.00
N =   3742 AIRCRAFT_TEMPERATURE             (type  21) tween levels 8690.00 and 90820.00
N =   4479 ACARS_U_WIND_COMPONENT           (type  25) tween levels 5530.00 and 103130.00
N =   4479 ACARS_V_WIND_COMPONENT           (type  26) tween levels 5530.00 and 103130.00
N =   4464 ACARS_TEMPERATURE                (type  27) tween levels 5530.00 and 103130.00
N =   1987 MARINE_SFC_U_WIND_COMPONENT      (type  31) tween levels 0.00 and 195.00
N =   1987 MARINE_SFC_V_WIND_COMPONENT      (type  32) tween levels 0.00 and 195.00
N =   2513 MARINE_SFC_TEMPERATURE           (type  33) tween levels 0.00 and 192.00
N =   1117 MARINE_SFC_SPECIFIC_HUMIDITY     (type  34) tween levels 0.00 and 192.00
N =  18673 SAT_U_WIND_COMPONENT             (type  43) tween levels 13700.00 and 92500.00
N =  18673 SAT_V_WIND_COMPONENT             (type  44) tween levels 13700.00 and 92500.00
DART quality control is QC copy   2
DART quality control is QC copy   2
replacing copies with [1 < QC flag < 5] with NaN
QC summary follows:
(DART quality control == 0)       3622 obs [assimilated]
(DART quality control == 4)          4 obs [prior forward operator failed]
(DART quality control == 6)         31 obs [prior QC rejected]
(DART quality control == 7)        111 obs [outlier rejected]
fx >>
```

```
   ObsCopyString = 'NCEP BUFR observati
   link_obs(fname, ObsTypeString, ObsCo
   CopyString    = 'prior ensemble mean
   link_obs(fname, ObsTypeString, ObsCo
   pwd
   print -f1 -dpng link_obs_example_F1.
   wysiwyg
   figure(1)
   wysiwyg
   print -f1 -dpng link_obs_example_F1.
   figure(2); orient tall; wysiwyg
   print -f2 -dpng link_obs_example_F2.
   close all
▼ %-- 7/17/10  1:37 PM --%
   help link_obs
   fname = 'POP11/obs_epoch_011.nc';
   ObsTypeString = 'AIRCRAFT_U_WIND_COM
   ObsCopyString = 'NCEP BUFR observati
   CopyString    = 'prior ensemble mean
   QCString      = 'DART quality contro
   region        = [0 360 -90 90 -Inf I
   global obsmat;
   link_obs(fname, ObsTypeString, ObsCo
```

6.13 Manhattan

6.13.1 DART Manhattan release documentation

6.13.2 DART overview

The Data Assimilation Research Testbed (DART) is designed to facilitate the combination of assimilation algorithms, models, and real (or synthetic) observations to allow increased understanding of all three. The DART programs are highly portable, having been compiled with many Fortran 90 compilers and run on linux compute-servers, linux clusters, OSX laptops/desktops, SGI Altix clusters, supercomputers running AIX, and more. Read the Customizations section for help in building on new platforms.

DART employs a modular programming approach to apply an Ensemble Kalman Filter which adjusts model values toward a state that is more consistent with information from a set of observations. Models may be swapped in and out, as can different algorithms in the Ensemble Kalman Filter. The method requires running multiple instances of a model to generate an ensemble of states. A forward operator appropriate for the type of observation being assimilated is applied to each of the states to generate the model's estimate of the observation. Comparing these estimates and their uncertainty to the observation and its uncertainty ultimately results in the adjustments to the model states. See the *DART_LAB Tutorial* demos or read more *DART Tutorial*.

DART diagnostic output can be written that contains the model state before and after the adjustment, along with the ensemble mean and standard deviation, and prior or posterior inflation values if inflation is enabled. There is also a text file, `obs_seq.final`, with the model estimates of the observations. There is a suite of MATLAB® functions that facilitate exploration of the results, but the netCDF files are inherently portable and contain all the necessary metadata to interpret the contents with other analysis programs such as NCL, R, etc.

To get started running with Lorenz 63 model refer to *DART Manhattan Release Notes*.

6.13.3 Notes for current users

If you have been updating from the rma_trunk branch of the DART subversion repository you will notice that the code tree has been simplified to be more intuitive for users. The new top level directory structure looks like :

- `README`
- `COPYRIGHT`
- *assimilation_code*
- *build_templates*
- *diagnostics*
- *documentation*
- *models*
- *observations*

if you do try to do an 'svn update' on an existing directory, you will encounter many 'tree conflicts'.

We suggest that current users checkout a fresh version of Manhattan in a new location. To see which files need to be moved, run 'svn status' on your original checked out version. Anything with an M or ? in the first column needs to be moved to the new location in the new tree. Please contact DART if you have any issues migrating your existing code to the new tree structure.

There is a list of non-backwards compatible changes (see below), and a list of new options and functions.

The Manhattan release will continue to be updated for the next few months as we continue to add features. Checking out the Manhattan release branch and running 'svn update' from time to time is the recommended way to update your DART tree.

6.13.4 Non-backwards compatible changes

Unlike previous releases of DART, this version contains more non-backwards compatible changes than usual. Please examine the following list carefully. We do suggest you check out the Manhattan release into a new location and migrate any local changes from previous versions as a second step.

Changes in the Manhattan release (15 May 2015) which are *not* backwards compatible with the Lanai release (13 Dec 2013):

1. We no longer require model data to be converted to DART format restart files. We directly read and write NetCDF format only. To specify the input and output files for filter, there are new namelist items in the &filter_nml namelist: `'input_state_file_list'` and `'output_state_file_list'` .
2. The information formerly in `Prior_Diag.nc` and `Posterior_Diag.nc` has been moved. If you are reading and writing ensemble members from different files, the state information, the ensemble mean and standard deviation, and the inflation mean and standard deviation will all be read and written to separate files:
   - `[stage]_member_####.nc`
   - `[stage]_mean.nc`

- [stage]_sd.nc
- [stage]_priorinf_{mean,sd}.nc (if prior inflation is turned on)
- [stage]_postinf_{mean,sd}.nc (if posterior inflation is turned on)

If you are reading and writing ensemble members from a single file, all this information will now be in a single NetCDF file but will be stored in different variables inside that file:

- [var].nc
- [var]_mean.nc
- [var]_sd.nc
- [var]_priorinf_{mean,sd}.nc (if prior inflation is turned on)
- [var]_postinf_{mean,sd}.nc (if posterior inflation is turned on)

We also now have options for writing files at four stages of the assimilation cycle: 'input', 'preassim', 'postassim', 'output'. This is set in the &filter_nml namelist with stages_to_write.

3. New model_mod.f90 required routines:

   - vert_convert()
   - query_vert_localization_coord()
   - pert_model_copies()
   - read_model_time()
   - write_model_time()

   There are default version of these available to use if you have no special requirements.

4. Several of the model_mod.f90 argument lists have changed

   - model_interpolate() now takes in the state_handle as an argument rather than a state vector array. It also return an array of expected_obs and istatus for each of the ensemble members
   - get_state_meta_data() also requires the state_handle as an argument rather than a state vector array.
   - nc_write_model_atts() has an additional argument moel_mod_writes_state_variables. If true then the model_mod is expected to write out the state variables, if false DART will write out the state variable (this is the prefered method for adding new models, it requires less code from the model developer)

5. There are several namelist changes mainly in the &filter_nml and &perfect_model_mod which are outlined in detail in *DART Manhattan Differences from Lanai Release Notes*

6. All modules have been moved to *DART/assimilation_code/modules/* directory. And similarly all of the programs have moved to *DART/assimilation_code/programs/*

7. The location modules which were stored in *locations* have moved to *DART/assimilation_code/location* directory

8. The observation converters which were stored in *observations* have moved to *DART/observations/obs_converters* directory

9. The forward operators have moved from *obs_def/obs_def_*_mod.f90* to *observations/forward_operators*

10. The tutorial files have moved to *DART/docs/tutorial directory*

11. The program fill_inflation_restart is OBSOLETE since DART inflation files are now in NetCDF format. Now inflation files can be filled using ncap2. Here is an example using version 4.4.2 or later of the NCO tools:

```
ncap2 -s "T=1.0;U=1.0;V=1.0" wrfinput_d01 prior_inf.nc'
ncap2 -s "T=0.6;U=0.6;V=0.6" wrfinput_d01 prior_sd.nc'
```

12. The default flags in the mkmf_template.XXX files have been updated to be more consistent with current compiler versions.
13. If you enable the sampling error correction option, the required data is now read from a single netcdf file which supports multiple ensemble sizes. A program is provided to compute additional ensemble sizes if they are not in the default file.
14. Our use of TYPES and KINDS has been very confusing in the past. In Manhattan we have tried to make it clearer which things in DART are generic quantities (QTY) - temperature, pressure, etc - and which things are specific types of observations - Radiosonde_temperature, Argo_salinity etc.

    Below is a mapping between old and new subroutine names here for reference. We have made these changes to all files distributed with DART. If you have lots of code developed outside of the subversion repository, please contact DART for a sed script to help automate the changes.

    Public subroutines, existing name on left, replacement on right:

```
assimilate_this_obs_kind()     =>     assimilate_this_type_of_obs(type_index)
evaluate_this_obs_kind()       =>       evaluate_this_type_of_obs(type_index)
use_ext_prior_this_obs_kind()  =>  use_ext_prior_this_type_of_obs(type_index)

get_num_obs_kinds()            =>  get_num_types_of_obs()
get_num_raw_obs_kinds()        =>  get_num_quantities()

get_obs_kind_index()           => get_index_for_type_of_obs(type_name)
get_obs_kind_name()            => get_name_for_type_of_obs(type_index)

get_raw_obs_kind_index()       =>  get_index_for_quantity(qty_name)
get_raw_obs_kind_name()        =>  get_name_for_quantity(qty_index)

get_obs_kind_var_type()        =>  get_quantity_for_type_of_obs(type_index)

get_obs_kind()                 =>  get_obs_def_type_of_obs(obs_def)
set_obs_def_kind()             =>  set_obs_def_type_of_obs(obs_def)

get_kind_from_menu()           =>  get_type_of_obs_from_menu()

read_obs_kind()                =>   read_type_of_obs_table(file_unit, file_format)
write_obs_kind()               =>  write_type_of_obs_table(file_unit, file_format)

maps obs_seq nums to specific type nums, only used in read_obs_seq:
map_def_index()                => map_type_of_obs_table()

removed this.  apparently unused, and simply calls get_obs_kind_name():
get_obs_name()

apparently unused anywhere, removed:
add_wind_names()
do_obs_form_pair()
```

    Public integer parameter constants and subroutine formal argument names, old on left, new on right:

```
KIND_ => QTY_
kind  => quantity
```

(continues on next page)

(continued from previous page)

```
TYPE_ => TYPE_
type  => type_of_obs

integer parameters:
max_obs_generic  =>  max_defined_quantities  (not currently public, stays private)
max_obs_kinds    =>  max_defined_types_of_obs
```

15. For smaller models we support single file input and output. These files contain all of the member information, mean, standard deviation and inflation values for all of the state variables. This can be run with cycling and all time steps will be appended to the file.

    For `perfect_model_obs` we provide a `perfect_input.cdl` file which contains a single ensemble member which will be considered the 'truth' and observations will be generated based on those values. The output will contain all of the cycling timesteps all of the state variables.

    For `filter` we provide a `filter_input.cdl` file which contains all of the state member variables and potentially inflation mean and standard deviation values. The output will contain all of the cycling timesteps all of the state variables. Additionally you have the option to write out different stages during the assimilation in the &filter_nml `stages_to_write` mentioned above.

    To generate a NetCDF file from a .cdl file run:

    ```
    ncgen -o perfect_input.nc perfect_input.cdl
    ncgen -o filter_input.nc filter_input.cdl
    ```

6.13.5 New features

- DART now reads and writes NetCDF files for the model state information. If your model uses NetCDF file format, you no longer need model_to_dart or dart_to_model to translate to a DART format file. If your model does not use NetCDF, you can adapt your model_to_dart and dart_to_model executables to read and write a NetCDF file for DART to use. The read/write code is part of the core DART routines so no code is needed in the model_mod model-specific module. There is a new routine *State Stucture* that a model_mod::static_init_model() can user to define which NetCDF variables should be part of the model state, and what DART quantity (formerly kind) they correspond to.
- DART no longer limits the size of a model state to the size of a single MPI task's memory. The state is read in variable by variable and distributed across all MPI tasks, so the memory use is much smaller than previous versions of DART. One-sided MPI communication is used during the computation of forward operator values to get required parts of the state from other tasks.
- Many of the DART namelists have been simplified, and some items have moved to a more specific namelist.
- Observation sequence files can include externally computed forward operator values which can be used in the assimilation instead of calling a forward operator inside DART.
- The DART directory structure has been reorganized to make it easier to identify the various software tools, modules, documentation and tutorials supplied with the system.
- The MATLAB® diagnostic routines have been updated to not require the MEXNC toolbox. These routines use the built-in NetCDF support that comes with MATLAB®.
- There is a new Particle Filter type. Please contact us if you are interested in using it.
- DART can now take subsets of observation types and restrict them from impacting certain quantities in the state during the assimilation. A tool to simplify constructing the table of interactions is provided (obs_impact_tool).
- State Structure

  - Contains information about dimensions and size of variables in your state. There is a number of accessor functions to get variable information such as `get_variable_size()`. See the *State Stucture* for more details.

- The POP model_mod now can interpolate Sea Surface Anomaly observations.

6.13.6 Supported models

Currently we support the models listed below. There are several new models that have been added that are not on the Lanai Release including CM1, CICE, and ROMS. Any previously supported models not on this list are still supported in DART classic

- **9var**
  - DART interface documentation for the *9-variable* model.
- **bgrid_solo**
  - DART interface documentation for the *bgrid_solo* model.
- **cam-fv**
  - DART interface documentation for the *CAM-FV* global atmospheric model.
  - Documentation for the CAM model.
- **cice (NEW)**
  - DART interface documentation for the *CICE* model.
  - Documentation for the CICE model.
- **cm1 (NEW)**
  - DART interface documentation for the *CM1*.
  - Documentation for the CM1 model.
- **forced_lorenz_96**
  - DART interface documentation for the *Forced Lorenz 96* model.
- **lorenz_63**
  - DART interface documentation for the *Lorenz 63* model.
- **lorenz_84**
  - DART interface documentation for the *Lorenz 84* model.
- **lorenz_96**
  - DART interface documentation for the *Lorenz 96* model.
- **lorenz_04**
  - DART interface documentation for the *Lorenz 05* model.
- **mpas_atm** (NetCDF overwrite not supported for update_u_from_reconstruct = .true.)
  - DART interface documentation for the *MPAS_ATM* model.
  - Documentation for the MPAS model.
- **POP**
  - DART interface documentation for the *POP* global ocean model.

  - Documentation for the POP model.

- **ROMS (NEW)**
  - DART interface documentation for the *ROMS* regional ocean model.
  - Documentation for the ROMS model.
- **simple_advection**
  - DART interface documentation for the *Simple advection* model.
- **wrf**
  - DART interface documentation for the *WRF* regional forecast model.
  - Documentation for the WRF model.

The `DART/models/template` directory contains sample files for adding a new model. See the Adding a Model section of the DART web pages for more help on adding a new model.

6.13.7 Changed models

- WRF
  - Allow advanced microphysics schemes (needed interpolation for 7 new kinds)
  - Interpolation in the vertical is now done in log(p) instead of linear pressure space. log(p) is the default, but a compile-time variable can restore the linear interpolation.
  - Added support in the namelist to avoid writing updated fields back into the wrf netcdf files. The fields are still updated during the assimilation but the updated data is not written back to the wrfinput file during the dart_to_wrf step.
  - Fixed an obscure bug in the vertical convert routine of the wrf model_mod that would occasionally fail to convert an obs. This would make tiny differences in the output as the number of mpi tasks change. No quantitative differences in the results but they were not bitwise compatible before and they are again now.
- CAM
  - DART/CAM now runs under the CESM framework, so all options available with the framework can be used.
  - Support for the SE core (HOMME) has been developed but is NOT part of this release. Please contact the DART Development Group if you have an interest in this configuration of CAM.
- Simple Advection Model
  - Fixed a bug where the random number generator was being used before being called with an initial seed.

6.13.8 New observation types/forward operators

- Many new observation types related to land and atmospheric chemistry have been added. See the obs_kind_mod.f90 for a list of the generic quantities now available.
- New forward operator for Sea Ice (cice) ice thickness observations. See the obs_def_cice_mod.f90 file for details.
- New forward operator for Carbon Monoxide (CO) Nadir observations. See the obs_def_CO_Nadir_mod.f90 file for details.
- New forward operator for Total Cloud Water in a column observations. See the obs_def_cwp_mod.f90 file for details.

6.13.9 New observation types/sources

- AVISO Added an observation converter for Sea Surface Height Anomaly observations. Documentation in convert_aviso.f90 (source).
- cice Added an obs_sequence converter for Sea Ice observations. Documentation in *PROGRAM cice_to_obs*.
- GPSPW Added an obs_sequence converter for GPS precipitable water observations. Documentation in convert_gpspw.f90 (source).
- MODIS Added an obs_sequence converter for MODIS FPAR (Fraction of Photosynthetically Active Radiation) and LAI (Leaf Area Index) obseverations. Documentation in *PROGRAM MOD15A2_to_obs*.
- ok_mesonet Added an obs_sequence converter for the Oklahoma Mesonet observations. Documentation in *Oklahoma Mesonet MDF Data*.
- ROMS Added an obs_sequence converter for ROMS ocean data. This converter includes externally computed forward operators output from the ROMS model using FGAT (First Guess At Time) during the model run. Documentation in convert_roms_obs.f90 (source).
- SSUSI Added an obs_sequence converter for wind profiler observations. Documentation in *SSUSI F16 EDR-DSK format to observation sequence converters*.
- tropical_cyclone Added an obs_sequence converter for ASCII format tropical cyclone track observations. Documentation in *PROGRAM tc_to_obs*.

6.13.10 New diagnostics and documentation

Better Web Pages. We've put a lot of effort into expanding our documentation. For example, please check out the MATLAB diagnostics section or the pages outlining the observation sequence file contents.

- The MATLAB® diagnostic routines have been updated to remove the dependency on third-party toolboxes. These routines use the built-in netCDF support that comes with basic MATLAB® (no other toolboxes needed).

But there's always more to add. **Please let us know where we are lacking.**

6.13.11 New utilities

This section describes updates and changes to the tutorial materials, scripting, setup, and build information since the Lanai release.

- `obs_impact_tool` please refer to Website or *PROGRAM obs_impact_tool*
- `gen_sampling_error_table` now computes sampling error correction tables for any ensemble size.
- `compute_error` Website or *PROGRAM compute_error*

6.13.12 Known problems

There are many changes in this release and more updates are expected to come soon. We are not aware of any obvious bugs, but if you encounter any unexpected behavior please contact us. Please watch the dart-users email list for announcements of updates to the release code, and be prepared to do an 'svn update' from time to time to get updated files.

6.14 Lanai

6.14.1 DART Lanai release documentation

Attention: Lanai is a prior release of DART. Its source code is available via the DART repository on Github. This documentation is preserved merely for reference. See the DART homepage for information on the latest release.

6.14.2 Dart overview

The Data Assimilation Research Testbed (DART) is designed to facilitate the combination of assimilation algorithms, models, and real (or synthetic) observations to allow increased understanding of all three. The DART programs are highly portable, having been compiled with many Fortran 90 compilers and run on linux compute-servers, linux clusters, OSX laptops/desktops, SGI Altix clusters, supercomputers running AIX, and more. Read the Customizations section for help in building on new platforms.

DART employs a modular programming approach to apply an Ensemble Kalman Filter which adjusts model values toward a state that is more consistent with information from a set of observations. Models may be swapped in and out, as can different algorithms in the Ensemble Kalman Filter. The method requires running multiple instances of a model to generate an ensemble of states. A forward operator appropriate for the type of observation being assimilated is applied to each of the states to generate the model's estimate of the observation. Comparing these estimates and their uncertainty to the observation and its uncertainty ultimately results in the adjustments to the model states. See the DARTLAB demos or read more in the tutorials included with the DART distribution. They are described below.

DART diagnostic output includes two netCDF files containing the model states just before the adjustment (`Prior_Diag.nc`) and just after the adjustment (`Posterior_Diag.nc`) as well as a file `obs_seq.final` with the model estimates of the observations. There is a suite of Matlab® functions that facilitate exploration of the results, but the netCDF files are inherently portable and contain all the necessary metadata to interpret the contents with other analysis programs such as NCL, R, etc.

In this document links are available which point to Web-based documentation files and also to the same information in html files distributed with DART. If you have used subversion to check out a local copy of the DART files you can open this file in a browser by loading `DART/docs/html/Lanai_release.html` and then use the `local file` links to see other documentation pages without requiring a connection to the internet. If you are looking at this documentation from the `www.image.ucar.edu` web server or you are connected to the internet you can use the `Website` links to view other documentation pages.

6.14.3 Getting started

What's required

1. a Fortran 90 compiler
2. a netCDF library including the F90 interfaces
3. the C shell
4. (optional, to run in parallel) an MPI library

DART has been tested on many Fortran compilers and platforms. We don't have any platform-dependent code sections and we use only the parts of the language that are portable across all the compilers we have access to. We explicitly set the Fortran 'kind' for all real values and do not rely on autopromotion or other compile-time flags to set the default byte size for numbers. It is possible that some model-specific interface code from outside sources may have specific compiler flag requirements; see the documentation for each model. The low-order models and all common portions of the DART code compile cleanly.

DART uses the netCDF self-describing data format with a particular metadata convention to describe output that is used to analyze the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools.

Since most of the models being used with DART are written in Fortran and run on various UNIX or *nix platforms, the development environment for DART is highly skewed to these machines. We do most of our development on a small linux workstation and a mac laptop running OSX 10.x, and we have an extensive test network. (I've never built nor run DART on a Windows machine - so I don't even know if it's possible. If you have run it (under Cygwin?) please let me know how it went – I'm curious. Tim - thoar 'at' ucar 'dot ' edu)

What's nice to have

- **ncview**: DART users have used ncview to create graphical displays of output data fields. The 2D rendering is good for 'quick-look' type uses, but I wouldn't want to publish with it.
- **NCO**: The NCO tools are able to perform operations on netCDF files like concatenating, slicing, and dicing.
- **Matlab®**: A set of Matlab® scripts designed to produce graphical diagnostics from DART. netCDF output files are also part of the DART project.
- **MPI**: The DART system includes an MPI option. MPI stands for 'Message Passing Interface', and is both a library and run-time system that enables multiple copies of a single program to run in parallel, exchange data, and combine to solve a problem more quickly. DART does **NOT** require MPI to run; the default build scripts do not need nor use MPI in any way. However, for larger models with large state vectors and large numbers of observations, the data assimilation step will run much faster in parallel, which requires MPI to be installed and used. However, if multiple ensembles of your model fit comfortably (in time and memory space) on a single processor, you need read no further about MPI.

Types of input

DART programs can require three different types of input. First, some of the DART programs, like those for creating synthetic observational datasets, require interactive input from the keyboard. For simple cases this interactive input can be made directly from the keyboard. In more complicated cases a file containing the appropriate keyboard input can be created and this file can be directed to the standard input of the DART program. Second, many DART programs expect one or more input files in DART specific formats to be available. For instance, `perfect_model_obs`, which creates a synthetic observation set given a particular model and a description of a sequence of observations, requires an input file that describes this observation sequence. At present, the observation files for DART are in a custom format in either human-readable ascii or more compact machine-specific binary. Third, many DART modules (including main programs) make use of the Fortran90 namelist facility to obtain values of certain parameters at run-time. All programs look for a namelist input file called `input.nml` in the directory in which the program is executed. The `input.nml` file can contain a sequence of individual Fortran90 namelists which specify values of particular parameters for modules that compose the executable program.

6.14.4 Installation

This document outlines the installation of the DART software and the system requirements. The entire installation process is summarized in the following steps:

1. Determine which F90 compiler is available.
2. Determine the location of the `netCDF` library.
3. Download the DART software into the expected source tree.
4. Modify certain DART files to reflect the available F90 compiler and location of the appropriate libraries.
5. Build the executables.

We have tried to make the code as portable as possible, but we do not have access to all compilers on all platforms, so there are no guarantees. We are interested in your experience building the system, so please email me (Tim Hoar) thoar 'at' ucar 'dot' edu (trying to cut down on the spam).

After the installation, you might want to peruse the following.

- Running the Lorenz_63 Model.
- Using the Matlab® diagnostic scripts.
- A short discussion on bias, filter divergence and covariance inflation.
- And another one on synthetic observations.

You should *absolutely* run the DARTLAB interactive tutorial (if you have Matlab available) and look at the DARTLAB presentation slides Website or *DART_LAB Tutorial* in the `DART_LAB` directory, and then take the tutorial in the `DART/tutorial` directory.

Requirements: an F90 compiler

The DART software has been successfully built on many Linux, OS/X, and supercomputer platforms with compilers that include GNU Fortran Compiler ("gfortran") (free), Intel Fortran Compiler for Linux and Mac OS/X, Portland Group Fortran Compiler, Lahey Fortran Compiler, Pathscale Fortran Compiler, and the Cray native compiler. Since recompiling the code is a necessity to experiment with different models, there are no binaries to distribute.

DART uses the netCDF self-describing data format for the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. In particular, DART also makes use of the F90 interface to the library which is available through the `netcdf.mod` and `typesizes.mod` modules. *IMPORTANT*: different compilers create these modules with different "case" filenames, and sometimes they are not

both installed into the expected directory. It is required that both modules be present. The normal place would be in the netcdf/include directory, as opposed to the netcdf/lib directory.

If the netCDF library does not exist on your system, you must build it (as well as the F90 interface modules). The library and instructions for building the library or installing from an RPM may be found at the netCDF home page: http://www.unidata.ucar.edu/packages/netcdf/

The location of the netCDF library, libnetcdf.a, and the locations of both netcdf.mod and typesizes.mod will be needed by the makefile template, as described in the compiling section. Depending on the netCDF build options, the Fortran 90 interfaces may be built in a separate library named netcdff.a and you may need to add -lnetcdff to the library flags.

6.14.5 Downloading the distribution

This release of the DART source code can be downloaded as a compressed zip or tar.gz file. When extracted, the source tree will begin with a directory named DART and will be approximately 175.3 Mb. Compiling the code in this tree (as is usually the case) will necessitate much more space.

```
$ gunzip DART-8.0.0.tar.gz
$ tar -xvf DART-8.0.0.tar
```

You should wind up with a directory named DART.

The code tree is very “bushy”; there are many directories of support routines, etc. but only a few directories involved with the customization and installation of the DART software. If you can compile and run ONE of the low-order models, you should be able to compile and run ANY of the low-order models. For this reason, we can focus on the Lorenz `63 model. Subsequently, the only directories with files to be modified to check the installation are: DART/mkmf, DART/models/lorenz_63/work, and DART/matlab (but only for analysis).

6.14.6 Customizing the build scripts – overview

DART executable programs are constructed using two tools: make and mkmf. The make utility is a very common piece of software that requires a user-defined input file that records dependencies between different source files. make then performs a hierarchy of actions when one or more of the source files is modified. The mkmf utility is a custom preprocessor that generates a make input file (named Makefile) and an example namelist *input.nml.program_default* with the default values. The Makefile is designed specifically to work with object-oriented Fortran90 (and other languages) for systems like DART.

mkmf requires two separate input files. The first is a `template’ file which specifies details of the commands required for a specific Fortran90 compiler and may also contain pointers to directories containing pre-compiled utilities required by the DART system. **This template file will need to be modified to reflect your system**. The second input file is a `path_names’ file which includes a complete list of the locations (either relative or absolute) of all Fortran90 source files that are required to produce a particular DART program. Each ‘path_names’ file must contain a path for exactly one Fortran90 file containing a main program, but may contain any number of additional paths pointing to files containing Fortran90 modules. An mkmf command is executed which uses the ‘path_names’ file and the mkmf template file to produce a Makefile which is subsequently used by the standard make utility.

Shell scripts that execute the mkmf command for all standard DART executables are provided as part of the standard DART software. For more information on mkmf see the FMS mkmf description.

One of the benefits of using mkmf is that it also creates an example namelist file for each program. The example namelist is called *input.nml.program_default*, so as not to clash with any exising input.nml that may exist in that directory.

Building and customizing the 'mkmf.template' file

A series of templates for different compilers/architectures exists in the DART/mkmf/ directory and have names with extensions that identify the compiler, the architecture, or both. This is how you inform the build process of the specifics of your system. Our intent is that you copy one that is similar to your system into mkmf.template and customize it. For the discussion that follows, knowledge of the contents of one of these templates (i.e. mkmf.template.gfortran) is needed. Note that only the LAST lines are shown here, the head of the file is just a big comment (worth reading, btw).

```
...
MPIFC = mpif90
MPILD = mpif90
FC = gfortran
LD = gfortran
NETCDF = /usr/local
INCS = ${NETCDF}/include
FFLAGS = -O2 -I$(INCS)
LIBS = -L${NETCDF}/lib -lnetcdf
LDFLAGS = -I$(INCS) $(LIBS)
```

Essentially, each of the lines defines some part of the resulting Makefile. Since make is particularly good at sorting out dependencies, the order of these lines really doesn't make any difference. The FC = gfortran line ultimately defines the Fortran90 compiler to use, etc. The lines which are most likely to need site-specific changes start with FFLAGS and NETCDF, which indicate where to look for the netCDF F90 modules and the location of the netCDF library and modules.

If you have MPI installed on your system MPIFC, MPILD dictate which compiler will be used in that instance. If you do not have MPI, these variables are of no consequence.

Netcdf

Modifying the NETCDF value should be relatively straightforward.

Change the string to reflect the location of your netCDF installation containing netcdf.mod and typesizes.mod. The value of the NETCDF variable will be used by the FFLAGS, LIBS, and LDFLAGS variables.

FFLAGS

Each compiler has different compile flags, so there is really no way to exhaustively cover this other than to say the templates as we supply them should work – depending on the location of your netCDF. The low-order models can be compiled without a -r8 switch, but the bgrid_solo model cannot.

Libs

The Fortran 90 interfaces may be part of the default netcdf.a library and -lnetcdf is all you need. However it is also common for the Fortran 90 interfaces to be built in a separate library named netcdff.a. In that case you will need -lnetcdf and also -lnetcdff on the **LIBS** line. This is a build-time option when the netCDF libraries are compiled so it varies from site to site.

Customizing the 'path_names_*' file

Several path_names_* files are provided in the work directory for each specific model, in this case: DART/models/lorenz_63/work. Since each model comes with its own set of files, the path_names_* files need no customization.

6.14.7 Building the Lorenz_63 DART project

DART executables are constructed in a work subdirectory under the directory containing code for the given model. From the top-level DART directory change to the L63 work directory and list the contents:

```
$ cd DART/models/lorenz_63/work
$ ls -1
```

With the result:

```
Posterior_Diag.nc
Prior_Diag.nc
True_State.nc
filter_ics
filter_restart
input.nml
mkmf_create_fixed_network_seq
mkmf_create_obs_sequence
mkmf_filter
mkmf_obs_diag
mkmf_obs_sequence_tool
mkmf_perfect_model_obs
mkmf_preprocess
mkmf_restart_file_tool
mkmf_wakeup_filter
obs_seq.final
obs_seq.in
obs_seq.out
obs_seq.out.average
obs_seq.out.x
obs_seq.out.xy
obs_seq.out.xyz
obs_seq.out.z
path_names_create_fixed_network_seq
path_names_create_obs_sequence
path_names_filter
path_names_obs_diag
path_names_obs_sequence_tool
path_names_perfect_model_obs
path_names_preprocess
path_names_restart_file_tool
path_names_wakeup_filter
perfect_ics
perfect_restart
quickbuild.csh
set_def.out
workshop_setup.csh
```

In all the work directories there will be a quickbuild.csh script that builds or rebuilds the executables. The following instructions do this work by hand to introduce you to the individual steps, but in practice running quickbuild will be the normal way to do the compiles.

There are nine mkmf_*xxxxxx* files for the programs

1. preprocess,
2. create_obs_sequence,
3. create_fixed_network_seq,
4. perfect_model_obs,
5. filter,
6. wakeup_filter,
7. obs_sequence_tool, and
8. restart_file_tool, and
9. obs_diag,

along with the corresponding path_names_*xxxxxx* files. There are also files that contain initial conditions, netCDF output, and several observation sequence files, all of which will be discussed later. You can examine the contents of one of the path_names_*xxxxxx* files, for instance path_names_filter, to see a list of the relative paths of all files that contain Fortran90 modules required for the program filter for the L63 model. All of these paths are relative to your DART directory. The first path is the main program (filter.f90) and is followed by all the Fortran90 modules used by this program (after preprocessing).

The mkmf_*xxxxxx* scripts are cryptic but should not need to be modified – as long as you do not restructure the code tree (by moving directories, for example). The function of the mkmf_*xxxxxx* script is to generate a Makefile and an *input.nml.program_default* file. It does not do the compile; make does that:

```
$ csh mkmf_preprocess
$ make
```

The first command generates an appropriate Makefile and the input.nml.preprocess_default file. The second command results in the compilation of a series of Fortran90 modules which ultimately produces an executable file: preprocess. Should you need to make any changes to the DART/mkmf/mkmf.template, you will need to regenerate the Makefile.

The preprocess program actually builds source code to be used by all the remaining modules. It is **imperative** to actually **run** preprocess before building the remaining executables. This is how the same code can assimilate state vector 'observations' for the Lorenz_63 model and real radar reflectivities for WRF without needing to specify a set of radar operators for the Lorenz_63 model!

preprocess reads the &preprocess_nml namelist to determine what observations and operators to incorporate. For this exercise, we will use the values in input.nml. preprocess is designed to abort if the files it is supposed to build already exist. For this reason, it is necessary to remove a couple files (if they exist) before you run the preprocessor. (The quickbuild.csh script will do this for you automatically.)

```
$ \rm -f ../../../obs_def/obs_def_mod.f90
$ \rm -f ../../../obs_kind/obs_kind_mod.f90
$ ./preprocess
$ ls -l ../../../obs_def/obs_def_mod.f90
$ ls -l ../../../obs_kind/obs_kind_mod.f90
```

This created ../../../obs_def/obs_def_mod.f90 from ../../../obs_kind/DEFAULT_obs_kind_mod.F90 and several other modules. ../../../obs_kind/obs_kind_mod.f90 was created similarly. Now we can build the rest of the project.

A series of object files for each module compiled will also be left in the work directory, as some of these are undoubtedly needed by the build of the other DART components. You can proceed to create the other programs needed to work with L63 in DART as follows:

```
$ csh mkmf_create_obs_sequence
$ make
$ csh mkmf_create_fixed_network_seq
$ make
$ csh mkmf_perfect_model_obs
$ make
$ csh mkmf_filter
$ make
$ csh mkmf_obs_diag
$ make
```

The result (hopefully) is that six executables now reside in your work directory. The most common problem is that the netCDF libraries and include files (particularly `typesizes.mod`) are not found. Edit the `DART/mkmf/mkmf.template`, recreate the `Makefile`, and try again.

program	purpose
`preprocess`	creates custom source code for just the observation types of interest
`create_obs_sequence`	specify a (set) of observation characteristics taken by a particular (set of) instruments
`create_fixed_network_seq`	repeat a set of observations through time to simulate observing networks where observations are taken in the same location at regular (or irregular) intervals
`perfect_model_obs`	generate "true state" for synthetic observation experiments. Can also be used to 'spin up' a model by running it for a long time.
`filter`	does the assimilation
`obs_diag`	creates observation-space diagnostic files to be explored by the Matlab® scripts.
`obs_sequence_tool`	manipulates observation sequence files. It is not generally needed (particularly for low-order models) but can be used to combine observation sequences or convert from ASCII to binary or vice-versa. We will not cover its use in this document.
`restart_file_tool`	manipulates the initial condition and restart files. We're going to ignore this one here.
`wakeup_filter`	is only needed for MPI applications. We're starting at the beginning here, so we're going to ignore this one, too.

6.14.8 Running Lorenz_63

This initial sequence of exercises includes detailed instructions on how to work with the DART code and allows investigation of the basic features of one of the most famous dynamical systems, the 3-variable Lorenz-63 model. The remarkable complexity of this simple model will also be used as a case study to introduce a number of features of a simple ensemble filter data assimilation system. To perform a synthetic observation assimilation experiment for the L63 model, the following steps must be performed (an overview of the process is given first, followed by detailed procedures for each step):

6.14.9 Experiment overview

1. Integrate the L63 model for a long time starting from arbitrary initial conditions to generate a model state that lies on the attractor. The ergodic nature of the L63 system means a 'lengthy' integration always converges to some point on the computer's finite precision representation of the model's attractor.
2. Generate a set of ensemble initial conditions from which to start an assimilation. Since L63 is ergodic, the ensemble members can be designed to look like random samples from the model's 'climatological distribution'. To generate an ensemble member, very small perturbations can be introduced to the state on the attractor generated by step 1. This perturbed state can then be integrated for a very long time until all memory of its initial condition can be viewed as forgotten. Any number of ensemble initial conditions can be generated by repeating this procedure.

3. Simulate a particular observing system by first creating an 'observation set definition' and then creating an 'observation sequence'. The 'observation set definition' describes the instrumental characteristics of the observations and the 'observation sequence' defines the temporal sequence of the observations.
4. Populate the 'observation sequence' with 'perfect' observations by integrating the model and using the information in the 'observation sequence' file to create simulated observations. This entails operating on the model state at the time of the observation with an appropriate forward operator (a function that operates on the model state vector to produce the expected value of the particular observation) and then adding a random sample from the observation error distribution specified in the observation set definition. At the same time, diagnostic output about the 'true' state trajectory can be created.
5. Assimilate the synthetic observations by running the filter; diagnostic output is generated.

1. Integrate the L63 model for a 'long' time

`perfect_model_obs` integrates the model for all the times specified in the 'observation sequence definition' file. To this end, begin by creating an 'observation sequence definition' file that spans a long time. Creating an 'observation sequence definition' file is a two-step procedure involving `create_obs_sequence` followed by `create_fixed_network_seq`. After they are both run, it is necessary to integrate the model with `perfect_model_obs`.

1.1 Create an observation set definition

`create_obs_sequence` creates an observation set definition, the time-independent part of an observation sequence. An observation set definition file only contains the `location, type,` and `observational error characteristics` (normally just the diagonal observational error variance) for a related set of observations. There are no actual observations, nor are there any times associated with the definition. For spin-up, we are only interested in integrating the L63 model, not in generating any particular synthetic observations. Begin by creating a minimal observation set definition.

In general, for the low-order models, only a single observation set need be defined. Next, the number of individual scalar observations (like a single surface pressure observation) in the set is needed. To spin-up an initial condition for the L63 model, only a single observation is needed. Next, the error variance for this observation must be entered. Since we do not need (nor want) this observation to have any impact on an assimilation (it will only be used for spinning up the model and the ensemble), enter a very large value for the error variance. An observation with a very large error variance has essentially no impact on deterministic filter assimilations like the default variety implemented in DART. Finally, the location and type of the observation need to be defined. For all types of models, the most elementary form of synthetic observations are called 'identity' observations. These observations are generated simply by adding a random sample from a specified observational error distribution directly to the value of one of the state variables. This defines the observation as being an identity observation of the first state variable in the L63 model. The program will respond by terminating after generating a file (generally named `set_def.out`) that defines the single identity observation of the first state variable of the L63 model. The following is a screenshot (much of the verbose logging has been left off for clarity), the user input looks *like this*.

```
[unixprompt]$ ./create_obs_sequence
 Starting program create_obs_sequence
 Initializing the utilities module.
 Trying to log to unit   10
 Trying to open file dart_log.out

 Registering module :
 $url: http://squish/DART/trunk/utilities/utilities_mod.f90 $
 $revision: 2713 $
 $date: 2007-03-25 22:09:04 -0600 (Sun, 25 Mar 2007) $
```

(continues on next page)

(continued from previous page)

```
 Registration complete.

 &UTILITIES_NML
 TERMLEVEL= 2,LOGFILENAME=dart_log.out

 /

 Registering module :
 $url: http://squish/DART/trunk/obs_sequence/create_obs_sequence.f90 $
 $revision: 2713 $
 $date: 2007-03-25 22:09:04 -0600 (Sun, 25 Mar 2007) $
 Registration complete.

 { ... }

 Input upper bound on number of observations in sequence
10

 Input number of copies of data (0 for just a definition)
0

 Input number of quality control values per field (0 or greater)
0

 input a -1 if there are no more obs
0

 Registering module :
 $url: http://squish/DART/trunk/obs_def/DEFAULT_obs_def_mod.F90 $
 $revision: 2820 $
 $date: 2007-04-09 10:37:47 -0600 (Mon, 09 Apr 2007) $
 Registration complete.


 Registering module :
 $url: http://squish/DART/trunk/obs_kind/DEFAULT_obs_kind_mod.F90 $
 $revision: 2822 $
 $date: 2007-04-09 10:39:08 -0600 (Mon, 09 Apr 2007) $
 Registration complete.

 ------------------------------------------------------

 initialize_module obs_kind_nml values are

 -------------- ASSIMILATE_THESE_OBS_TYPES --------------
 RAW_STATE_VARIABLE
 -------------- EVALUATE_THESE_OBS_TYPES --------------
 ------------------------------------------------------

      Input -1 * state variable index for identity observations
      OR input the name of the observation kind from table below:
      OR input the integer index, BUT see documentation...
        1 RAW_STATE_VARIABLE

-1

 input time in days and seconds
```

(continues on next page)

(continued from previous page)

```
1 0

 Input error variance for this observation definition
1000000

 input a -1 if there are no more obs
-1

 Input filename for sequence (  set_def.out   usually works well)
 set_def.out
 write_obs_seq  opening formatted file set_def.out
 write_obs_seq  closed file set_def.out
```

1.2 Create an observation sequence definition

`create_fixed_network_seq` creates an 'observation sequence definition' by extending the 'observation set definition' with the temporal attributes of the observations.

The first input is the name of the file created in the previous step, i.e. the name of the observation set definition that you've just created. It is possible to create sequences in which the observation sets are observed at regular intervals or irregularly in time. Here, all we need is a sequence that takes observations over a long period of time - indicated by entering a 1. Although the L63 system normally is defined as having a non-dimensional time step, the DART system arbitrarily defines the model timestep as being 3600 seconds. If we declare that we have one observation per day for 1000 days, we create an observation sequence definition spanning 24000 'model' timesteps; sufficient to spin-up the model onto the attractor. Finally, enter a name for the 'observation sequence definition' file. Note again: there are no observation values present in this file. Just an observation type, location, time and the error characteristics. We are going to populate the observation sequence with the `perfect_model_obs` program.

```
[unixprompt]$ ./create_fixed_network_seq

 ...

 Registering module :
 $url: http://squish/DART/trunk/obs_sequence/obs_sequence_mod.f90 $
 $revision: 2749 $
 $date: 2007-03-30 15:07:33 -0600 (Fri, 30 Mar 2007) $
 Registration complete.

 static_init_obs_sequence obs_sequence_nml values are
 &OBS_SEQUENCE_NML
 WRITE_BINARY_OBS_SEQUENCE =  F,
 /
 Input filename for network definition sequence (usually  set_def.out  )
set_def.out

 ...

 To input a regularly repeating time sequence enter 1
 To enter an irregular list of times enter 2
1
 Input number of observations in sequence
1000
 Input time of initial ob in sequence in days and seconds
1, 0
```

(continues on next page)

(continued from previous page)

```
 Input period of obs in days and seconds
1, 0
           1
           2
           3
...
         997
         998
         999
        1000
What is output file name for sequence (  obs_seq.in   is recommended )
obs_seq.in
 write_obs_seq  opening formatted file obs_seq.in
 write_obs_seq closed file obs_seq.in
```

1.3 Initialize the model onto the attractor

perfect_model_obs can now advance the arbitrary initial state for 24,000 timesteps to move it onto the attractor.

perfect_model_obs uses the Fortran90 namelist input mechanism instead of (admittedly gory, but temporary) interactive input. All of the DART software expects the namelists to found in a file called input.nml. When you built the executable, an example namelist was created input.nml.perfect_model_obs_default that contains all of the namelist input for the executable. If you followed the example, each namelist was saved to a unique name. We must now rename and edit the namelist file for perfect_model_obs. Copy input.nml.perfect_model_obs_default to input.nml and edit it to look like the following: (just worry about the highlighted stuff - and whitespace doesn't matter)

```
$ cp input.nml.perfect_model_obs_default input.nml
```

```
&perfect_model_obs_nml
   start_from_restart    = .false.,
   output_restart        = .true.,
   async                 = 0,
   init_time_days        = 0,
   init_time_seconds     = 0,
   first_obs_days        = -1,
   first_obs_seconds     = -1,
   last_obs_days         = -1,
   last_obs_seconds      = -1,
   output_interval       = 1,
   restart_in_file_name  = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   obs_seq_in_file_name  = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   adv_ens_command       = "./advance_ens.csh"  /

&ensemble_manager_nml
   single_restart_file_in  = .true.,
   single_restart_file_out = .true.,
   perturbation_amplitude  = 0.2  /

&assim_tools_nml
   filter_kind                     = 1,
   cutoff                          = 0.2,
```

(continues on next page)

(continued from previous page)

```
   sort_obs_inc                    = .false.,
   spread_restoration              = .false.,
   sampling_error_correction       = .false.,
   adaptive_localization_threshold = -1,
   print_every_nth_obs             = 0  /

&cov_cutoff_nml
   select_localization = 1  /

&reg_factor_nml
   select_regression    = 1,
   input_reg_file       = "time_mean_reg",
   save_reg_diagnostics = .false.,
   reg_diagnostics_file = "reg_diagnostics"  /

&obs_sequence_nml
   write_binary_obs_sequence = .false.  /

&obs_kind_nml
   assimilate_these_obs_types = 'RAW_STATE_VARIABLE'  /

&assim_model_nml
   write_binary_restart_files = .true. /

&model_nml
   sigma  = 10.0,
   r      = 28.0,
   b      = 2.6666666666667,
   deltat = 0.01,
   time_step_days = 0,
   time_step_seconds = 3600  /

&utilities_nml
   TERMLEVEL = 1,
   logfilename = 'dart_log.out'  /
```

For the moment, only two namelists warrant explanation. Each namelists is covered in detail in the html files accompanying the source code for the module.

perfect_model_obs_nml

namelist variable	description
start_from_restart	When set to 'false', perfect_model_obs generates an arbitrary initial condition (which cannot be guaranteed to be on the L63 attractor). When set to 'true', a restart file (specified by restart_in_file_name) is read.
output_restart	When set to 'true', perfect_model_obs will record the model state at the end of this integration in the file named by restart_out_file_name.
async	The lorenz_63 model is advanced through a subroutine call - indicated by async = 0. There is no other valid value for this model.
init_time_*xxx*	the start time of the integration.
first_obs_*xxx*	the time of the first observation of interest. While not needed in this example, you can skip observations if you want to. A value of -1 indicates to start at the beginning.
last_obs_*xxx*	the time of the last observation of interest. While not needed in this example, you do not have to assimilate all the way to the end of the observation sequence file. A value of -1 indicates to use all the observations.
output_interval	interval at which to save the model state (in True_State.nc).
restart_in_file_name	is ignored when 'start_from_restart' is 'false'.
restart_out_file_name	if output_restart is 'true', this specifies the name of the file containing the model state at the end of the integration.
obs_seq_in_file_name	specifies the file name that results from running create_fixed_network_seq, i.e. the 'observation sequence definition' file.
obs_seq_out_file_name	specifies the output file name containing the 'observation sequence', finally populated with (perfect?) 'observations'.
advance_ens_command	specifies the shell commands or script to execute when async /= 0.

utilities_nml

namelist variable	description
TERMLEVEL	When set to '1' the programs terminate when a 'warning' is generated. When set to '2' the programs terminate only with 'fatal' errors.
logfilename	Run-time diagnostics are saved to this file. This namelist is used by all programs, so the file is opened in APPEND mode. Subsequent executions cause this file to grow.

Executing perfect_model_obs will integrate the model 24,000 steps and output the resulting state in the file perfect_restart. Interested parties can check the spinup in the True_State.nc file.

```
$ ./perfect_model_obs
```

2. Generate a set of ensemble initial conditions

The set of initial conditions for a 'perfect model' experiment is created in several steps. 1) Starting from the spun-up state of the model (available in `perfect_restart`), run `perfect_model_obs` to generate the 'true state' of the experiment and a corresponding set of observations. 2) Feed the same initial spun-up state and resulting observations into `filter`.

The first step is achieved by changing a perfect_model_obs namelist parameter, copying `perfect_restart` to `perfect_ics`, and rerunning `perfect_model_obs`. This execution of `perfect_model_obs` will advance the model state from the end of the first 24,000 steps to the end of an additional 24,000 steps and place the final state in `perfect_restart`. The rest of the namelists in `input.nml` should remain unchanged.

```
&perfect_model_obs_nml
   start_from_restart    = .true.,
   output_restart        = .true.,
   async                 = 0,
   init_time_days        = 0,
   init_time_seconds     = 0,
   first_obs_days        = -1,
   first_obs_seconds     = -1,
   last_obs_days         = -1,
   last_obs_seconds      = -1,
   output_interval       = 1,
   restart_in_file_name  = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   obs_seq_in_file_name  = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   adv_ens_command       = "./advance_ens.csh"  /
```

```
$ cp perfect_restart perfect_ics
$ ./perfect_model_obs
```

A `True_State.nc` file is also created. It contains the 'true' state of the integration.

Generating the ensemble

This step (#2 from above) is done with the program `filter`, which also uses the Fortran90 namelist mechanism for input. It is now necessary to copy the `input.nml.filter_default` namelist to `input.nml`.

```
cp input.nml.filter_default input.nml
```

You may also build one master namelist containting all the required namelists. Having unused namelists in the `input.nml` does not hurt anything, and it has been so useful to be reminded of what is possible that we made it an error to NOT have a required namelist. Take a peek at any of the other models for examples of a "fully qualified" `input.nml`.

HINT: if you used `svn` to get the project, try 'svn revert input.nml' to restore the namelist that was distributed with the project - which DOES have all the namelist blocks. Just be sure the values match the examples here.

```
&filter_nml
   async                    = 0,
   adv_ens_command          = "./advance_model.csh",
   ens_size                 = 100,
   start_from_restart       = .false.,
   output_restart           = .true.,
   obs_sequence_in_name     = "obs_seq.out",
```

(continues on next page)

(continued from previous page)

```
   obs_sequence_out_name    = "obs_seq.final",
   restart_in_file_name     = "perfect_ics",
   restart_out_file_name    = "filter_restart",
   init_time_days           = 0,
   init_time_seconds        = 0,
   first_obs_days           = -1,
   first_obs_seconds        = -1,
   last_obs_days            = -1,
   last_obs_seconds         = -1,
   num_output_state_members = 20,
   num_output_obs_members   = 20,
   output_interval          = 1,
   num_groups               = 1,
   input_qc_threshold       =  4.0,
   outlier_threshold        = -1.0,
   output_forward_op_errors = .false.,
   output_timestamps        = .false.,
   output_inflation         = .true.,

   inf_flavor               = 0,                       0,
   inf_start_from_restart   = .false.,                 .false.,
   inf_output_restart       = .false.,                 .false.,
   inf_deterministic        = .true.,                  .true.,
   inf_in_file_name         = 'not_initialized',       'not_initialized',
   inf_out_file_name        = 'not_initialized',       'not_initialized',
   inf_diag_file_name       = 'not_initialized',       'not_initialized',
   inf_initial              = 1.0,                     1.0,
   inf_sd_initial           = 0.0,                     0.0,
   inf_lower_bound          = 1.0,                     1.0,
   inf_upper_bound          = 1000000.0,               1000000.0,
   inf_sd_lower_bound       = 0.0,                     0.0
/

&smoother_nml
   num_lags              = 0,
   start_from_restart    = .false.,
   output_restart        = .false.,
   restart_in_file_name  = 'smoother_ics',
   restart_out_file_name = 'smoother_restart'  /

&ensemble_manager_nml
   single_restart_file_in  = .true.,
   single_restart_file_out = .true.,
   perturbation_amplitude  = 0.2  /

&assim_tools_nml
   filter_kind                     = 1,
   cutoff                          = 0.2,
   sort_obs_inc                    = .false.,
   spread_restoration              = .false.,
   sampling_error_correction       = .false.,
   adaptive_localization_threshold = -1,
   print_every_nth_obs             = 0  /

&cov_cutoff_nml
   select_localization = 1  /
```

(continues on next page)

(continued from previous page)

```
&reg_factor_nml
   select_regression    = 1,
   input_reg_file       = "time_mean_reg",
   save_reg_diagnostics = .false.,
   reg_diagnostics_file = "reg_diagnostics"  /

&obs_sequence_nml
   write_binary_obs_sequence = .false.  /

&obs_kind_nml
   assimilate_these_obs_types = 'RAW_STATE_VARIABLE'  /

&assim_model_nml
   write_binary_restart_files = .true. /

&model_nml
   sigma  = 10.0,
   r      = 28.0,
   b      = 2.6666666666667,
   deltat = 0.01,
   time_step_days = 0,
   time_step_seconds = 3600  /

&utilities_nml
   TERMLEVEL = 1,
   logfilename = 'dart_log.out'  /
```

Only the non-obvious(?) entries for `filter_nml` will be discussed.

namelist variable	description
`ens_size`	Number of ensemble members. 100 is sufficient for most of the L63 exercises.
`start_from_restart`	when '.false.', `filter` will generate its own ensemble of initial conditions. It is important to note that the filter still makes use of the file named by `restart_in_file_name` (i.e. `perfect_ics`) by randomly perturbing these state variables.
`num_output_state_members`	specifies the number of state vectors contained in the netCDF diagnostic files. May be a value from 0 to `ens_size`.
`num_output_obs_members`	specifies the number of 'observations' (derived from applying the forward operator to the state vector) are contained in the `obs_seq.final` file. May be a value from 0 to `ens_size`
`inf_flavor`	A value of 0 results in no inflation.(spin-up)

The filter is told to generate its own ensemble initial conditions since `start_from_restart` is '.false.'. However, it is important to note that the filter still makes use of `perfect_ics` which is set to be the `restart_in_file_name`. This is the model state generated from the first 24,000 step model integration by `perfect_model_obs`. `Filter` generates its ensemble initial conditions by randomly perturbing the state variables of this state.

`num_output_state_members` are '.true.' so the state vector is output at every time for which there are observations (once a day here). `Posterior_Diag.nc` and `Prior_Diag.nc` then contain values for 20 ensemble members once a day. Once the namelist is set, execute `filter` to integrate the ensemble forward for 24,000 steps with the final ensemble state written to the `filter_restart`. Copy the `perfect_model_obs` restart file `perfect_restart` (the `true state') to `perfect_ics`, and the `filter` restart file `filter_restart` to `filter_ics` so that future assimilation experiments can be initialized from these spun-up states.

```
./filter
cp perfect_restart perfect_ics
cp filter_restart filter_ics
```

The spin-up of the ensemble can be viewed by examining the output in the netCDF files `True_State.nc` generated by `perfect_model_obs` and `Posterior_Diag.nc` and `Prior_Diag.nc` generated by `filter`. To do this, see the detailed discussion of matlab diagnostics in Appendix I.

3. Simulate a particular observing system

Begin by using `create_obs_sequence` to generate an observation set in which each of the 3 state variables of L63 is observed with an observational error variance of 1.0 for each observation. To do this, use the following input sequence (the text including and after # is a comment and does not need to be entered):

4	# upper bound on num of observations in sequence
0	# number of copies of data (0 for just a definition)
0	# number of quality control values per field (0 or greater)
0	# -1 to exit/end observation definitions
-1	# observe state variable 1
0 0	# time – days, seconds
1.0	# observational variance
0	# -1 to exit/end observation definitions
-2	# observe state variable 2
0 0	# time – days, seconds
1.0	# observational variance
0	# -1 to exit/end observation definitions
-3	# observe state variable 3
0 0	# time – days, seconds
1.0	# observational variance
-1	# -1 to exit/end observation definitions
set_def.out	# Output file name

Now, generate an observation sequence definition by running `create_fixed_network_seq` with the following input sequence:

set_def.out	# Input observation set definition file
1	# Regular spaced observation interval in time
1000	# 1000 observation times
0, 43200	# First observation after 12 hours (0 days, 12 * 3600 seconds)
0, 43200	# Observations every 12 hours
obs_seq.in	# Output file for observation sequence definition

4. Generate a particular observing system and true state

An observation sequence file is now generated by running `perfect_model_obs` with the namelist values (unchanged from step 2):

```
&perfect_model_obs_nml
   start_from_restart    = .true.,
   output_restart        = .true.,
   async                 = 0,
   init_time_days        = 0,
   init_time_seconds     = 0,
   first_obs_days        = -1,
   first_obs_seconds     = -1,
   last_obs_days         = -1,
   last_obs_seconds      = -1,
   output_interval       = 1,
   restart_in_file_name  = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   obs_seq_in_file_name  = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   adv_ens_command       = "./advance_ens.csh"   /
```

This integrates the model starting from the state in `perfect_ics` for 1000 12-hour intervals outputting synthetic observations of the three state variables every 12 hours and producing a netCDF diagnostic file, `True_State.nc`.

5. Filtering

Finally, `filter` can be run with its namelist set to:

```
&filter_nml
   async                    = 0,
   adv_ens_command          = "./advance_model.csh",
   ens_size                 = 100,
   start_from_restart       = .true.,
   output_restart           = .true.,
   obs_sequence_in_name     = "obs_seq.out",
   obs_sequence_out_name    = "obs_seq.final",
   restart_in_file_name     = "filter_ics",
   restart_out_file_name    = "filter_restart",
   init_time_days           = 0,
   init_time_seconds        = 0,
   first_obs_days           = -1,
   first_obs_seconds        = -1,
   last_obs_days            = -1,
   last_obs_seconds         = -1,
   num_output_state_members = 20,
   num_output_obs_members   = 20,
   output_interval          = 1,
   num_groups               = 1,
   input_qc_threshold       =  4.0,
   outlier_threshold        = -1.0,
   output_forward_op_errors = .false.,
   output_timestamps        = .false.,
   output_inflation         = .true.,

   inf_flavor               = 0,                       0,
```

(continues on next page)

(continued from previous page)

```
   inf_start_from_restart    = .false.,                .false.,
   inf_output_restart        = .false.,                .false.,
   inf_deterministic         = .true.,                 .true.,
   inf_in_file_name          = 'not_initialized',      'not_initialized',
   inf_out_file_name         = 'not_initialized',      'not_initialized',
   inf_diag_file_name        = 'not_initialized',      'not_initialized',
   inf_initial               = 1.0,                    1.0,
   inf_sd_initial            = 0.0,                    0.0,
   inf_lower_bound           = 1.0,                    1.0,
   inf_upper_bound           = 1000000.0,              1000000.0,
   inf_sd_lower_bound        = 0.0,                    0.0
 /
```

`filter` produces two output diagnostic files, `Prior_Diag.nc` which contains values of the ensemble mean, ensemble spread, and ensemble members for 12- hour lead forecasts before assimilation is applied and `Posterior_Diag.nc` which contains similar data for after the assimilation is applied (sometimes referred to as analysis values).

Now try applying all of the matlab diagnostic functions described in the Matlab® Diagnostics section.

6.14.10 The tutorial

The `DART/tutorial` documents are an excellent way to kick the tires on DART and learn about ensemble data assimilation. If you have gotten this far, you can run anything in the tutorial.

6.14.11 Matlab® diagnostics

The output files are netCDF files and may be examined with many different software packages. We use Matlab®, and provide our diagnostic scripts in the hopes that they are useful.

The diagnostic scripts and underlying functions reside in two places: `DART/diagnostics/matlab` and `DART/matlab`. They are reliant on the public-domain MEXNC/SNCTOOLS netCDF interface from http://mexcdf.sourceforge.net. If you do not have them installed on your system and want to use Matlab to peruse netCDF, you must follow their installation instructions. The 'interested reader' may want to look at the `DART/matlab/startup.m` file I use on my system. If you put it in your `$HOME/matlab` directory it is invoked every time you start up Matlab.

Once you can access the `nc_varget` function from within Matlab you can use our diagnostic scripts. It is necessary to prepend the location of the `DART/matlab` scripts to the `matlabpath`. Keep in mind the location of the netcdf operators on your system WILL be different from ours . . . and that's OK.

```
[models/lorenz_63/work]$ matlab -nodesktop

                                          < M A T L A B >
                              Copyright 1984-2002 The MathWorks, Inc.
                                  Version 6.5.0.180913a Release 13
                                            Jun 18 2002

  Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.

  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.
```

(continues on next page)

(continued from previous page)

```
>> which nc_varget
/contrib/matlab/snctools/4024/nc_varget.m
>>ls *.nc

ans =

Posterior_Diag.nc  Prior_Diag.nc  True_State.nc


>>path('../../../matlab',path)
>>path('../../../diagnostics/matlab',path)
>>which plot_ens_err_spread
../../../matlab/plot_ens_err_spread.m
>>help plot_ens_err_spread

  DART : Plots summary plots of the ensemble error and ensemble spread.
                         Interactively queries for the needed information.
                         Since different models potentially need different
                         pieces of information ... the model types are
                         determined and additional user input may be queried.

  Ultimately, plot_ens_err_spread will be replaced by a GUI.
  All the heavy lifting is done by PlotEnsErrSpread.

  Example 1 (for low-order models)

  truth_file = 'True_State.nc';
  diagn_file = 'Prior_Diag.nc';
  plot_ens_err_spread

>>plot_ens_err_spread
```

And the matlab graphics window will display the spread of the ensemble error for each state variable. The scripts are designed to do the "obvious" thing for the low-order models and will prompt for additional information if needed. The philosophy of these is that anything that starts with a lower-case *plot_some_specific_task* is intended to be user-callable and should handle any of the models. All the other routines in `DART/matlab` are called BY the high-level routines.

Matlab script	description
`plot_bins`	plots ensemble rank histograms
`plot_correl`	Plots space-time series of correlation between a given variable at a given time and other variables at all times in a n ensemble time sequence.
`plot_ens_err_spread`	Plots summary plots of the ensemble error and ensemble spread. Interactively queries for the needed information. Since different models potentially need different pieces of information ... the model types are determined and additional user input may be queried.
`plot_ens_mean_time_series`	Queries for the state variables to plot.
`plot_ens_time_series`	Queries for the state variables to plot.
`plot_phase_space`	Plots a 3D trajectory of (3 state variables of) a single ensemble member. Additional trajectories may be superimposed.
`plot_total_err`	Summary plots of global error and spread.
`plot_var_var_correl`	Plots time series of correlation between a given variable at a given time and another variable at all times in an ensemble time sequence.

6.14.12 Bias, filter divergence and covariance inflation (with the l63 model)

One of the common problems with ensemble filters is filter divergence, which can also be an issue with a variety of other flavors of filters including the classical Kalman filter. In filter divergence, the prior estimate of the model state becomes too confident, either by chance or because of errors in the forecast model, the observational error characteristics, or approximations in the filter itself. If the filter is inappropriately confident that its prior estimate is correct, it will then tend to give less weight to observations than they should be given. The result can be enhanced overconfidence in the model's state estimate. In severe cases, this can spiral out of control and the ensemble can wander entirely away from the truth, confident that it is correct in its estimate. In less severe cases, the ensemble estimates may not diverge entirely from the truth but may still be too confident in their estimate. The result is that the truth ends up being farther away from the filter estimates than the spread of the filter ensemble would estimate. This type of behavior is commonly detected using rank histograms (also known as Talagrand diagrams). You can see the rank histograms for the L63 initial assimilation by using the matlab script `plot_bins`.

A simple, but surprisingly effective way of dealing with filter divergence is known as covariance inflation. In this method, the prior ensemble estimate of the state is expanded around its mean by a constant factor, effectively increasing the prior estimate of uncertainty while leaving the prior mean estimate unchanged. The program `filter` has a group of namelist parameters that controls the application of covariance inflation. For a simple set of inflation values, you will set `inf_flavor`, and `inf_initial`. These values come in pairs; the first value controls inflation of the prior ensemble values, while the second controls inflation of the posterior values. Up to this point `inf_flavor` has been set to 0 indicating that the prior ensemble is left unchanged. Setting the first value of `inf_flavor` to 3 enables one variety of inflation. Set `inf_initial` to different values (try 1.05 and 1.10 and other values). In each case, use the diagnostic matlab tools to examine the resulting changes to the error, the ensemble spread (via rank histogram bins, too), etc. What kind of relation between spread and error is seen in this model?

There are many more options for inflation, including spatially and temporarily varying values, with and without damping. See the discussion of all inflation-related namelist items Website or local file.

6.14.13 Synthetic observations

Synthetic observations are generated from a `perfect' model integration, which is often referred to as the `truth' or a `nature run'. A model is integrated forward from some set of initial conditions and observations are generated as $y = H(x) + e$ where H is an operator on the model state vector, x, that gives the expected value of a set of observations, y, and e is a random variable with a distribution describing the error characteristics of the observing instrument(s) being simulated. Using synthetic observations in this way allows students to learn about assimilation algorithms while being isolated from the additional (extreme) complexity associated with model error and unknown observational error characteristics. In other words, for the real-world assimilation problem, the model has (often substantial) differences from what happens in the real system and the observational error distribution may be very complicated and is certainly not well known. Be careful to keep these issues in mind while exploring the capabilities of the ensemble filters with synthetic observations.

6.14.14 Notes for current users

If you have been updating from the development branch of the DART subversion repository you will not notice much difference between that and the Lanai release. If you are still running the Kodiak release there are many new models, new observation types, capabilities in the assimilation tools, new diagnostics, and new utilities. There is a short list of non-backwards compatible changes (see below), and then a long list of new options and functions.

In the near future we will be making substantial changes to the internal structure of DART to accomodate both larger models and machines with thousands of processors. We will continue to maintain the Lanai release with bug fixes, but we will be updating the subversion trunk with new and non-backwards-compatible code. Checking out the Lanai release branch and running 'svn update' from time to time is the recommended way to update your DART tree.

6.14.15 Non-backwards compatible changes

Changes in the Lanai release (13 Dec 2013) which are *not* backwards compatible with the Kodiak release (30 June 2011):

1. The DART system uses a new random number generator based on the Mersenne Twister algorithm from the GNU scientific library. It is believed to have better behavior in general, and in particular when it is frequently reseeded, as may be the case in some perfect_model_obs experiments. The seed in perfect_model_obs is now based on the time-stamp associated with the data, so running single advances as separate invocations of the executable will still result in a good random distribution of the observation errors. The seeds in several other places in the code have been changed so they are more consistent in the face of different numbers of MPI tasks when executing. The random values should reproduce if an identical run is repeated, but there are still a few places in the code where changing the number of MPI tasks results in different seeds being created for the random number generator, and so the non-deterministic values will differ.
2. The WRF model_mod now interpolates in the vertical in log(pressure) space instead of linear pressure space. This is the new default. There is a module global variable that can be set at compile time to restore the previous behavior.
3. The POP model_mod used to interpolate sensible temperature observations using a potential temperature field in the state vector. The code now correctly does the conversion from potential temperature to sensible (in-situ) temperature during the forward operator process.
4. If your `model_mod.f90` provides a customized `get_close_obs()` routine that makes use of the types/kinds arguments for either the base location or the close location list, there is an important change in this release. The fifth argument to the `get_close_obs()` call is now a list of generic kinds corresponding to the location list. The fourth argument to the `get_dist()` routine is now also a generic kind and not a specific type. In previous versions of the system the list of close locations was sometimes a list of specific types and other times a list of generic kinds. The system now always passes generic kinds for the close locations list for consistency. The base location and specific type remains the same as before. If you have a `get_close_obs()` routine in your `model_mod.f90` file and have questions about usage, contact the DART development team.
5. The `obs_common_subset` program namelist has changed. The program compares `obs_seq.final` files that were produced by different runs of filter using the same input obs_sequence file. The old version supported comparing only 2 files at a time; this version supports up to 50. It also enforces the implicit assumption that the incoming obs_seq.final files are identical except for the DART QC and the obs values.
6. The simple_advection model was incorrectly calling the random number generator initialization routines after generating some random numbers. It now correctly initializes the generator before getting any random values.
7. The gts_to_dart converter now creates separate obs types for surface dewpoint vs obs aloft because they have different vertical coordinates. The obs_diag program (and other diagnostic routines) do not cope with the same obs type having different vertical coordinates because it is trying to bin observations in the vertical (it is unable to convert pressure to height after the fact, for example, or bin surface obs with a height with pressure obs).
8. Shell scripts which used to contain MSS (mass store) commands for long-term archiving have been converted to call HSI (HPSS) commands.
9. The ‘wrf_dart_obs_preprocess’ program will now refuse to superob observations which are too close to the poles. If the superob radius includes either pole, the computation of an average obs location becomes more complicated than the existing code is prepared to deal with. (If this case is of interest to you, contact the DART development team. We have ideas on how to implement this.)
10. The default namelist values for the ‘obs_seq_to_netcdf’ program has changed so the default is a single large time bin, which means you don’t have to know the exact time extents when converting an obs_seq.final file into a netCDF file. You can still set specific bins and get multiple netCDF files as output if you prefer.
11. The tutorial files are now directly in the DART/tutorial directory and no longer in separate subdirectories.

12. The default flags in the mkmf_template.XXX files have been updated to be more consistent with current compiler versions.
13. The default work/input.nml namelists for Lorenz 63 and Lorenz 96 have been changed to give good assimilation results by default. Originally these were set to work with a workshop tutorial in which the settings did not work and as part of the tutorial they were changed to good values. Now the workshop versions of the namelists are separate and copied into place by a workshop_setup script.
14. filter now calls the end_model() subroutine in the model_mod for the first time. It should have been called all along, but was not.
15. The 'rat_cri' namelist item has been removed from the &obs_diag namelist.
16. The preprocess program has a new namelist item 'overwrite_output' and it is .true. by default. The program will no longer fail if the target obs_kind_mod.f90 or obs_def_mod.f90 files exist but will silently overwrite them. Set this namelist item to .false. to recover the previous behavior.

6.14.16 New features

- Customizable Outlier-Threshold Handling
  - Filter contains code to compute whether an observation should not be assimilated because the forward operator mean is too different from the observation value. This is done uniformly for all observation values and types. To customize this computation (e.g. to allow all obs of a particular type to be assimilated without having to pass the outlier threshold test), there is a new namelist item `enable_special_outlier_code` in the &filter_nml namelist that enables a call to a subroutine at the end of the filter.f90 source file. That subroutine can be customized by the user to do any computation required. See the filter namelist documentation Website or local file for more details.
- Fill inflation restart files
  - There is a new utility that will write inflation restart files based on values read from the console. This enables multi-step runs to start with the 'read inflation values from a file' option set to .true. for all steps instead of having to change the namelist after the first cycle. See the documentation Website or *PROGRAM fill_inflation_restart* for more details.
- New location module options
  - There are additional options for the model and observation coordinate systems. Note that only a single location option can be chosen and all observations and all model locations must use that coordinate system. New options include:
    - Channel coordinate system
    - [0-1] periodic 3D coordinate system
    - X,Y,Z 3D Cartesian coordinate system
    - 2D annulus coordinate system

    See the documentation Website or *MODULE location_mod* for more details.
- Missing values in state
  - In some models there are values which are not valid in all ensemble members. With this release there is limited support for this in DART. There are still serious questions about what the correct results should be if the ensemble count for some state vector item is smaller than the total ensemble size. Nevertheless, with this release we have implemented support for missing state vector values in the CLM Land model. There is a new namelist item `allow_missing_in_clm` in the &assim_tools_nml namelist. Setting this to .true. will allow DART to avoid updating any state vector items in which one or more of the ensemble members in CLM have a missing value. Inflation will be disabled for any state vector items where one or

  more ensemble members have missing values. All CLM forward operators must test for and be prepared to return with a failed forward operator code if any of the interpolation items it requires are missing. See the documentation Website or local file for more details.

- Different task layout options
  - The ensemble manager has a new option to distribute MPI tasks round robin across the available nodes instead of assigning them sequentially. The first N tasks, where N is the ensemble size, require more memory than other tasks. Distributing them round-robin may allow assigning more tasks per node with a more uniform memory usage. This may result in a small decrease in performance at runtime, but it might allow using fewer nodes for the job and thus reduce the job cost. See the documentation for the `layout` and `tasks_per_node` in the &ensemble_manager_nml namelist Website or local file for more details.
- Different MPI communication options
  - The ensemble manager has 3 new options for the order in which the communication is done when transposing the ensemble of state vectors. There is a new namelist option in the &ensemble_manager_nml called `communication_configuration` which can have the values 1-4. If DART is running slower than expected, try the various options and see which is fastest on your hardware. The fastest value depends on the MPI library implementation, the type and speed of interconnect, the processor speed, and node memory size and so it is almost impossible to recommend a value without doing timing tests on the target system. See the documentation in the &ensemble_manager_nml namelist Website or local file for more details.
- Several more places where large arrays were put on the stack have been removed, decreasing the total amount of stack required by DART.

6.14.17 New models

- CESM framework components
  - DART now supports running CESM components CAM, POP, and CLM under the CESM framework. Setup scripts are provided to configure a single or multiple component assimilation. See:
    - Website or *CESM* for multi-component assimilation,
    - Website
    - Website or *POP* for POP single component assimilation
    - Website or *CLM* for CLM single component assimilation

    Documentation for the model:

    - the user's guide for CESM version 1.1.1: http://www.cesm.ucar.edu/models/cesm1.1/cesm/doc/usersguide/book1.html
    - the page that explains how to download the release code: http://www.cesm.ucar.edu/models/cesm1.1/cesm/doc/usersguide/x388.html
    - the web page that shows the names of the 'compsets' which are the configurations of the various models: http://www.cesm.ucar.edu/models/cesm1.1/cesm/doc/modelnl/compsets.html
    - list of recent CESM versions: http://www2.cesm.ucar.edu/models/current
- MPAS Atmosphere and Ocean Models
  - DART interface documentation for the MPAS Atmosphere component: Website or *MPAS_ATM*.
  - DART interface documentation for the MPAS Ocean component: Website or *MPAS OCN*.
  - Documentation for the model: MPAS.

- NOAH Land Model
  - Dart interface documentation Website or *NOAH, NOAH-MP*.
  - Documentation for the model: The Community NOAH Land Surface Model (LSM).
- NAAPS Aerosol Model
  - Dart interface documentation Website
  - Documentation for the model: NRL/Monterery Aerosol Model.
- GITM Global Ionosphere Thermosphere Model
  - Dart interface documentation Website or *GITM*.
  - Documentation for the model: GITM Web Pages.
- NOGAPS Global Atmosphere Model
  - Dart interface documentation Website
  - Documentation for the model: NOGAPS.
- SQG Surface Quasi-Geostrophic Model
  - Dart interface documentation Website or *SQG*.
  - Documentation for the model: Paper on SQG model.

The `DART/models/template` directory contains sample files for adding a new model. See this section of the DART web pages for more help on adding a new model.

6.14.18 Changed models

- WRF
  - Allow advanced microphysics schemes (needed interpolation for 7 new kinds)
  - Interpolation in the vertical is now done in log(p) instead of linear pressure space. log(p) is the default, but a compile-time variable can restore the linear interpolation.
  - Added support in the namelist to avoid writing updated fields back into the wrf netcdf files. The fields are still updated during the assimilation but the updated data is not written back to the wrfinput file during the dart_to_wrf step.
  - Fixed an obscure bug in the vertical convert routine of the wrf model_mod that would occasionally fail to convert an obs. This would make tiny differences in the output as the number of mpi tasks change. No quantitative differences in the results but they were not bitwise compatible before and they are again now.
- CAM
  - DART/CAM now runs under the CESM framework, so all options available with the framework can be used.
  - Support for the SE core (HOMME) has been developed but is NOT part of this release. Please contact the DART Development Group if you have an interest in this configuration of CAM.
- Simple Advection Model
  - Fixed a bug where the random number generator was being used before being called with an initial seed.

6.14.19 New observation types/forward operators

- Many new observation types related to land and atmospheric chemistry have been added. See the `obs_kind/obs_kind_mod.f90` for a list of the generic kinds now available.
- New forward operator for total precipitable water. It loops over model levels to compute the accumulated value. See Website or *Total Precipitable Water Observations*.
- New forward operator for COSMOS ground moisture observations. See Website or *PROGRAM COSMOS_to_obs*.
- New forward operator for MIDAS total electron count observations. See Website or *PROGRAM MIDAS_to_obs*.
- Added example of how to set additional metadata in an observation to the *obs_def_1d_state_mod.f90* file. Website or *MODULE obs_def_1d_state_mod*.

6.14.20 New observation types/sources

- MADIS Added a converter for wind profiler data to the set of existing MADIS converters. More scripting support in the MADIS obs converters; more error checks added to the rawin converter. Documentation Website or *MADIS Data Ingest System*.
- Ameriflux Added an obs_sequence converter for Ameriflux land observations of latent heat flux, sensible heat flux, net ecosystem production). Documentation Website or *PROGRAM level4_to_obs*.
- MODIS Added an obs_sequence converter for MODIS snow coverage measurements. Documentation Website or *PROGRAM snow_to_obs*.
- COSMOS Added an obs_sequence converter for COSMOS ground moisture observations. Documentation Website or *PROGRAM COSMOS_to_obs*.
- MIDAS Added an obs_sequence converter for MIDAS observations of Total Electron Count. Documentation Website or *PROGRAM MIDAS_to_obs*.
- GPS Updated scripts for the GPS converter; added options to convert data from multiple satellites. Documentation Website or *GPS Observations*.
- wrf_dart_obs_preprocess Added processing for wind profiler observation to the wrf_dart_obs_preprocess program. Documentation Website or *PROGRAM wrf_dart_obs_preprocess*.
- AIRS Fix BUG in AIRS converter - the humidity obs are accumulated across the layers and so the best location for them is the layer midpoint and not on the edges (levels) as the temperature obs are. Also fixed off-by-one error where the converter would make one more obs above the requested top level. Documentation Website or *AIRS Observations*.
- GTS Made gts_to_dart converter create separate obs types for surface dewpoint vs obs aloft because they have different vertical coordinates. Documentation Website or *GTSPP Observations*.
- Tape Archive scripts Converted mss commands to hpss commands for a couple observation converter shell scripts (inc AIRS).
- Evenly distributed obs New matlab code to generate evenly spaced observations on the surface of a sphere (e.g. the globe). Documentation Website or local file.
- observation utilities Added obs_loop.f90 example file in obs_sequence directory; example template for how to construct special purpose obs_sequence tools. Documentation (source) Website or local file.
- PREPBUFR Change the default in the script for the prepbufr converter so it will swap bytes, since all machines except ibms will need this now. Documentation Website or *PROGRAM prepbufr*.

6.14.21 New diagnostics and documentation

Better Web Pages. We've put a lot of effort into expanding our documentation. For example, please check out the Matlab diagnostics section or the pages outlining the observation sequence file contents.
But there's always more to add. **Please let us know where we are lacking.**

Other new stuff:

- Handle empty epochs in the obs_seq_to_netcdf converter.
- Added a matlab utility to show the output of a 'hop' test (running a model for a continuous period vs. stopping and restarting a run).
- Improved the routine that computes axes tick values in plots with multiple values plotted on the same plot.
- The obs_common_subset program can select common observations from up to 4 observation sequence files at a time.
- Add code in obs_seq_verify to ensure that the ensemble members are in the same order in all netcdf files.
- Added support for the unstructured grids of mpas to our matlab diagnostics.
- Fix to writing of ReportTime in obs_seq_coverage.
- Fixed logic in obs_seq_verify when determining the forecast lat.
- Fixed loops inside obs_seq_coverage which were using the wrong limits on the loops. Fixed writing of 'ntimes' in output netcdf variable.
- Rewrote the algorithm in the obs_selection tool so it had better scaling with large numbers of obs.
- Several improvements to the 'obs_diag' program:
  - Added preliminary support for a list of 'trusted obs' in the obs_diag program.
  - Can disable the rank histogram generation with a namelist item.
  - Can define height_edges or heights in the namelist, but not both.
  - The 'rat_cri' namelist item (critical ratio) has been deprecated.
- Extend obs_seq_verify so it can be used for forecasts from a single member. minor changes to obs_selection, obs_seq_coverage and obs_seq_verify to support a single member.
- Added Matlab script to read/print timestamps from binary dart restart/ic files.
- Default for obs_seq_to_netcdf in all the namelists is now 'one big time bin' so you don't have to know the exact timespan of an obs_seq.final file before converting to netCDF.

6.14.22 New utilities

This section describes updates and changes to the tutorial materials, scripting, setup, and build information since the Kodiak release.

- The mkmf-generated Makefiles now take care of calling 'fixsystem' if needed so the mpi utilities code compiles without further user intervention for any compiler.
- Make the default input.nml for the Lorenz 96 and Lorenz 63 model gives good assimilation results. Rename the original input.nml to input.workshop.nml. The workshop_setup script renames it back before doing anything else so this won't break the workshop instructions. Simplify all the workshop_setup.csh scripts to do the minimal work needed by the DART tutorial.

- Updates to the models/template directory with the start of a full 3d geophysical model template. Still under construction.
- Move the pdf files in the tutorial directory up a level. Removed framemaker source files because we no longer have access to a working version of the Framemaker software. Moved routines that generate figures and diagrams to a non-distributed directory of the subversion repository.
- Enable netCDF large file support in the work/input.nml for models which are likely to have large state vectors.
- Minor updates to the doc.css file, make pages look identical in the safari and firefox browsers.
- Added a utility that sorts and reformats namelists, culls all comments to the bottom of the file. Useful for doing diffs and finding duplicated namelists in a file.
- Cleaned up mkmf files - removed files for obsolete platforms and compilers, updated suggested default flags for intel.
- Update the mkmf template for gfortran to allow fortran source lines longer than 132 characters.

6.15 Kodiak

6.15.1 DART Kodiak release documentation

Attention: Kodiak is a prior release of DART. Its source code is available via the DART repository on Github. This documentation is preserved merely for reference. See the DART homepage to learn about the latest release.

6.15.2 DART overview

The Data Assimilation Research Testbed (DART) is designed to facilitate the combination of assimilation algorithms, models, and real (or synthetic) observations to allow increased understanding of all three. The DART programs are highly portable, having been compiled with many Fortran 90 compilers and run on linux compute-servers, linux clusters, OSX laptops/desktops, SGI Altix clusters, supercomputers running AIX, and more. Read the Customizations section for help in building on new platforms.

DART employs a modular programming approach to apply an Ensemble Kalman Filter which nudges models toward a state that is more consistent with information from a set of observations. Models may be swapped in and out, as can different algorithms in the Ensemble Kalman Filter. The method requires running multiple instances of a model to generate an ensemble of states. A forward operator appropriate for the type of observation being assimilated is applied to each of the states to generate the model's estimate of the observation. Comparing these estimates and their uncertainty to the observation and its uncertainty ultimately results in the adjustments to the model states. There's much more to it, described in detail in the tutorial directory of the package.

DART diagnostic output includes two netCDF files containing the model states just before the adjustment (`Prior_Diag.nc`) and just after the adjustment (`Posterior_Diag.nc`) as well as a file `obs_seq.final` with the model estimates of the observations. There is a suite of Matlab® functions that facilitate exploration of the results, but the netCDF files are inherently portable and contain all the necessary metadata to interpret the contents.

In this document links are available which point to Web-based documentation files and also to the same information in html files distributed with DART. If you have used subversion to check out a local copy of the DART files you can open this file in a browser by loading `DART/doc/html/Kodiak_release.html` and then use the `local file` links to see other documentation pages without requiring a connection to the internet. If you are looking at this documentation from the `www.image.ucar.edu` web server or you are connected to the internet you can use the `Website` links to view other documentation pages.

6.15.3 Getting started

What's required

1. a Fortran 90 compiler
2. a netCDF library including the F90 interfaces
3. the C shell
4. (optional, to run in parallel) an MPI library

DART has been tested on many Fortran compilers and platforms. We don't have any platform-dependent code sections and we use only the parts of the language that are portable across all the compilers we have access to. We explicitly set the Fortran 'kind' for all real values and do not rely on autopromotion or other compile-time flags to set the default byte size for numbers. It is possible that some model-specific interface code from outside sources may have specific compiler flag requirements; see the documentation for each model. The low-order models and all common portions of the DART code compile cleanly.

DART uses the netCDF self-describing data format with a particular metadata convention to describe output that is used to analyze the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools.

Since most of the models being used with DART are written in Fortran and run on various UNIX or *nix platforms, the development environment for DART is highly skewed to these machines. We do most of our development on a small linux workstation and a mac laptop running OSX 10.x, and we have an extensive test network. (I've never built nor run DART on a Windows machine - so I don't even know if it's possible. If you have run it (under Cygwin?) please let me know how it went – I'm curious. Tim - thoar 'at' ucar 'dot ' edu)

What's nice to have

- **ncview**: DART users have used ncview to create graphical displays of output data fields. The 2D rendering is good for 'quick-look' type uses, but I wouldn't want to publish with it.
- **NCO**: The NCO tools are able to perform operations on netCDF files like concatenating, slicing, and dicing.
- **Matlab®**: A set of Matlab® scripts designed to produce graphical diagnostics from DART netCDF output files are also part of the DART project.
- **MPI**: The DART system includes an MPI option. MPI stands for 'Message Passing Interface', and is both a library and run-time system that enables multiple copies of a single program to run in parallel, exchange data, and combine to solve a problem more quickly. DART does **NOT** require MPI to run; the default build scripts do not need nor use MPI in any way. However, for larger models with large state vectors and large numbers of observations, the data assimilation step will run much faster in parallel, which requires MPI to be installed and used. However, if multiple ensembles of your model fit comfortably (in time and memory space) on a single processor, you need read no further about MPI.

Types of input

DART programs can require three different types of input. First, some of the DART programs, like those for creating synthetic observational datasets, require interactive input from the keyboard. For simple cases this interactive input can be made directly from the keyboard. In more complicated cases a file containing the appropriate keyboard input can be created and this file can be directed to the standard input of the DART program. Second, many DART programs expect one or more input files in DART specific formats to be available. For instance, `perfect_model_obs`, which creates a synthetic observation set given a particular model and a description of a sequence of observations, requires an input file that describes this observation sequence. At present, the observation files for DART are in a custom format in either human-readable ascii or more compact machine-specific binary. Third, many DART modules (including main programs) make use of the Fortran90 namelist facility to obtain values of certain parameters at run-time. All programs look for a namelist input file called `input.nml` in the directory in which the program is executed. The `input.nml` file can contain a sequence of individual Fortran90 namelists which specify values of particular parameters for modules that compose the executable program.

6.15.4 Installation

This document outlines the installation of the DART software and the system requirements. The entire installation process is summarized in the following steps:

1. Determine which F90 compiler is available.
2. Determine the location of the `netCDF` library.
3. Download the DART software into the expected source tree.
4. Modify certain DART files to reflect the available F90 compiler and location of the appropriate libraries.
5. Build the executables.

We have tried to make the code as portable as possible, but we do not have access to all compilers on all platforms, so there are no guarantees. We are interested in your experience building the system, so please email me (Tim Hoar) thoar ‘at’ ucar ‘dot’ edu (trying to cut down on the spam).

After the installation, you might want to peruse the following.

- Running the Lorenz_63 Model.
- Using the Matlab® diagnostic scripts.
- A short discussion on bias, filter divergence and covariance inflation.
- And another one on synthetic observations.

You should *absolutely* run the DART_LAB interactive tutorial (if you have Matlab available) and look at the DART_LAB presentation slides Website or *DART_LAB Tutorial* in the `DART_LAB` directory, and then take the tutorial in the `DART/tutorial` directory.

Requirements: an F90 compiler

The DART software has been successfully built on several Linux/x86 platforms with several versions of the Intel Fortran Compiler for Linux, which (at one point) is/was free for individual scientific use. Also Intel Fortran for Mac OSX. It has also been built and successfully run with several versions of each of the following: Portland Group Fortran Compiler, Lahey Fortran Compiler, Pathscale Fortran Compiler, GNU Fortran 95 Compiler (“gfortran”), Absoft Fortran 90/95 Compiler (Mac OSX). Since recompiling the code is a necessity to experiment with different models, there are no binaries to distribute.

DART uses the netCDF self-describing data format for the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. In particular, DART also makes use of the F90 interface to the library which is available through the netcdf.mod and typesizes.mod modules. *IMPORTANT*: different compilers create these modules with different "case" filenames, and sometimes they are not **both** installed into the expected directory. It is required that both modules be present. The normal place would be in the netcdf/include directory, as opposed to the netcdf/lib directory.

If the netCDF library does not exist on your system, you must build it (as well as the F90 interface modules). The library and instructions for building the library or installing from an RPM may be found at the netCDF home page: http://www.unidata.ucar.edu/packages/netcdf/ Pay particular attention to the compiler-specific patches that must be applied for the Intel Fortran Compiler. (Or the PG compiler, for that matter.)

The location of the netCDF library, libnetcdf.a, and the locations of both netcdf.mod and typesizes.mod will be needed by the makefile template, as described in the compiling section. Depending on the netCDF build options, the Fortran 90 interfaces may be built in a separate library named netcdff.a and you may need to add -lnetcdff to the library flags.

6.15.5 Unpacking the distribution

This release of the DART source code can be downloaded as a compressed zip or tar.gz file. When extracted, the source tree will begin with a directory named DART and will be approximately 206.5 Mb. Compiling the code in this tree (as is usually the case) will necessitate much more space.

```
$ gunzip DART-7.0.0.tar.gz
$ tar -xvf DART-7.0.0.tar
```

You should wind up with a directory named DART.

The code tree is very "bushy"; there are many directories of support routines, etc. but only a few directories involved with the customization and installation of the DART software. If you can compile and run ONE of the low-order models, you should be able to compile and run ANY of the low-order models. For this reason, we can focus on the Lorenz `63 model. Subsequently, the only directories with files to be modified to check the installation are: DART/mkmf, DART/models/lorenz_63/work, and DART/matlab (but only for analysis).

6.15.6 Customizing the build scripts – overview

DART executable programs are constructed using two tools: make and mkmf. The make utility is a very common piece of software that requires a user-defined input file that records dependencies between different source files. make then performs a hierarchy of actions when one or more of the source files is modified. The mkmf utility is a custom preprocessor that generates a make input file (named Makefile) and an example namelist *input.nml.program_default* with the default values. The Makefile is designed specifically to work with object-oriented Fortran90 (and other languages) for systems like DART.

mkmf requires two separate input files. The first is a `template' file which specifies details of the commands required for a specific Fortran90 compiler and may also contain pointers to directories containing pre-compiled utilities required by the DART system. **This template file will need to be modified to reflect your system**. The second input file is a `path_names' file which includes a complete list of the locations (either relative or absolute) of all Fortran90 source files that are required to produce a particular DART program. Each 'path_names' file must contain a path for exactly one Fortran90 file containing a main program, but may contain any number of additional paths pointing to files containing Fortran90 modules. An mkmf command is executed which uses the 'path_names' file and the mkmf template file to produce a Makefile which is subsequently used by the standard make utility.

Shell scripts that execute the mkmf command for all standard DART executables are provided as part of the standard DART software. For more information on mkmf see the FMS mkmf description.

One of the benefits of using mkmf is that it also creates an example namelist file for each program. The example namelist is called *input.nml.program_default*, so as not to clash with any exising input.nml that may exist in that directory.

Building and customizing the 'mkmf.template' file

A series of templates for different compilers/architectures exists in the `DART/mkmf/` directory and have names with extensions that identify the compiler, the architecture, or both. This is how you inform the build process of the specifics of your system. Our intent is that you copy one that is similar to your system into `mkmf.template` and customize it. For the discussion that follows, knowledge of the contents of one of these templates (i.e. `mkmf.template.gfortran`) is needed. Note that only the LAST lines are shown here, the head of the file is just a big comment (worth reading, btw).

```
...
MPIFC = mpif90
MPILD = mpif90
FC = gfortran
LD = gfortran
NETCDF = /usr/local
INCS = ${NETCDF}/include
FFLAGS = -O2 -I$(INCS)
LIBS = -L${NETCDF}/lib -lnetcdf
LDFLAGS = -I$(INCS) $(LIBS)
```

Essentially, each of the lines defines some part of the resulting `Makefile`. Since `make` is particularly good at sorting out dependencies, the order of these lines really doesn't make any difference. The `FC = gfortran` line ultimately defines the Fortran90 compiler to use, etc. The lines which are most likely to need site-specific changes start with `FFLAGS` and `NETCDF`, which indicate where to look for the netCDF F90 modules and the location of the netCDF library and modules.

If you have MPI installed on your system `MPIFC, MPILD` dictate which compiler will be used in that instance. If you do not have MPI, these variables are of no consequence.

Netcdf

Modifying the `NETCDF` value should be relatively straightforward.
Change the string to reflect the location of your netCDF installation containing `netcdf.mod` and `typesizes.mod`. The value of the `NETCDF` variable will be used by the `FFLAGS, LIBS,` and `LDFLAGS` variables.

FFLAGS

Each compiler has different compile flags, so there is really no way to exhaustively cover this other than to say the templates as we supply them should work – depending on the location of your netCDF. The low-order models can be compiled without a `-r8` switch, but the `bgrid_solo` model cannot.

Libs

The Fortran 90 interfaces may be part of the default `netcdf.a` library and `-lnetcdf` is all you need. However it is also common for the Fortran 90 interfaces to be built in a separate library named `netcdff.a`. In that case you will need `-lnetcdf` and also `-lnetcdff` on the **LIBS** line. This is a build-time option when the netCDF libraries are compiled so it varies from site to site.

Customizing the 'path_names_*' file

Several path_names_* files are provided in the work directory for each specific model, in this case: DART/models/lorenz_63/work. Since each model comes with its own set of files, the path_names_* files need no customization.

6.15.7 Building the Lorenz_63 DART project

DART executables are constructed in a work subdirectory under the directory containing code for the given model. From the top-level DART directory change to the L63 work directory and list the contents:

```
$ cd DART/models/lorenz_63/work
$ ls -1
```

With the result:

```
Posterior_Diag.nc
Prior_Diag.nc
True_State.nc
filter_ics
filter_restart
input.nml
mkmf_create_fixed_network_seq
mkmf_create_obs_sequence
mkmf_filter
mkmf_obs_diag
mkmf_obs_sequence_tool
mkmf_perfect_model_obs
mkmf_preprocess
mkmf_restart_file_tool
mkmf_wakeup_filter
obs_seq.final
obs_seq.in
obs_seq.out
obs_seq.out.average
obs_seq.out.x
obs_seq.out.xy
obs_seq.out.xyz
obs_seq.out.z
path_names_create_fixed_network_seq
path_names_create_obs_sequence
path_names_filter
path_names_obs_diag
path_names_obs_sequence_tool
path_names_perfect_model_obs
path_names_preprocess
path_names_restart_file_tool
path_names_wakeup_filter
perfect_ics
perfect_restart
quickbuild.csh
set_def.out
workshop_setup.csh
```

In all the work directories there will be a quickbuild.csh script that builds or rebuilds the executables. The following instructions do this work by hand to introduce you to the individual steps, but in practice running quickbuild will be the normal way to do the compiles.

There are nine mkmf_*xxxxxx* files for the programs

1. preprocess,
2. create_obs_sequence,
3. create_fixed_network_seq,
4. perfect_model_obs,
5. filter,
6. wakeup_filter,
7. obs_sequence_tool, and
8. restart_file_tool, and
9. obs_diag,

along with the corresponding path_names_*xxxxxx* files. There are also files that contain initial conditions, netCDF output, and several observation sequence files, all of which will be discussed later. You can examine the contents of one of the path_names_*xxxxxx* files, for instance path_names_filter, to see a list of the relative paths of all files that contain Fortran90 modules required for the program filter for the L63 model. All of these paths are relative to your DART directory. The first path is the main program (filter.f90) and is followed by all the Fortran90 modules used by this program (after preprocessing).

The mkmf_*xxxxxx* scripts are cryptic but should not need to be modified – as long as you do not restructure the code tree (by moving directories, for example). The function of the mkmf_*xxxxxx* script is to generate a Makefile and an *input.nml.program_default* file. It does not do the compile; make does that:

```
$ csh mkmf_preprocess
$ make
```

The first command generates an appropriate Makefile and the input.nml.preprocess_default file. The second command results in the compilation of a series of Fortran90 modules which ultimately produces an executable file: preprocess. Should you need to make any changes to the DART/mkmf/mkmf.template, you will need to regenerate the Makefile.

The preprocess program actually builds source code to be used by all the remaining modules. It is **imperative** to actually **run** preprocess before building the remaining executables. This is how the same code can assimilate state vector 'observations' for the Lorenz_63 model and real radar reflectivities for WRF without needing to specify a set of radar operators for the Lorenz_63 model!

preprocess reads the &preprocess_nml namelist to determine what observations and operators to incorporate. For this exercise, we will use the values in input.nml. preprocess is designed to abort if the files it is supposed to build already exist. For this reason, it is necessary to remove a couple files (if they exist) before you run the preprocessor. (The quickbuild.csh script will do this for you automatically.)

```
$ \rm -f ../../obs_def/obs_def_mod.f90
$ \rm -f ../../obs_kind/obs_kind_mod.f90
$ ./preprocess
$ ls -l ../../obs_def/obs_def_mod.f90
$ ls -l ../../obs_kind/obs_kind_mod.f90
```

This created ../../obs_def/obs_def_mod.f90 from ../../obs_kind/DEFAULT_obs_kind_mod.F90 and several other modules. ../../obs_kind/obs_kind_mod.f90 was created similarly. Now we can build the rest of the project.

A series of object files for each module compiled will also be left in the work directory, as some of these are undoubtedly needed by the build of the other DART components. You can proceed to create the other programs needed to work with L63 in DART as follows:

```
$ csh mkmf_create_obs_sequence
$ make
$ csh mkmf_create_fixed_network_seq
$ make
$ csh mkmf_perfect_model_obs
$ make
$ csh mkmf_filter
$ make
$ csh mkmf_obs_diag
$ make
```

The result (hopefully) is that six executables now reside in your work directory. The most common problem is that the netCDF libraries and include files (particularly `typesizes.mod`) are not found. Edit the `DART/mkmf/mkmf.template`, recreate the `Makefile`, and try again.

program	purpose
`preprocess`	creates custom source code for just the observation types of interest
`create_obs_sequence`	specify a (set) of observation characteristics taken by a particular (set of) instruments
`create_fixed_network_seq`	repeat a set of observations through time to simulate observing networks where observations are taken in the same location at regular (or irregular) intervals
`perfect_model_obs`	generate "true state" for synthetic observation experiments. Can also be used to 'spin up' a model by running it for a long time.
`filter`	does the assimilation
`obs_diag`	creates observation-space diagnostic files to be explored by the Matlab® scripts.
`obs_sequence_tool`	manipulates observation sequence files. It is not generally needed (particularly for low-order models) but can be used to combine observation sequences or convert from ASCII to binary or vice-versa. We will not cover its use in this document.
`restart_file_tool`	manipulates the initial condition and restart files. We're going to ignore this one here.
`wakeup_filter`	is only needed for MPI applications. We're starting at the beginning here, so we're going to ignore this one, too.

6.15.8 Running Lorenz_63

This initial sequence of exercises includes detailed instructions on how to work with the DART code and allows investigation of the basic features of one of the most famous dynamical systems, the 3-variable Lorenz-63 model. The remarkable complexity of this simple model will also be used as a case study to introduce a number of features of a simple ensemble filter data assimilation system. To perform a synthetic observation assimilation experiment for the L63 model, the following steps must be performed (an overview of the process is given first, followed by detailed procedures for each step):

6.15.9 Experiment overview

1. Integrate the L63 model for a long time starting from arbitrary initial conditions to generate a model state that lies on the attractor. The ergodic nature of the L63 system means a 'lengthy' integration always converges to some point on the computer's finite precision representation of the model's attractor.
2. Generate a set of ensemble initial conditions from which to start an assimilation. Since L63 is ergodic, the ensemble members can be designed to look like random samples from the model's 'climatological distribution'. To generate an ensemble member, very small perturbations can be introduced to the state on the attractor generated by step 1. This perturbed state can then be integrated for a very long time until all memory of its initial condition can be viewed as forgotten. Any number of ensemble initial conditions can be generated by repeating this procedure.

3. Simulate a particular observing system by first creating an 'observation set definition' and then creating an 'observation sequence'. The 'observation set definition' describes the instrumental characteristics of the observations and the 'observation sequence' defines the temporal sequence of the observations.
4. Populate the 'observation sequence' with 'perfect' observations by integrating the model and using the information in the 'observation sequence' file to create simulated observations. This entails operating on the model state at the time of the observation with an appropriate forward operator (a function that operates on the model state vector to produce the expected value of the particular observation) and then adding a random sample from the observation error distribution specified in the observation set definition. At the same time, diagnostic output about the 'true' state trajectory can be created.
5. Assimilate the synthetic observations by running the filter; diagnostic output is generated.

1. Integrate the L63 model for a 'long' time

`perfect_model_obs` integrates the model for all the times specified in the 'observation sequence definition' file. To this end, begin by creating an 'observation sequence definition' file that spans a long time. Creating an 'observation sequence definition' file is a two-step procedure involving `create_obs_sequence` followed by `create_fixed_network_seq`. After they are both run, it is necessary to integrate the model with `perfect_model_obs`.

1.1 Create an observation set definition

`create_obs_sequence` creates an observation set definition, the time-independent part of an observation sequence. An observation set definition file only contains the `location, type,` and `observational error characteristics` (normally just the diagonal observational error variance) for a related set of observations. There are no actual observations, nor are there any times associated with the definition. For spin-up, we are only interested in integrating the L63 model, not in generating any particular synthetic observations. Begin by creating a minimal observation set definition.

In general, for the low-order models, only a single observation set need be defined. Next, the number of individual scalar observations (like a single surface pressure observation) in the set is needed. To spin-up an initial condition for the L63 model, only a single observation is needed. Next, the error variance for this observation must be entered. Since we do not need (nor want) this observation to have any impact on an assimilation (it will only be used for spinning up the model and the ensemble), enter a very large value for the error variance. An observation with a very large error variance has essentially no impact on deterministic filter assimilations like the default variety implemented in DART. Finally, the location and type of the observation need to be defined. For all types of models, the most elementary form of synthetic observations are called 'identity' observations. These observations are generated simply by adding a random sample from a specified observational error distribution directly to the value of one of the state variables. This defines the observation as being an identity observation of the first state variable in the L63 model. The program will respond by terminating after generating a file (generally named `set_def.out`) that defines the single identity observation of the first state variable of the L63 model. The following is a screenshot (much of the verbose logging has been left off for clarity), the user input looks *like this*.

```
[unixprompt]$ ./create_obs_sequence
 Starting program create_obs_sequence
 Initializing the utilities module.
 Trying to log to unit   10
 Trying to open file dart_log.out

 Registering module :
 $url: http://squish/DART/trunk/utilities/utilities_mod.f90 $
 $revision: 2713 $
 $date: 2007-03-25 22:09:04 -0600 (Sun, 25 Mar 2007) $
```

(continues on next page)

(continued from previous page)

```
 Registration complete.

 &UTILITIES_NML
 TERMLEVEL= 2,LOGFILENAME=dart_log.out

 /

 Registering module :
 $url: http://squish/DART/trunk/obs_sequence/create_obs_sequence.f90 $
 $revision: 2713 $
 $date: 2007-03-25 22:09:04 -0600 (Sun, 25 Mar 2007) $
 Registration complete.

 { ... }

 Input upper bound on number of observations in sequence
10

 Input number of copies of data (0 for just a definition)
0

 Input number of quality control values per field (0 or greater)
0

 input a -1 if there are no more obs
0

 Registering module :
 $url: http://squish/DART/trunk/obs_def/DEFAULT_obs_def_mod.F90 $
 $revision: 2820 $
 $date: 2007-04-09 10:37:47 -0600 (Mon, 09 Apr 2007) $
 Registration complete.


 Registering module :
 $url: http://squish/DART/trunk/obs_kind/DEFAULT_obs_kind_mod.F90 $
 $revision: 2822 $
 $date: 2007-04-09 10:39:08 -0600 (Mon, 09 Apr 2007) $
 Registration complete.

 ------------------------------------------------------

 initialize_module obs_kind_nml values are

 -------------- ASSIMILATE_THESE_OBS_TYPES --------------
 RAW_STATE_VARIABLE
 -------------- EVALUATE_THESE_OBS_TYPES --------------
 ------------------------------------------------------

      Input -1 * state variable index for identity observations
      OR input the name of the observation kind from table below:
      OR input the integer index, BUT see documentation...
        1 RAW_STATE_VARIABLE

-1

 input time in days and seconds
```

(continues on next page)

(continued from previous page)

```
1 0

 Input error variance for this observation definition
1000000

 input a -1 if there are no more obs
-1

 Input filename for sequence (  set_def.out   usually works well)
 set_def.out
 write_obs_seq  opening formatted file set_def.out
 write_obs_seq  closed file set_def.out
```

1.2 Create an observation sequence definition

`create_fixed_network_seq` creates an 'observation sequence definition' by extending the 'observation set definition' with the temporal attributes of the observations.

The first input is the name of the file created in the previous step, i.e. the name of the observation set definition that you've just created. It is possible to create sequences in which the observation sets are observed at regular intervals or irregularly in time. Here, all we need is a sequence that takes observations over a long period of time - indicated by entering a 1. Although the L63 system normally is defined as having a non-dimensional time step, the DART system arbitrarily defines the model timestep as being 3600 seconds. If we declare that we have one observation per day for 1000 days, we create an observation sequence definition spanning 24000 'model' timesteps; sufficient to spin-up the model onto the attractor. Finally, enter a name for the 'observation sequence definition' file. Note again: there are no observation values present in this file. Just an observation type, location, time and the error characteristics. We are going to populate the observation sequence with the `perfect_model_obs` program.

```
[unixprompt]$ ./create_fixed_network_seq

 ...

 Registering module :
 $url: http://squish/DART/trunk/obs_sequence/obs_sequence_mod.f90 $
 $revision: 2749 $
 $date: 2007-03-30 15:07:33 -0600 (Fri, 30 Mar 2007) $
 Registration complete.

 static_init_obs_sequence obs_sequence_nml values are
 &OBS_SEQUENCE_NML
 WRITE_BINARY_OBS_SEQUENCE =  F,
 /
 Input filename for network definition sequence (usually  set_def.out  )
set_def.out

 ...

 To input a regularly repeating time sequence enter 1
 To enter an irregular list of times enter 2
1
 Input number of observations in sequence
1000
 Input time of initial ob in sequence in days and seconds
1, 0
```

(continues on next page)

(continued from previous page)

```
 Input period of obs in days and seconds
1, 0
           1
           2
           3
...
         997
         998
         999
        1000
What is output file name for sequence (  obs_seq.in   is recommended )
obs_seq.in
 write_obs_seq  opening formatted file obs_seq.in
 write_obs_seq closed file obs_seq.in
```

1.3 Initialize the model onto the attractor

perfect_model_obs can now advance the arbitrary initial state for 24,000 timesteps to move it onto the attractor.

perfect_model_obs uses the Fortran90 namelist input mechanism instead of (admittedly gory, but temporary) interactive input. All of the DART software expects the namelists to found in a file called input.nml. When you built the executable, an example namelist was created input.nml.perfect_model_obs_default that contains all of the namelist input for the executable. If you followed the example, each namelist was saved to a unique name. We must now rename and edit the namelist file for perfect_model_obs. Copy input.nml.perfect_model_obs_default to input.nml and edit it to look like the following: (just worry about the highlighted stuff - and whitespace doesn't matter)

```
$ cp input.nml.perfect_model_obs_default
$ input.nml
```

```
&perfect_model_obs_nml
   start_from_restart    = .false.,
   output_restart        = .true.,
   async                 = 0,
   init_time_days        = 0,
   init_time_seconds     = 0,
   first_obs_days        = -1,
   first_obs_seconds     = -1,
   last_obs_days         = -1,
   last_obs_seconds      = -1,
   output_interval       = 1,
   restart_in_file_name  = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   obs_seq_in_file_name  = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   adv_ens_command       = "./advance_ens.csh"   /

&ensemble_manager_nml
   single_restart_file_in  = .true.,
   single_restart_file_out = .true.,
   perturbation_amplitude  = 0.2   /

&assim_tools_nml
   filter_kind                     = 1,
```

(continues on next page)

(continued from previous page)

```
   cutoff                          = 0.2,
   sort_obs_inc                    = .false.,
   spread_restoration              = .false.,
   sampling_error_correction       = .false.,
   adaptive_localization_threshold = -1,
   print_every_nth_obs             = 0   /

&cov_cutoff_nml
   select_localization = 1   /

&reg_factor_nml
   select_regression    = 1,
   input_reg_file       = "time_mean_reg",
   save_reg_diagnostics = .false.,
   reg_diagnostics_file = "reg_diagnostics"   /

&obs_sequence_nml
   write_binary_obs_sequence = .false.   /

&obs_kind_nml
   assimilate_these_obs_types = 'RAW_STATE_VARIABLE'   /

&assim_model_nml
   write_binary_restart_files = .true. /

&model_nml
   sigma  = 10.0,
   r      = 28.0,
   b      = 2.6666666666667,
   deltat = 0.01,
   time_step_days = 0,
   time_step_seconds = 3600   /

&utilities_nml
   TERMLEVEL = 1,
   logfilename = 'dart_log.out'   /
```

For the moment, only two namelists warrant explanation. Each namelists is covered in detail in the html files accompanying the source code for the module.

perfect_model_obs_nml

namelist variable	description
start_from_restart	When set to ‘false’, perfect_model_obs generates an arbitrary initial condition (which cannot be guaranteed to be on the L63 attractor). When set to ‘true’, a restart file (specified by restart_in_file_name) is read.
output_restart	When set to ‘true’, perfect_model_obs will record the model state at the end of this integration in the file named by restart_out_file_name.
async	The lorenz_63 model is advanced through a subroutine call - indicated by async = 0. There is no other valid value for this model.
init_time_*xxx*	the start time of the integration.
first_obs_*xxx*	the time of the first observation of interest. While not needed in this example, you can skip observations if you want to. A value of -1 indicates to start at the beginning.
last_obs_*xxx*	the time of the last observation of interest. While not needed in this example, you do not have to assimilate all the way to the end of the observation sequence file. A value of -1 indicates to use all the observations.
output_interval	interval at which to save the model state (in True_State.nc).
restart_in_file_name	is ignored when ‘start_from_restart’ is ‘false’.
restart_out_file_name	if output_restart is ‘true’, this specifies the name of the file containing the model state at the end of the integration.
obs_seq_in_file_name	specifies the file name that results from running create_fixed_network_seq, i.e. the ‘observation sequence definition’ file.
obs_seq_out_file_name	specifies the output file name containing the ‘observation sequence’, finally populated with (perfect?) ‘observations’.
advance_ens_command	specifies the shell commands or script to execute when async /= 0.

utilities_nml

namelist variable	description
TERMLEVEL	When set to ‘1’ the programs terminate when a ‘warning’ is generated. When set to ‘2’ the programs terminate only with ‘fatal’ errors.
logfilename	Run-time diagnostics are saved to this file. This namelist is used by all programs, so the file is opened in APPEND mode. Subsequent executions cause this file to grow.

Executing perfect_model_obs will integrate the model 24,000 steps and output the resulting state in the file perfect_restart. Interested parties can check the spinup in the True_State.nc file.

```
$ ./perfect_model_obs
```

2. Generate a set of ensemble initial conditions

The set of initial conditions for a ‘perfect model’ experiment is created in several steps. 1) Starting from the spun-up state of the model (available in `perfect_restart`), run `perfect_model_obs` to generate the ‘true state’ of the experiment and a corresponding set of observations. 2) Feed the same initial spun-up state and resulting observations into `filter`.

The first step is achieved by changing a perfect_model_obs namelist parameter, copying `perfect_restart` to `perfect_ics`, and rerunning `perfect_model_obs`. This execution of `perfect_model_obs` will advance the model state from the end of the first 24,000 steps to the end of an additional 24,000 steps and place the final state in `perfect_restart`. The rest of the namelists in `input.nml` should remain unchanged.

```
&perfect_model_obs_nml
   start_from_restart    = .true.,
   output_restart        = .true.,
   async                 = 0,
   init_time_days        = 0,
   init_time_seconds     = 0,
   first_obs_days        = -1,
   first_obs_seconds     = -1,
   last_obs_days         = -1,
   last_obs_seconds      = -1,
   output_interval       = 1,
   restart_in_file_name  = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   obs_seq_in_file_name  = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   adv_ens_command       = "./advance_ens.csh"   /
```

```
$ cp perfect_restart perfect_ics
$ ./perfect_model_obs
```

A `True_State.nc` file is also created. It contains the ‘true’ state of the integration.

Generating the ensemble

This step (#2 from above) is done with the program `filter`, which also uses the Fortran90 namelist mechanism for input. It is now necessary to copy the `input.nml.filter_default` namelist to `input.nml`.

```
$ cp input.nml.filter_default
$ input.nml
```

You may also build one master namelist containting all the required namelists. Having unused namelists in the `input.nml` does not hurt anything, and it has been so useful to be reminded of what is possible that we made it an error to NOT have a required namelist. Take a peek at any of the other models for examples of a “fully qualified” `input.nml`.

HINT: if you used `svn` to get the project, try ‘svn revert input.nml’ to restore the namelist that was distributed with the project - which DOES have all the namelist blocks. Just be sure the values match the examples here.

```
&filter_nml
   async                    = 0,
   adv_ens_command          = "./advance_model.csh",
   ens_size                 = 100,
   start_from_restart       = .false.,
   output_restart           = .true.,
```

(continues on next page)

(continued from previous page)

```
   obs_sequence_in_name     = "obs_seq.out",
   obs_sequence_out_name    = "obs_seq.final",
   restart_in_file_name     = "perfect_ics",
   restart_out_file_name    = "filter_restart",
   init_time_days           = 0,
   init_time_seconds        = 0,
   first_obs_days           = -1,
   first_obs_seconds        = -1,
   last_obs_days            = -1,
   last_obs_seconds         = -1,
   num_output_state_members = 20,
   num_output_obs_members   = 20,
   output_interval          = 1,
   num_groups               = 1,
   input_qc_threshold       =  4.0,
   outlier_threshold        = -1.0,
   output_forward_op_errors = .false.,
   output_timestamps        = .false.,
   output_inflation         = .true.,

   inf_flavor               = 0,                       0,
   inf_start_from_restart   = .false.,                 .false.,
   inf_output_restart       = .false.,                 .false.,
   inf_deterministic        = .true.,                  .true.,
   inf_in_file_name         = 'not_initialized',       'not_initialized',
   inf_out_file_name        = 'not_initialized',       'not_initialized',
   inf_diag_file_name       = 'not_initialized',       'not_initialized',
   inf_initial              = 1.0,                     1.0,
   inf_sd_initial           = 0.0,                     0.0,
   inf_lower_bound          = 1.0,                     1.0,
   inf_upper_bound          = 1000000.0,               1000000.0,
   inf_sd_lower_bound       = 0.0,                     0.0
/

&smoother_nml
   num_lags              = 0,
   start_from_restart    = .false.,
   output_restart        = .false.,
   restart_in_file_name  = 'smoother_ics',
   restart_out_file_name = 'smoother_restart'   /

&ensemble_manager_nml
   single_restart_file_in  = .true.,
   single_restart_file_out = .true.,
   perturbation_amplitude  = 0.2   /

&assim_tools_nml
   filter_kind                     = 1,
   cutoff                          = 0.2,
   sort_obs_inc                    = .false.,
   spread_restoration              = .false.,
   sampling_error_correction       = .false.,
   adaptive_localization_threshold = -1,
   print_every_nth_obs             = 0   /

&cov_cutoff_nml
   select_localization = 1   /
```

(continues on next page)

(continued from previous page)

```
&reg_factor_nml
   select_regression    = 1,
   input_reg_file       = "time_mean_reg",
   save_reg_diagnostics = .false.,
   reg_diagnostics_file = "reg_diagnostics"  /

&obs_sequence_nml
   write_binary_obs_sequence = .false.  /

&obs_kind_nml
   assimilate_these_obs_types = 'RAW_STATE_VARIABLE'  /

&assim_model_nml
   write_binary_restart_files = .true. /

&model_nml
   sigma  = 10.0,
   r      = 28.0,
   b      = 2.6666666666667,
   deltat = 0.01,
   time_step_days = 0,
   time_step_seconds = 3600  /

&utilities_nml
   TERMLEVEL = 1,
   logfilename = 'dart_log.out'  /
```

Only the non-obvious(?) entries for `filter_nml` will be discussed.

namelist variable	description
`ens_size`	Number of ensemble members. 100 is sufficient for most of the L63 exercises.
`start_from_restart`	when '.false.', `filter` will generate its own ensemble of initial conditions. It is important to note that the filter still makes use of the file named by `restart_in_file_name` (i.e. `perfect_ics`) by randomly perturbing these state variables.
`num_output_state_members`	specifies the number of state vectors contained in the netCDF diagnostic files. May be a value from 0 to `ens_size`.
`num_output_obs_members`	specifies the number of 'observations' (derived from applying the forward operator to the state vector) are contained in the `obs_seq.final` file. May be a value from 0 to `ens_size`
`inf_flavor`	A value of 0 results in no inflation.(spin-up)

The filter is told to generate its own ensemble initial conditions since `start_from_restart` is '.false.'. However, it is important to note that the filter still makes use of `perfect_ics` which is set to be the `restart_in_file_name`. This is the model state generated from the first 24,000 step model integration by `perfect_model_obs`. `Filter` generates its ensemble initial conditions by randomly perturbing the state variables of this state.

`num_output_state_members` are '.true.' so the state vector is output at every time for which there are observations (once a day here). `Posterior_Diag.nc` and `Prior_Diag.nc` then contain values for 20 ensemble members once a day. Once the namelist is set, execute `filter` to integrate the ensemble forward for 24,000 steps with the final ensemble state written to the `filter_restart`. Copy the `perfect_model_obs` restart file `perfect_restart` (the `true state') to `perfect_ics`, and the `filter` restart file `filter_restart` to `filter_ics` so that future assimilation experiments can be initialized from these spun-up states.

```
./filter
cp perfect_restart perfect_ics
cp filter_restart filter_ics
```

The spin-up of the ensemble can be viewed by examining the output in the netCDF files `True_State.nc` generated by `perfect_model_obs` and `Posterior_Diag.nc` and `Prior_Diag.nc` generated by `filter`. To do this, see the detailed discussion of matlab diagnostics in Appendix I.

3. Simulate a particular observing system

Begin by using `create_obs_sequence` to generate an observation set in which each of the 3 state variables of L63 is observed with an observational error variance of 1.0 for each observation. To do this, use the following input sequence (the text including and after # is a comment and does not need to be entered):

4	# upper bound on num of observations in sequence
0	# number of copies of data (0 for just a definition)
0	# number of quality control values per field (0 or greater)
0	# -1 to exit/end observation definitions
-1	# observe state variable 1
0 0	# time – days, seconds
1.0	# observational variance
0	# -1 to exit/end observation definitions
-2	# observe state variable 2
0 0	# time – days, seconds
1.0	# observational variance
0	# -1 to exit/end observation definitions
-3	# observe state variable 3
0 0	# time – days, seconds
1.0	# observational variance
-1	# -1 to exit/end observation definitions
set_def.out	# Output file name

Now, generate an observation sequence definition by running `create_fixed_network_seq` with the following input sequence:

set_def.out	# Input observation set definition file
1	# Regular spaced observation interval in time
1000	# 1000 observation times
0, 43200	# First observation after 12 hours (0 days, 12 * 3600 seconds)
0, 43200	# Observations every 12 hours
obs_seq.in	# Output file for observation sequence definition

4. Generate a particular observing system and true state

An observation sequence file is now generated by running `perfect_model_obs` with the namelist values (unchanged from step 2):

```
&perfect_model_obs_nml
   start_from_restart    = .true.,
   output_restart        = .true.,
   async                 = 0,
   init_time_days        = 0,
   init_time_seconds     = 0,
   first_obs_days        = -1,
   first_obs_seconds     = -1,
   last_obs_days         = -1,
   last_obs_seconds      = -1,
   output_interval       = 1,
   restart_in_file_name  = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   obs_seq_in_file_name  = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   adv_ens_command       = "./advance_ens.csh"   /
```

This integrates the model starting from the state in `perfect_ics` for 1000 12-hour intervals outputting synthetic observations of the three state variables every 12 hours and producing a netCDF diagnostic file, `True_State.nc`.

5. Filtering

Finally, `filter` can be run with its namelist set to:

```
&filter_nml
   async                    = 0,
   adv_ens_command          = "./advance_model.csh",
   ens_size                 = 100,
   start_from_restart       = .true.,
   output_restart           = .true.,
   obs_sequence_in_name     = "obs_seq.out",
   obs_sequence_out_name    = "obs_seq.final",
   restart_in_file_name     = "filter_ics",
   restart_out_file_name    = "filter_restart",
   init_time_days           = 0,
   init_time_seconds        = 0,
   first_obs_days           = -1,
   first_obs_seconds        = -1,
   last_obs_days            = -1,
   last_obs_seconds         = -1,
   num_output_state_members = 20,
   num_output_obs_members   = 20,
   output_interval          = 1,
   num_groups               = 1,
   input_qc_threshold       =  4.0,
   outlier_threshold        = -1.0,
   output_forward_op_errors = .false.,
   output_timestamps        = .false.,
   output_inflation         = .true.,

   inf_flavor               = 0,                       0,
```

(continues on next page)

(continued from previous page)

```
   inf_start_from_restart    = .false.,                .false.,
   inf_output_restart        = .false.,                .false.,
   inf_deterministic         = .true.,                 .true.,
   inf_in_file_name          = 'not_initialized',      'not_initialized',
   inf_out_file_name         = 'not_initialized',      'not_initialized',
   inf_diag_file_name        = 'not_initialized',      'not_initialized',
   inf_initial               = 1.0,                    1.0,
   inf_sd_initial            = 0.0,                    0.0,
   inf_lower_bound           = 1.0,                    1.0,
   inf_upper_bound           = 1000000.0,              1000000.0,
   inf_sd_lower_bound        = 0.0,                    0.0
 /
```

`filter` produces two output diagnostic files, `Prior_Diag.nc` which contains values of the ensemble mean, ensemble spread, and ensemble members for 12- hour lead forecasts before assimilation is applied and `Posterior_Diag.nc` which contains similar data for after the assimilation is applied (sometimes referred to as analysis values).

Now try applying all of the matlab diagnostic functions described in the Matlab® Diagnostics section.

6.15.10 The tutorial

The `DART/tutorial` documents are an excellent way to kick the tires on DART and learn about ensemble data assimilation. If you have gotten this far, you can run anything in the tutorial.

6.15.11 Matlab® diagnostics

The output files are netCDF files, and may be examined with many different software packages. We happen to use Matlab®, and provide our diagnostic scripts in the hopes that they are useful.

The diagnostic scripts and underlying functions reside in two places: `DART/diagnostics/matlab` and `DART/matlab`. They are reliant on the public-domain netcdf toolbox from `http://woodshole.er.usgs.gov/staffpages/cdenham/public_html/MexCDF/nc4ml5.html` as well as the public-domain CSIRO matlab/netCDF interface from `http://www.marine.csiro.au/sw/matlab-netcdf.html`. If you do not have them installed on your system and want to use Matlab to peruse netCDF, you must follow their installation instructions. The 'interested reader' may want to look at the `DART/matlab/startup.m` file I use on my system. If you put it in your `$HOME/matlab` directory, it is invoked every time you start up Matlab.

Once you can access the `getnc` function from within Matlab, you can use our diagnostic scripts. It is necessary to prepend the location of the `DART/matlab` scripts to the `matlabpath`. Keep in mind the location of the netcdf operators on your system WILL be different from ours … and that's OK.

```
[models/lorenz_63/work]$ matlab -nojvm

                                        < M A T L A B >
                            Copyright 1984-2002 The MathWorks, Inc.
                                Version 6.5.0.180913a Release 13
                                          Jun 18 2002

  Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.
```

(continues on next page)

(continued from previous page)

```
  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

>> which getnc
/contrib/matlab/matlab_netcdf_5_0/getnc.m
>>ls *.nc

ans =

Posterior_Diag.nc  Prior_Diag.nc  True_State.nc


>>path('../../matlab',path)
>>path('../../diagnostics/matlab',path)
>>which plot_ens_err_spread
../../matlab/plot_ens_err_spread.m
>>help plot_ens_err_spread

  DART : Plots summary plots of the ensemble error and ensemble spread.
                         Interactively queries for the needed information.
                         Since different models potentially need different
                         pieces of information ... the model types are
                         determined and additional user input may be queried.

  Ultimately, plot_ens_err_spread will be replaced by a GUI.
  All the heavy lifting is done by PlotEnsErrSpread.

  Example 1 (for low-order models)

  truth_file = 'True_State.nc';
  diagn_file = 'Prior_Diag.nc';
  plot_ens_err_spread

>>plot_ens_err_spread
```

And the matlab graphics window will display the spread of the ensemble error for each state variable. The scripts are designed to do the “obvious” thing for the low-order models and will prompt for additional information if needed. The philosophy of these is that anything that starts with a lower-case *plot_some_specific_task* is intended to be user-callable and should handle any of the models. All the other routines in `DART/matlab` are called BY the high-level routines.

Matlab script	description
`plot_bins`	plots ensemble rank histograms
`plot_correl`	Plots space-time series of correlation between a given variable at a given time and other variables at all times in a n ensemble time sequence.
`plot_ens_err_spread`	Plots summary plots of the ensemble error and ensemble spread. Interactively queries for the needed information. Since different models potentially need different pieces of information … the model types are determined and additional user input may be queried.
`plot_ens_mean_time_series`	Queries for the state variables to plot.
`plot_ens_time_series`	Queries for the state variables to plot.
`plot_phase_space`	Plots a 3D trajectory of (3 state variables of) a single ensemble member. Additional trajectories may be superimposed.
`plot_total_err`	Summary plots of global error and spread.
`plot_var_var_correl`	Plots time series of correlation between a given variable at a given time and another variable at all times in an ensemble time sequence.

6.15.12 Bias, filter divergence and covariance inflation (with the l63 model)

One of the common problems with ensemble filters is filter divergence, which can also be an issue with a variety of other flavors of filters including the classical Kalman filter. In filter divergence, the prior estimate of the model state becomes too confident, either by chance or because of errors in the forecast model, the observational error characteristics, or approximations in the filter itself. If the filter is inappropriately confident that its prior estimate is correct, it will then tend to give less weight to observations than they should be given. The result can be enhanced overconfidence in the model's state estimate. In severe cases, this can spiral out of control and the ensemble can wander entirely away from the truth, confident that it is correct in its estimate. In less severe cases, the ensemble estimates may not diverge entirely from the truth but may still be too confident in their estimate. The result is that the truth ends up being farther away from the filter estimates than the spread of the filter ensemble would estimate. This type of behavior is commonly detected using rank histograms (also known as Talagrand diagrams). You can see the rank histograms for the L63 initial assimilation by using the matlab script `plot_bins`.

A simple, but surprisingly effective way of dealing with filter divergence is known as covariance inflation. In this method, the prior ensemble estimate of the state is expanded around its mean by a constant factor, effectively increasing the prior estimate of uncertainty while leaving the prior mean estimate unchanged. The program `filter` has a group of namelist parameters that controls the application of covariance inflation. For a simple set of inflation values, you will set `inf_flavor`, and `inf_initial`. These values come in pairs; the first value controls inflation of the prior ensemble values, while the second controls inflation of the posterior values. Up to this point `inf_flavor` has been set to 0 indicating that the prior ensemble is left unchanged. Setting the first value of `inf_flavor` to 3 enables one variety of inflation. Set `inf_initial` to different values (try 1.05 and 1.10 and other values). In each case, use the diagnostic matlab tools to examine the resulting changes to the error, the ensemble spread (via rank histogram bins, too), etc. What kind of relation between spread and error is seen in this model?

There are many more options for inflation, including spatially and temporarily varying values, with and without damping. See the discussion of all inflation-related namelist items Website or local file.

6.15.13 Synthetic observations

Synthetic observations are generated from a `perfect' model integration, which is often referred to as the `truth' or a `nature run'. A model is integrated forward from some set of initial conditions and observations are generated as *y = H(x) + e* where *H* is an operator on the model state vector, *x*, that gives the expected value of a set of observations, *y*, and *e* is a random variable with a distribution describing the error characteristics of the observing instrument(s) being simulated. Using synthetic observations in this way allows students to learn about assimilation algorithms while being isolated from the additional (extreme) complexity associated with model error and unknown observational error characteristics. In other words, for the real-world assimilation problem, the model has (often substantial) differences from what happens in the real system and the observational error distribution may be very complicated and is certainly not well known. Be careful to keep these issues in mind while exploring the capabilities of the ensemble filters with synthetic observations.

6.15.14 Notes for current users

If you have been updating from the head of the DART subversion repository (the "trunk") you will not notice much difference between that and the Kodiak release. If you are still running the Jamaica release there are many new models, new observation types, capabilities in the assimilation tools, new diagnostics, and new utilities. There is a very short list of non-backwards compatible changes (see below), and then a long list of new options and functions.

In recent years we have been adding new functionality to the head of the subversion trunk and just testing it and keeping it in working order, maintaining backwards compatibility. We now have many development tasks which will require non-compatible changes in interfaces and behavior. Further DART development will occur on a branch, so checking out either the Kodiak branch or the head of the repository is the recommended way to update your DART tree.

6.15.15 Non-backwards compatible changes

Changes in the Kodiak release which are *not* backwards compatible with the Jamaica release (svn revision number 2857, 12 April 2007):

1. &filter_nml used to have a single entry to control whether to read in both the inflation values and standard deviations from a file or use the settings in the namelist. The old namelist item, `inf_start_from_file`, has been replaced by two items that allow the inflation values and the standard deviation to be read in separately. The new namelist items are `inf_initial_from_file` and `inf_sd_initial_from_file`. See the filter namelist documentation Website or local file for more details.

2. The WRF/DART converter program used to be called `dart_tf_wrf`, had no namelist, and you entered `T` or `F` to indicate which direction you were converting. Now we have `dart_to_wrf` and `wrf_to_dart` (documentation Website) each with a namelist to control various options.

3. The CAM/DART converter programs used to be called `trans_sv_pv` and `trans_pv_sv`, with no namelists, and with several specialized variants (e.g. `trans_pv_sv_time0`). Now we have `cam_to_dart` (documentation Website) and `dart_to_cam` (documentation Website) each with a namelist to control various options.

4. The `obs_def_radar_mod.f90` radar observation module was completely rewritten and the namelist has changed substantially. See the module documentation Website or *MODULE obs_def_radar_mod* for details. For example, the defaults for the old code were:

```
&obs_def_radar_mod_nml
   convert_to_dbz              =  .true. ,
   dbz_threshold               =   0.001 ,
   apply_ref_limit_to_obs      = .false. ,
   reflectivity_limit_obs      =     0.0 ,
   lowest_reflectivity_obs     = -888888.0,
```

(continues on next page)

(continued from previous page)

```
   apply_ref_limit_to_state  = .false. ,
   reflectivity_limit_state  =     0.0 ,
   lowest_reflectivity_state = -888888.0 /
```

and the new ones are:

```
&obs_def_radar_mod_nml
   apply_ref_limit_to_obs     =  .true. ,
   reflectivity_limit_obs     =     0.0 ,
   lowest_reflectivity_obs    =     0.0 ,
   apply_ref_limit_to_fwd_op  =  .true. ,
   reflectivity_limit_fwd_op  =     0.0 ,
   lowest_reflectivity_fwd_op =     0.0 ,
   dielectric_factor          =   0.224 ,
   n0_rain                    =   8.0e6 ,
   n0_graupel                 =   4.0e6 ,
   n0_snow                    =   3.0e6 ,
   rho_rain                   =  1000.0 ,
   rho_graupel                =   400.0 ,
   rho_snow                   =   100.0 ,
   allow_wet_graupel          = .false.,
   microphysics_type          =       3 ,
   allow_dbztowt_conv         = .false. /
```

5. The WRF &model_mod namelist has changed. It now requires a `wrf_state_variables` list to choose which WRF fields are put into the state vector. The order of the field names in the list sets the order of the fields in the state vector. See the WRF model_mod documentation Website or local file for details. Although they haven't been removed from the namelist, the following items have no effect on the code anymore:

   - num_moist_vars
   - surf_obs
   - soil_data
   - h_diab

6. The WRF model_mod now computes geometric heights instead of geopotential heights. It also uses the staggered grids as read in from the `wrfinput_dNN` file(s) instead of interpolating in the non-staggered grid to get individual cell corners.

7. The code in `filter.f90` was corrected to match the documentation for how the namelist item `input_qc_threshold` is handled. (See filter namelist documentation Website or local file.) In the Jamaica release, observations with incoming data QC values greater than or equal to the namelist setting were discarded. Now only incoming data QC values greater than the `input_qc_threshold` are discarded (values equal to the threshold are now kept).

8. The `merge_obs_seq` utility has been replaced by the more comprehensive `obs_sequence_tool` utility. See the documentation Website or *program obs_sequence_tool*.

9. The prepbufr observation converter was located in the `DART/ncep_obs` directory in the last release. It has been moved to be with the other programs that convert various types of observation files into DART format. It is now located in `DART/observations/NCEP`.

10. The sampling error correction generator program in `DART/system_simulation` now has a namelist &full_error_nml. See the documentation Website or *system simulation programs* for more details. Tables for 40 common ensemble sizes are pregenerated in the `DART/system_simulation/final_full_precomputed_tables` directory, and instructions for generating tables for other ensemble sizes are given.

11. Most work directories now have a quickbuild.csh script which recompiles all the executables instead of a workshop_setup.csh script. (Those directories used in the tutorial have both.) To control whether filter is compiled with or without MPI (as a parallel program or not) the quickbuild.csh script takes the optional arguments -mpi or -nompi.
12. The preprocess program was changed so that any obs_def files with module definitions are directly included in the single obs_def_mod.f90 file. This means that as you add and delete obs_def modules from your &preprocess_nml namelist and rerun preprocess you no longer have to add and delete different obs_def modules from your path_names_* files.
13. The utilities module now calls a function in the mpi_utilities code to exit MPI jobs cleanly. This requires that non-mpi programs now include the null_mpi_utilities_mod.f90 file in their path_names_* files.
14. The DART/mpi_utilities directory as distributed now works with all compilers except for gfortran. In DART/mpi_utilities is a ./fixsystem script that when executed will change the source files so they will compile with gfortran. Previous releases compiled with gfortran as distributed but no other compilers.
15. The GPS Radio Occultation observation forward operator code now requires a namelist, &obs_def_gps_nml. See the GPS documentation Website or local file for details on what to add. All input.nml files in the repository have had this added if they have the GPS module in their &preprocess_nml namelist.

6.15.16 New features

- Inflation Damping
  - Handles the case where observation density is irregular in time, e.g. areas which were densely observed at one point are no longer observed. Adaptive inflation values can grow large where the observations are dense, and if that region is no longer observed the inflation is not recomputed. Inflation damping shrinks the inflation values and compensates for this. See the inflation documentation Website or local file for more details and paper references.
- Sampling Error Correction
  - Compensates for the numbers of ensembles being small compared to the number of degrees of freedom in the system. See the last item in this section of the documentation Website or local file for more details.
- Adaptive Localization and Localization Diagnostics
  - See a discussion of localization-related issues Website or local file.
- Scale height vertical localization option in 3d models
  - See a discussion of specifying vertical localization in terms of scale height Website or local file. See the Wikipedia page for a discussion of how scale height is defined. Note that there is no support in the diagnostic Matlab routines for observations using scale height as the vertical coordinate.
- CAM supports FV code, PBS scripting
  - See details on the features of the CAM/DART system Website .
- Boxcar Kernel Filter Option
  - See how to select this filter option in the namelist Website or local file.
- Option for "undefined vertical location" for obs using the 3d sphere locations
  - See how to specify this option when creating observations Website or *MODULE location_mod (threed_sphere)*.
- Schedule module for repeated time intervals
  - See documentation Website or *MODULE schedule_mod*.

- Support for 2 different Mars calendars in time manager
  - Gregorian Mars
  - Solar Mars
- Code corrections to make the smoother run correctly
- Forward operators now have access to the ensemble number and the state time if they want to make use of this information
  - See the "Get Expected Obs From Def" section of the obs_def documentation Website for details on how to use these values. This change is fully backwards-compatible with existing forward operator code.
- Option to output all echo of namelist values to a separate log file
  - See the utilities module documentation Website or local file for how to select where the contents of all namelists are output.
- Large file support for netCDF
  - See the Unidata netCDF documentation pages for more information about what large file support gives you and what it is compatible with.
- Better support for adaptive localization
  - See the Localization section of the assim_tools documentation Website or local file for details.
- Option to localize with different distances based on observation type
  - See the Localization section of the assim_tools documentation Website or local file for details.
- The error handler can take up to 3 lines of text so you can give more informative error messages on exit
  - See the utilities module documentation Website or local file for details.
- Option to output ensemble mean in restart file format when filter exits
  - See the filter program namelist documentation Website or local file for details.
- The start of a suite of forecast verification and evaluation tools
  - See the verification tool documentation Website or *program obs_seq_verify* for details.
- Performance improvement in the internal transposes for very large state vectors. all_vars_to_all_copies() now has a single receiver and multiple senders, which is much faster than the converse.
- Better support for users who redefine R8 to be R4, so that filter runs in single precision. Fixed code which was technically correct but numerically unstable in single precision when computing variance and covariances.
- Fixed a case in the 3D sphere locations code which made it possible that some observations and state variables at higher latitudes might not be impacted by observations which were barely within the localization cutoff.
- The observation type table at the top of all obs_seq files now only contains the types actually found in the file.
- When one or more ensemble members fail to compute a valid forward operator, the prior and/or posterior mean and standard deviation will be set to MISSING_R8 in the output obs_seq.final file in addition to setting the DART QC flag.
- Use less stack space by allocating large arrays instead of declaring them as local (stack) variables in routines
- The copyright has changed from GPL (GNU) to an NCAR-specific one which is found here.

6.15.17 New models

- POP Ocean Model
  - DART interface documentation Website or *POP*. Documentation for the model itself in CESM and standalone version from Los Alamos.
- NCOMMAS Mesoscale Atmospheric Model
  - DART interface documentation Website or *NCOMMAS*. Documentation for the model itself from NSSL, Norman, OK. is at NCOMMAS.
- COAMPS Atmosphere Model
  - Dart interface documentation Website or *COAMPS Nest*. Documentation for the model itself is at COAMPS. The original version of the COAMPS interface code and scripts was contributed by Tim Whitcomb, NRL, Monterey. An updated version was contributed by Alex Reinecke, NRL, Monterey. The primary differences between the current version and the original COAMPS model code are:
    - the ability to assimilate nested domains
    - assimilates real observations
    - a simplified way to specify the state vector
    - I/O COAMPS data files
    - extensive script updates to accommodate additional HPC environments
- NOGAPS Global Atmosphere Model
  - The Navy's operational global atmospheric prediction system. See here for an overview of the 4.0 version of the model. For more information on the NOGAPS/DART system, contact Jim Hansen, jim.hansen at nrlmry.navy.mil
- AM2 Atmosphere Model
  - Dart interface documentation Website or *AM2*. The GFDL atmosphere model documentation is at AM2.
- MIT Global Ocean Model
  - Dart interface documentation Website or *MITgcm_ocean*. The ocean component of the MIT global model suite.
- Simple Advection Model
  - Dart interface documentation Website or *Simple advection*. A simple model of advecting tracers such as CO.
- Global/Planet WRF
  - A version of the WRF weather model adapted for global use or for other planets.
- TIEgcm Thermosphere/Ionosphere Model
  - Dart interface documentation Website or *TIEGCM*. Documentation for the thermosphere and ionosphere model from the NCAR HAO (High Altitude Observatory) Division is at TIEgcm.

The `DART/models/template` directory contains sample files for adding a new model. See this section of the DART web pages for more help on adding a new model.

6.15.18 Changed models

- WRF
  - The WRF fields in the DART state vector are namelist settable, with the order of the names in the namelist controlling the order in the state vector. No assumptions are made about number of moist variables; all WRF fields must be listed explicitly. The conversion tools dart_to_wrf and wrf_to_dart (Documented here Website) use this same namelist, so it is simpler to avoid mismatches between the DART restart files and what the WRF model_mod is expecting.
  - Support for the single column version of WRF has been incorporated into the standard WRF model_mod.
  - advance_model.csh script reworked by Josh Hacker, Ryan Torn, and Glen Romine to add function and simplify the script. It now supports a restart-file-per-member, simplifies the time computations by using the advance_time executable, and handles boundary files more cleanly. Plus added many additional comments, and ways to select various options by setting shell variables at the top of the script.
  - Updates from Tim and Glen: - Changed the variable name for the longitude array to better match that used in WRF: XLON_d0* to XLONG_d0* - Added the staggered coordinate variables (XLONG_U_d0*, XLAT_U_d0*, XLONG_V_d0*, XLAT_V_d0*, ZNW_d0*) - Use the staggered variables to look up point locations when interpolating in a staggered grid. Old code used to compute the staggered points from the unstaggered grid, which was slightly inaccurate. - Added additional attribute information, supporting long_name, description (same info as long_name which is the standard, but WRF calls this attribute 'description'), units (previously supported) and named coordinates for the X and Y directions (in keeping with WRF, we do not name the vertical coordinate).
  - New scripts to generate LBC (lateral boundary condition) files for WRF runs.
- CAM
  - support for versions 4 and 5 of CAM, including tar files of changes that must be made to the CAM source tree and incorporated into the CAM executable
  - support leap years
  - use CLM restart file instead of initial file
  - various scripting changes to support archiving
  - save information from CAM for ocean and land forcing
  - scripts to archive months of obs_seq.finals
  - Added the changes needed to the CAM build tree for CAM 4.0.x
  - Updates to CAM documentation to bring it in sync with the current code.
  - All trans routines replaced with: dart_to_cam, cam_to_dart, and advance_time.
  - Minor changes to CAM model_mod, including adding a routine to write out the times file so utilities can call it in a single location, plus additional optional arg on write routine.
  - Most debugging output is off by default; a new namelist item 'print_details' will re-enable the original output.
  - Added build support for new tools (closest member, common subset, fill inflation) and removed for obsolete (merge obs).
  - The original 'trans' build files and src are now in a 'deprecated' directory so if users need them for backwards compatibility, they are still available.
  - The archive scripts are updated for the HPSS (hsi) and the MSS versions (msrcp) are removed.
  - The shell_scripts and full_experiment scripts are updated.

- Lorenz 2004/2005
  - Fixed a bug in the model advance code which was doing an extra divide by 2, causing incorrect results.

6.15.19 New observation types/sources

- MADIS Converters for METAR, Mesonet, Rawinsondes, ACARS, Marine, and Satellite Wind observations. Optionally output moisture obs as specific humidity, relative humidity, and/or dewpoint obs. Documentation Website .
- SSEC Convert Satellite Wind obs to DART format. Documentation Website .
- AIRS Satellite observed Temperature and Moisture obs. Documentation Website .
- QUIKscat Satellite observed surface winds. Documentation Website .
- GTSPP Ocean obs. Documentation Website .
- WOD World Ocean Database (currently 2009) Temperature and Salinity obs. Documentation Website .
- CODAR High frequency radar obs of ocean surface velocity. Documentation Website or local file.
- VAR Little-r and radar obs. Documentation Website .
- Text Reads text data files, a good template for converting obs stored in files without some kind of data library format (netCDF, HDF, etc). Documentation Website .
- Altimeter Altimeter observation type available from a variety of sources. Forward operator code Website or local file.
- Dewpoint Dewpoint observation type available from a variety of sources. Forward operator code Website or local file.
- Dropsonde Dropsonde observation type available to allow these observations to be distinguished from standard Radiosondes. Type defined in code Website or local file.
- TES Radiances TES satellite radiance observations of Mars. Forward operator code Website or local file.
- Hurricane/Tropical Storm Vortex Position Storm location, minimum central pressure, and maximum windspeed. Currently only implemented in the WRF model_mod interface code. Code Website or *WRF*.

All the observation converters have moved to their own top level directory `observations`. See the overview documentation Website for general information on creating observation files for use in the ensemble assimilation system.

The GPS forward operators aren't new with this release, but the code has been revised several times. In particular, there is a new namelist to set the maximum number of GPS obs supported in a single execution of filter or the obs diag program. Generally the default value is large enough for anything less than a couple days, but if you are running a month or longer of diagnostics for a time series you can easily exceed the compiled in maximum. See the GPS documentation for creating GPS observation files Website or *GPS Observations*, and the forward operator documentation Website or *MODULE obs_def_gps_mod*. There are also heavily revised scripts which download and convert multiple days of GPS obs at a time, with options to delete downloaded files automatically. The scripts are able to download GPS RO observations from any of about seven available satellites (in addition to the COSMIC array) from the CDAAC web site.

There are two modules to set observation error values when creating new observation sequence files. One contains the default values used by NCEP, and the other contains the values used by ECMWF. See the README file Website or local file for more details.

The radar module was completely overhauled and the namelist changed substantially. See the item above in the non-backward compatible changes section for details.

The scripting for converting NCEP prepbufr observations has been improved. There are options to enable or disable the 'blocking' conversion, to create 6 hour or daily output files, to swap bytes for little-endian machines, and to run up to a month of conversions in parallel if you have parallel hardware available.

There is a `DART/observations/utilities` directory where generic utilities can be built which are not dependent on any particular model.

6.15.20 New diagnostics and documentation

Better Web Pages. We've put a lot of effort into expanding our documentation. For example, please check out the Matlab diagnostics section or the pages outlining the observation sequence file contents.

But there's always more to add. **Please let us know where we are lacking.**

Other new stuff:

- There is now a main `index.html` file (Website) in the DART distribution to quickly guide you to any of the documentation for the routines or modules.
- DART_LAB Interactive Matlab GUI experiments and Powerpoint presentation of fundamental assimilation concepts Website or *DART_LAB Tutorial*.
- link_obs.m Allows one to view multiple observation attributes simultaneously and dynamically select subsets of observations in one view and have those same obs highlighted in the other views. Commonly called 'data brushing'. Matlab source Website or local file.
- obs_diag The `obs_diag` program has undergone extensive revision. User-defined levels for all coordinate (height/pressure/etc), arbitrary number of regions, the inclusion of separate copies for all DART QC values, can creates rank histograms from the `obs_seq.final` files, if possible, and more. See the documentation Website .
- Comparing two (or more) experiments Matlab scripts to compare **multiple** (not just two) `obs_diag_output.nc` files on the same graphic to allow for easy examination of experiment attributes (rmse, biases, etc.). Some new utilities for subsetting observation sequence files in order to make fair comparisons are described below. Matlab source for `two_experiments_profile.m` Website or local file and `two_experiments_evolution.m` Website or local file.
- netCDF and Matlab The DART Matlab routines no longer depend on 4 third-party toolboxes; we are down to just mexnc and snctools. Soon, we may just use snctools! See the documentation for how to configure Matlab to run the DART-provided scripts Website or local file.
- Matlab support for CAM. CAM is now fully supported for all the Matlab interfaces that are used in the demos - this includes the state-space tools in `DART/matlab` that allow for determining correlations among state variables, among other things.
- Matlab support for WRF. WRF is now fully supported for all the Matlab interfaces that are used in the demos. This predominantly includes the state-space tools in the `DART/matlab` directory like `plot_total_err`. The `map_wrf.m` script (Website or local file) can finally plot WRF fields now that the required metadata is part of the `Posterior_Diag.nc`, `Prior_Diag.nc`, and (not required) `True_State.nc` files. It's a small step to augment this routine to make publication-quality figures of WRF fields.
- Regression tests for WRF WRF test cases for WRF V2 and V3 for CONUS (Continental or Contiguous United States), a Global WRF case, and a Radar test case. The data files are on a web server because they are too large to add to the repository. The README files in each directory gives instructions on how to download them. Website or local file.
- Other New Model Support The `simple_advection` and `MITgcm_ocean` are fully supported in the Matlab diagnostics.

- Better execution traces Optional detailed execution trace messages from filter by setting the namelist variable `trace_execution`. See the details of the filter namelist Website or *PROGRAM filter* .
- `input.nml` contents saved The contents of the `input.nml` namelist file are now preserved in the `True_State.nc`, `Prior_Diag.nc`, and `Posterior_Diag.nc` diagnostic files in variable `inputnml`.
- Better error checking in obs_sequence creation subroutines to avoid out-of-time-order observations being inserted by incorrect programs.
- Better error checking in `open_file()` Better error checking in the `utilities_mod` subroutine `open_file()`. See documentation Website or local file.
- In the DART code tree, individual html pages have links back to the index page, the namelists are moved up to be more prominent, and have other minor formatting improvements.
- The following Matlab observation-space diagnostic routines have been **removed**:

fit_ens_mean_time.m	plotted the temporal evolution of the ensemble mean of some quantity.
fit_ens_spread_time.m	plotted the temporal evolution of the ensemble spread of some quantity.
fit_mean_spread_time.m	plotted the temporal evolution of the mean and spread of some quantity.
obs_num_time.m	plotted the temporal evolution of the observation density.
fit_ens_mean_vertical.m	plotted the vertical profile of the ensemble mean of some quantity.
fit_ens_bias_vertical.m	plotted the vertical profile of the bias of the ensemble mean of some quantity.
obs_num_vertical.m	plotted the vertical profile of the observation density.

- The following Matlab observation-space diagnostic routines have been **added**:

plot_profile	plots the vertical profile of any quantity for any copy with an overlay of the observation density and number of observations assimilated. Matlab source Website or local file.
plot_rmse_xxx_profile	plots the vertical profile of the rmse and any quantity for any copy with an overlay of the observation density and number of observations assimilated. Matlab source Website or local file.
plot_bias_xxx_profile	plots the vertical profile of the bias and any quantity for any copy with an overlay of the observation density and number of observations assimilated. Matlab source Website or local file.
two_experiments_profile	plots the vertical profile of any quantity for any copy for multiple experiments with an overlay of the observation density and number of observations assimilated in each experiment. Matlab source Website or local file.
plot_evolution	plots the temporal evolution of any quantity for any copy with an overlay of the observation density and number of observations assimilated. Matlab source Website or local file.
plot_rmse_xxx_evolution	plots the temporal evolution of the rmse and any quantity for any copy with an overlay of the observation density and number of observations assimilated. Matlab source Website or local file.
two_experiments_evolution	plots the temporal evolution for any quantity for any copy for multiple experiements with an overlay of the observation density and number of observations assimilated in each experiment. Matlab source Website or local file.
read_obs_netcdf	reads a netCDF format observation sequence file. Simply need a single copy and a single qc - no actual observation required. Matlab source Website or local file.
plot_obs_netcdf	reads and plots the locations and values of any copy of the observations in a DART netCDF format observation sequence file. Matlab source Website or local file.
plot_obs_netcdf_diffs	reads and plots the locations and the difference of any two copies of the observations in a DART netCDF format observation sequence file. Matlab source Website or local file.
plot_wind_vectors	reads and plots the wind vectors of the observations in a DART netCDF format observation sequence file (created by `obs_seq_to_netcdf`, documentation Website or *PROGRAM obs_seq_to_netcdf*) Matlab source Website or local file.
link_obs	a data brushing tool. Explores many facets of the observations simultaneously. Multiple plots allow groups of observations to be selected in one view and the corresponding observations are indicated in all the other views. Matlab source Website or local file.
plot_rank_histogram	If the individual ensemble member observation values were output from `filter` (selected by namelist option in the filter namelist) into the `obs_seq.final` file, `obs_diag` will create rank histogram information and store it in the `obs_diag_output.nc` file. `plot_rank_histogram.m` will then plot it. There are instructions on how to view the results with `ncview` or with this Matlab script on the DART Observation-space Diagnos tics web page. Matlab source Website or local file.

6.15.21 New utilities

- obs_seq_to_netcdf Any DART observation sequence may be converted to a netCDF format file. All information in the sequence file is preserved EXCEPT for any observations with additional user-added metadata, e.g. Radar obs, GPS RO obs for the non-local operator. But all core observation data such as location, time, type, QC, observation value and error will be converted. This allows for variety of new diagnostics. Documentation Website or *PROGRAM obs_seq_to_netcdf*.

- obs_seq_coverage A step towards determining what locations and quantities are repeatedly observed during a specific time interval. This may be used to determine a network of observations that will be used to verify forecasts. Documentation Website or *program obs_seq_coverage*.

- obs_selection An optional companion routine to `obs_seq_coverage`. This thins the observation sequence files to contain just the desired set of observations to use in the forecast step. This speeds performance by avoiding the cost of evaluating observations that will not be used in the verification. Documentation Website or *program obs_selection*.

- obs_seq_verify is a companion routine to `obs_seq_coverage`. This creates a netCDF file with variables that should make the calculation of skill scores, etc. easier. It creates variables of the form: `METAR_U_10_METER_WIND(analysisT, stations, levels, copy, nmembers, forecast_lead)` Documentation Website or *program obs_seq_verify*.

- Select common observation subsets A tool that operates on two (will be extended to more) `obs_seq.final` files which were output from two different runs of filter. Assumes the same `obs_seq.out` input file was used in both cases. Outputs two new `obs_seq.final.new` files containing only the observations which were assimilated in both experiments. It allows for a fair comparision with the diagnostic tools. Documentation Website or *program obs_common_subset*.

- Restart File tool Generic tool that works on any DART restart file. It is compiled with the corresponding model_mod which tells it how large the state vector is. It can alter the timestamps on the data, add or remove model advance times, split a single file into 1-per-ensemble or the reverse, and can be used to convert between ASCII and binary formats. Documentation Website or *PROGRAM restart_file_tool*.

- Advance Time tool A generic utility for adding intervals to a Gregorian calendar date and printing out the new date, including handling leap year and month and year rollovers. An earlier version of this program was taken from the WRF distribution. This version maintains a similar interface but was completely rewritten to use the DART time manager subroutines to do the time computations. It reads from the console/standard input to avoid trying to handle command line arguments in a compiler-independent manner, and outputs in various formats depending on what is requested via additional flags. Documentation Website or *PROGRAM advance_time*.

- WRF observation preprocessor tool Observation preprocessor which is WRF aware, contributed by Ryan Torn. Will select obs only within the WRF domain, will superob, will select only particular obs types based on the namelist. Source is in the `DART/models/wrf/WRF_DART_utilities` directory.

- Closest Member tool Used in combination with the new option in filter to output the ensemble mean values in a DART restart file format, this tool allows you to select the N *closest* members, where there are multiple choices for how that metric is computed. There are also ways to select a subset of the state vector by item kind as returned from the `get_state_meta_data()` routine from the corresponding model interface code in `model_mod.f90` (see subroutine documentation Website or local file) and compute the metric based only on those values. Tool documentation Website or *PROGRAM closest_member_tool*.

- Fill Inflation restart file tool Simple tool that creates an inflation restart file with constant initial inflation and standard deviation values. Often the first step of a multi-step assimilation job differs in the namelist only for how the initial inflation values are defined. Running this tool creates the equivalent of an IC file for inflation, so the first job step can start from a restart file as all subsequent job steps do and allows the use of a single `input.nml` file. Documentation Website or *PROGRAM fill_inflation_restart*.

- Replace WRF fields tool WRF-specific tool that copies netCDF variables from one file to another. The field must exist in the target file and the data will be overwritten by data from the source file. Field names to be copied can be specified directly in the namelist or can be listed in a separate file. Missing fields can be ignored or cause the program to stop with a fatal error depending on namelist settings. Documentation Website or *PROGRAM replace_wrf_fields*.

- model_mod Verification/Check tool Tool to help when creating a new model interface file (usually named `model_mod.f90`). Calls routines to help with debugging. Documentation Website or *program model_mod_check*.

Minor items:

- Most tools which work with observation sequence files now have a namelist option to specify the input files in one of two methods: an explicit list of input obs_seq files, or the name of a file which contains the list of obs_seq files.

- The `DART/shell_scripts` directory contains example scripts which loop over multiple days, in formats for various shell syntaxes. They are intended as an example for use in advance_model or job scripts, or observation conversion programs contributed by users.

6.15.22 Known problems

We get an internal compiler error when compiling the `obs_diag` program on a Linux machine using the gfortran compiler version 4.1.2. If you get this error try a newer version of the Gnu compiler tools. We have used 4.3 and 4.4 successfully.

6.16 Jamaica

6.16.1 DART Jamaica release documentation

Attention: Jamaica is a prior release of DART. Its source code is available via the DART repository on Github. This documentation is preserved merely for reference. See the DART homepage to learn about the latest release.

6.16.2 Overview of DART

The Data Assimilation Research Testbed (DART) is designed to facilitate the combination of assimilation algorithms, models, and **real** (as well as synthetic) observations to allow increased understanding of all three. The DART programs have been compiled with several (many?) Fortran 90 compilers and run on linux compute-servers, linux clusters, OSX laptops/desktops, SGI Altix clusters, supercomputers running AIX … a pretty broad range, really. You should definitely read the Customizations section.

DART employs a modular programming approach to apply an Ensemble Kalman Filter which nudges models toward a state that is more consistent with information from a set of observations. Models may be swapped in and out, as can different algorithms in the Ensemble Kalman Filter. The method requires running multiple instances of a model to generate an ensemble of states. A forward operator appropriate for the type of observation being used is applied to each of the states to generate the model's estimate of the observation. Comparing these estimates and their uncertainty to the observation and its uncertainty ultimately results in the adjustments to the model states. There's much more to it, described in detail in the tutorial directory of the package.

DART ultimately creates a few netCDF files containing the model states just before the adjustment (`Prior_Diag.nc`) and just after the adjustment (`Posterior_Diag.nc`) as well as a file `obs_seq.final` with the model estimates of the observations. There is a suite of Matlab® functions that facilitate exploration of the results, but the netCDF files are inherently portable and contain all the necessary metadata to interpret the contents.

6.16.3 What's new

The Jamaica release has been tested on several supercomputers, including the NCAR IBM machines 'bluevista' and 'blueice', the SGI Altix machines 'columbia' and 'origin'. There are **many** more changes - much more thoroughly described in the Jamaica_diffs_from_I document. What follows here is only meant to draw attention to the fact that experienced users should really read the change document.

The most visible changes in the Jamaica release are as follows.

Changes in the inflation strategies

The changes in the inflation mechanism now provide the ability to perform inflation in observation-space, fixed-state-space, or spatially-varying-state-space in the prior and/or posterior with either a deterministic or stochastic algorithm.

Fundamental changes for parallel behavior

The **optional** use of MPI required a complete rewrite of many of the routines that previously governed attempts at parallel computations. This resulted in the removal of the `async = 3` method of achieving parallel assimilations and model advances. As a consequence, `assim_region.csh`, `advance_ens.csh`, and `filter_server.csh` no longer exist.

Some assimilation strategies previously controlled by `assim_tools_nml`, (namely, whether to perform a parallel assimilation, how many domains to use, and a few others) have been replaced by a new parallel assimilation algorithm. Associated inflation computations are documented in the manuscript by Anderson and Collins that is to appear in the Journal of Atmospheric and Oceanic Technology and in section 22 of the tutorial. Namelist entries `do_parallel`, `num_domains`, and `parallel_command` no longer exist.

The (optional) MPI implementation allows for very large state vectors - some model configurations (i.e. state vectors too large to fit in core) are now possible. And **yes**, it scales rather nicely, thank you. There are references throughout the documents and code to MPI - keep in mind all the MPI interfaces are in one module `mpi_utilities_mod.f90`. If you are NOT using MPI - there is a simple equivalent module named `null_mpi_utilities_mod.f90` which is the default.

Using MPI greatly increases the performance for large models and/or large numbers of observations. It also greatly **decreases** the shell script complexity. The use of MPI is enabled by giving `mkmf_filter` an optional argument `-mpi`.

Vertical localization

It is now possible to perform localization in the vertical in 3D models, which necessitated a change in the `model_mod` interface. The `model_mod:get_close_states()` and `model_mod:get_num_close_states()` routines have been replaced by “pass-through” interfaces to routines in `location_mod`. An optional interface `model_mod:ens_mean_for_model()` is required only by some models when performing localization in the vertical. This routine provides the ensemble mean so that the same model state is used when computing the vertical localization.

3d plots of observation locations & type

It’s more than just eye candy. It is also possible to plot (in 3D) the locations of observations by processing an observation sequence through `obs_diag` and running the resulting output file through the matlab function `plot_observation_locations`.

6.16.4 What's required

1. a working fortran compiler
2. a working F90 interface to the netCDF libraries
3. something that can run 'csh'

DART has been tested on many Fortran compilers and platforms. Our philosophy seems to be to burn away the 'impurities' by compiling on any architecture we can get our hands on. We completely buy into the use of the Fortran 'kind' strategy to declare variable types rather than relying on compiler autopromotion flags - which are a BAD idea in my opinion. There are some models in DART that come from outside developers and _may_ contain code segments that require the use of some special compiler flags - we are not so draconian as to mandate the use of the Fortran 'kind'. In general, the low-order models and all common portions of the DART code compile very cleanly.

DART uses the netCDF self-describing data format with a particular metadata convention to describe output that is used to analyze the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools.

Since most (all?) of the models envisioned being used by DART are written in Fortran and run on various UNIX or *nix platforms, the development environment for DART is highly skewed to these machines. We do most of our development on a small linux workstation and a mac laptop running OSX 10.4.9 - but have an extensive test network. I've never built nor run DART on a Windows machine - so I don't even know if it's possible. If you have run it (under Cygwin?) please let me know how it went – I'm curious.

6.16.5 What's nice to have

- **ncview**: DART users have used ncview to create graphical displays of output data fields. The 2D rendering is good for 'quick-look' type uses, but I wouldn't want to publish with it.
- **NCO**: The NCO tools are able to perform operations on netCDF files like concatenating, slicing, and dicing.
- **Matlab®**: A set of Matlab® scripts, designed to produce graphical diagnostics from DART netCDF output files are also part of the DART project.
- **MPI**: The latest release of the DART system includes an MPI option. MPI stands for 'Message Passing Interface', and is both a library and run-time system that enables multiple copies of a single program to run in parallel, exchange data, and combine to solve a problem more quickly. The latest release of DART does *NOT* require MPI to run; the default build scripts do not need nor use MPI in any way. However, for larger models with large state vectors and large numbers of observations, the data assimilation step will run much faster in parallel, which requires MPI to be installed and used. However, if multiple ensembles of your model fit comfortably (in time and memory space) on a single processor, you need read no further about MPI.

6.16.6 Where's the (gui) interface?

DART programs can require three different types of input. First, some of the DART programs, those for creating synthetic observational datasets, require interactive input from the keyboard. For simple cases, this interactive input can be made directly from the keyboard. In more complicated cases, a file containing the appropriate keyboard input can be created and this file can be directed to the standard input of the DART program. Second, many DART programs expect one or more input files in DART specific formats to be available. For instance, `perfect_model_obs`, which creates a synthetic observation set given a particular model and a description of a sequence of observations, requires an input file that describes this observation sequence. At present, the observation files for DART are in a custom format in either human-readable ascii or more compact machine-specific binary. Third, many DART modules (including main programs) make use of the Fortan90 namelist facility to obtain values of certain parameters at run-time. All programs look for a namelist input file called `input.nml` in the directory in which the program is executed. The `input.nml` file can contain a sequence of individual Fortran90 namelists which specify values of particular parameters for modules that compose the executable program. DART provides a mechanism that automatically generates namelists with the default values for each program to be run.

6.16.7 Installation

This document outlines the installation of the DART software and the system requirements. The entire installation process is summarized in the following steps:

1. Determine which F90 compiler is available.
2. Determine the location of the `netCDF` library.
3. Download the DART software into the expected source tree.
4. Modify certain DART files to reflect the available F90 compiler and location of the appropriate libraries.
5. Build the executables.

We have tried to make the code as portable as possible, but we do not have access to all compilers on all platforms, so there are no guarantees. We are interested in your experience building the system, so please email me (Tim Hoar) thoar 'at' ucar 'dot' edu (trying to cut down on the spam).

After the installation, you might want to peruse the following.

- Running the Lorenz_63 Model.
- Using the Matlab® diagnostic scripts.
- A short discussion on bias, filter divergence and covariance inflation.
- And another one on synthetic observations.

Requirements: an F90 compiler

The DART software has been successfully built on several Linux/x86 platforms with several versions of the Intel Fortran Compiler for Linux, which (at one point) is/was free for individual scientific use. It has also been built and successfully run with several versions of each of the following: Portland Group Fortran Compiler, Lahey Fortran Compiler, Pathscale Fortran Compiler, GNU Fortran 95 Compiler ("gfortran"), Absoft Fortran 90/95 Compiler (Mac OSX). Since recompiling the code is a necessity to experiment with different models, there are no binaries to distribute.

DART uses the netCDF self-describing data format for the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. In particular, DART also makes use of the F90 interface to the library which is available through the `netcdf.mod` and `typesizes.mod` modules. *IMPORTANT*: different compilers create these modules with different "case" filenames, and sometimes they are not **both** installed into the expected directory. It is required that both modules be present. The normal place would be in the `netcdf/include` directory, as opposed to the `netcdf/lib` directory.

If the netCDF library does not exist on your system, you must build it (as well as the F90 interface modules). The library and instructions for building the library or installing from an RPM may be found at the netCDF home page: http://www.unidata.ucar.edu/packages/netcdf/ Pay particular attention to the compiler-specific patches that must be applied for the Intel Fortran Compiler. (Or the PG compiler, for that matter.)

The location of the netCDF library, `libnetcdf.a`, and the locations of both `netcdf.mod` and `typesizes.mod` will be needed by the makefile template, as described in the compiling section.

Downloading the distribution

This release of the DART source code can be downloaded as a compressed zip or tar.gz file. When extracted, the source tree will begin with a directory named `DART` and will be approximately 221.7 Mb. Compiling the code in this tree (as is usually the case) will necessitate much more space.

```
$ gunzip DART-6.0.0.tar.gz
$ tar -xvf DART-6.0.0.tar
```

You should wind up with a directory named `DART`.

The code tree is very "bushy"; there are many directories of support routines, etc. but only a few directories involved with the customization and installation of the DART software. If you can compile and run ONE of the low-order models, you should be able to compile and run ANY of the low-order models. For this reason, we can focus on the Lorenz `63 model. Subsequently, the only directories with files to be modified to check the installation are: `DART/mkmf`, `DART/models/lorenz_63/work`, and `DART/matlab` (but only for analysis).

Customizing the build scripts – overview

DART executable programs are constructed using two tools: `make` and `mkmf`. The `make` utility is a very common piece of software that requires a user-defined input file that records dependencies between different source files. `make` then performs a hierarchy of actions when one or more of the source files is modified. The `mkmf` utility is a custom preprocessor that generates a `make` input file (named `Makefile`) and an example namelist *input.nml.program_default* with the default values. The `Makefile` is designed specifically to work with object-oriented Fortran90 (and other languages) for systems like DART.

`mkmf` requires two separate input files. The first is a `template' file which specifies details of the commands required for a specific Fortran90 compiler and may also contain pointers to directories containing pre-compiled utilities required by the DART system. **This template file will need to be modified to reflect your system**. The second input file is a `path_names' file which includes a complete list of the locations (either relative or absolute) of all Fortran90 source files that are required to produce a particular DART program. Each 'path_names' file must contain a path for exactly one Fortran90 file containing a main program, but may contain any number of additional paths pointing to files containing Fortran90 modules. An `mkmf` command is executed which uses the 'path_names' file and the mkmf template file to produce a `Makefile` which is subsequently used by the standard `make` utility.

Shell scripts that execute the mkmf command for all standard DART executables are provided as part of the standard DART software. For more information on `mkmf` see the FMS mkmf description.

One of the benefits of using `mkmf` is that it also creates an example namelist file for each program. The example namelist is called *input.nml.program_default*, so as not to clash with any exising `input.nml` that may exist in that directory.

6.16.8 Building and customizing the 'mkmf.template' file

A series of templates for different compilers/architectures exists in the `DART/mkmf/` directory and have names with extensions that identify the compiler, the architecture, or both. This is how you inform the build process of the specifics of your system. Our intent is that you copy one that is similar to your system into `mkmf.template` and customize it. For the discussion that follows, knowledge of the contents of one of these templates (i.e. `mkmf.template.gfortran`) is needed. Note that only the LAST lines are shown here, the head of the file is just a big comment (worth reading, btw).

```
MPIFC = mpif90
MPILD = mpif90
FC = gfortran
```

(continues on next page)

(continued from previous page)

```
LD = gfortran
NETCDF = /usr/local
INCS = ${NETCDF}/include
FFLAGS = -O2 -I$(INCS)
LIBS = -L${NETCDF}/lib -lnetcdf
LDFLAGS = -I$(INCS) $(LIBS)
```

Essentially, each of the lines defines some part of the resulting `Makefile`. Since `make` is particularly good at sorting out dependencies, the order of these lines really doesn't make any difference. The `FC = gfortran` line ultimately defines the Fortran90 compiler to use, etc. The lines which are most likely to need site-specific changes start with `FFLAGS` and `NETCDF`, which indicate where to look for the netCDF F90 modules and the location of the netCDF library and modules.

If you have MPI installed on your system `MPIFC, MPILD` dictate which compiler will be used in that instance. If you do not have MPI, these variables are of no consequence.

Netcdf

Modifying the `NETCDF` value should be relatively straightforward. Change the string to reflect the location of your netCDF installation containing `netcdf.mod` and `typesizes.mod`. The value of the `NETCDF` variable will be used by the `FFLAGS, LIBS,` and `LDFLAGS` variables.

FFLAGS

Each compiler has different compile flags, so there is really no way to exhaustively cover this other than to say the templates as we supply them should work – depending on the location of your netCDF. The low-order models can be compiled without a `-r8` switch, but the `bgrid_solo` model cannot.

Customizing the 'path_names_*' file

Several `path_names_*` files are provided in the `work` directory for each specific model, in this case: `DART/models/lorenz_63/work`. Since each model comes with its own set of files, the `path_names_*` files need no customization.

The tutorial

The `DART/tutorial` documents are an excellent way to kick the tires on DART and learn about ensemble data assimilation. If you have correctly configured your `mkmf.template`, you can run anything in the tutorial.

6.16.9 Building the Lorenz_63 DART project

Currently, DART executables are constructed in a `work` subdirectory under the directory containing code for the given model. In the top-level DART directory, change to the L63 work directory and list the contents:

```
$ cd DART/models/lorenz_63/work
$ ls -1
```

With the result:

```
Posterior_Diag.nc
Prior_Diag.nc
True_State.nc
filter_ics
filter_restart
input.nml
mkmf_create_fixed_network_seq
mkmf_create_obs_sequence
mkmf_filter
mkmf_merge_obs_seq
mkmf_obs_diag
mkmf_perfect_model_obs
mkmf_preprocess
mkmf_wakeup_filter
obs_seq.final
obs_seq.in
obs_seq.out
obs_seq.out.average
obs_seq.out.x
obs_seq.out.xy
obs_seq.out.xyz
obs_seq.out.z
path_names_create_fixed_network_seq
path_names_create_obs_sequence
path_names_filter
path_names_merge_obs_seq
path_names_obs_diag
path_names_perfect_model_obs
path_names_preprocess
path_names_wakeup_filter
perfect_ics
perfect_restart
set_def.out
workshop_setup.csh
```

There are eight mkmf_*xxxxxx* files for the programs

1. preprocess,
2. create_obs_sequence,
3. create_fixed_network_seq,
4. perfect_model_obs,
5. filter,
6. wakeup_filter,
7. merge_obs_seq, and
8. obs_diag,

along with the corresponding path_names_*xxxxxx* files. There are also files that contain initial conditions, netCDF output, and several observation sequence files, all of which will be discussed later. You can examine the contents of one of the path_names_*xxxxxx* files, for instance path_names_filter, to see a list of the relative paths of all files that contain Fortran90 modules required for the program filter for the L63 model. All of these paths are relative to your DART directory. The first path is the main program (filter.f90) and is followed by all the Fortran90 modules used by this program (after preprocessing).

The mkmf_*xxxxxx* scripts are cryptic but should not need to be modified – as long as you do not restructure the code tree (by moving directories, for example). The only function of the `mkmf_`*xxxxxx* script is to generate a `Makefile` and an *input.nml.program_default* file. It is not supposed to compile anything – `make` does that:

```
$ csh mkmf_preprocess
$ make
```

The first command generates an appropriate `Makefile` and the `input.nml.preprocess_default` file. The second command results in the compilation of a series of Fortran90 modules which ultimately produces an executable file: `preprocess`. Should you need to make any changes to the `DART/mkmf/mkmf.template`, you will need to regenerate the `Makefile`.

The `preprocess` program actually builds source code to be used by all the remaining modules. It is **imperative** to actually **run** `preprocess` before building the remaining executables. This is how the same code can assimilate state vector 'observations' for the Lorenz_63 model and real radar reflectivities for WRF without needing to specify a set of radar operators for the Lorenz_63 model!

`preprocess` reads the `&preprocess_nml` namelist to determine what observations and operators to incorporate. For this exercise, we will use the values in `input.nml`. `preprocess` is designed to abort if the files it is supposed to build already exist. For this reason, it is necessary to remove a couple files (if they exist) before you run the preprocessor. It is just a good habit to develop.

```
$ \rm -f ../../../obs_def/obs_def_mod.f90
$ \rm -f ../../../obs_kind/obs_kind_mod.f90
$ ./preprocess
$ ls -l ../../../obs_def/obs_def_mod.f90
$ ls -l ../../../obs_kind/obs_kind_mod.f90
```

This created `../../../obs_def/obs_def_mod.f90` from `../../../obs_kind/DEFAULT_obs_kind_mod.F90` and several other modules. `../../../obs_kind/obs_kind_mod.f90` was created similarly. Now we can build the rest of the project.

A series of object files for each module compiled will also be left in the work directory, as some of these are undoubtedly needed by the build of the other DART components. You can proceed to create the other programs needed to work with L63 in DART as follows:

```
$ csh mkmf_create_obs_sequence
$ make
$ csh mkmf_create_fixed_network_seq
$ make
$ csh mkmf_perfect_model_obs
$ make
$ csh mkmf_filter
$ make
$ csh mkmf_obs_diag
$ make
```

The result (hopefully) is that six executables now reside in your work directory. The most common problem is that the netCDF libraries and include files (particularly `typesizes.mod`) are not found. Edit the `DART/mkmf/mkmf.template`, recreate the `Makefile`, and try again.

program	purpose
`preprocess`	creates custom source code for just the observations of interest
`create_obs_sequence`	specify a (set) of observation characteristics taken by a particular (set of) instruments
`create_fixed_network_seq`	specify the temporal attributes of the observation sets
`perfect_model_obs`	spinup, generate "true state" for synthetic observation experiments, . . .
`filter`	perform experiments
`obs_diag`	creates observation-space diagnostic files to be explored by the Matlab® scripts.
`merge_obs_seq`	manipulates observation sequence files. It is not generally needed (particularly for low-order models) but can be used to combine observation sequences or convert from ASCII to binary or vice-versa. Since this is a specialty routine - we will not cover its use in this document.
`wakeup_filter`	is only needed for MPI applications. We're starting at the beginning here, so we're going to ignore this one, too.

6.16.10 Running Lorenz_63

This initial sequence of exercises includes detailed instructions on how to work with the DART code and allows investigation of the basic features of one of the most famous dynamical systems, the 3-variable Lorenz-63 model. The remarkable complexity of this simple model will also be used as a case study to introduce a number of features of a simple ensemble filter data assimilation system. To perform a synthetic observation assimilation experiment for the L63 model, the following steps must be performed (an overview of the process is given first, followed by detailed procedures for each step):

6.16.11 Experiment overview

1. Integrate the L63 model for a long time starting from arbitrary initial conditions to generate a model state that lies on the attractor. The ergodic nature of the L63 system means a 'lengthy' integration always converges to some point on the computer's finite precision representation of the model's attractor.
2. Generate a set of ensemble initial conditions from which to start an assimilation. Since L63 is ergodic, the ensemble members can be designed to look like random samples from the model's 'climatological distribution'. To generate an ensemble member, very small perturbations can be introduced to the state on the attractor generated by step 1. This perturbed state can then be integrated for a very long time until all memory of its initial condition can be viewed as forgotten. Any number of ensemble initial conditions can be generated by repeating this procedure.
3. Simulate a particular observing system by first creating an 'observation set definition' and then creating an 'observation sequence'. The 'observation set definition' describes the instrumental characteristics of the observations and the 'observation sequence' defines the temporal sequence of the observations.
4. Populate the 'observation sequence' with 'perfect' observations by integrating the model and using the information in the 'observation sequence' file to create simulated observations. This entails operating on the model state at the time of the observation with an appropriate forward operator (a function that operates on the model state vector to produce the expected value of the particular observation) and then adding a random sample from the observation error distribution specified in the observation set definition. At the same time, diagnostic output about the 'true' state trajectory can be created.
5. Assimilate the synthetic observations by running the filter; diagnostic output is generated.

6.16.12 1. Integrate the L63 model for a 'long' time

`perfect_model_obs` integrates the model for all the times specified in the 'observation sequence definition' file. To this end, begin by creating an 'observation sequence definition' file that spans a long time. Creating an 'observation sequence definition' file is a two-step procedure involving `create_obs_sequence` followed by `create_fixed_network_seq`. After they are both run, it is necessary to integrate the model with `perfect_model_obs`.

1.1 Create an observation set definition

`create_obs_sequence` creates an observation set definition, the time-independent part of an observation sequence. An observation set definition file only contains the `location, type,` and `observational error characteristics` (normally just the diagonal observational error variance) for a related set of observations. There are no actual observations, nor are there any times associated with the definition. For spin-up, we are only interested in integrating the L63 model, not in generating any particular synthetic observations. Begin by creating a minimal observation set definition.

In general, for the low-order models, only a single observation set need be defined. Next, the number of individual scalar observations (like a single surface pressure observation) in the set is needed. To spin-up an initial condition for the L63 model, only a single observation is needed. Next, the error variance for this observation must be entered. Since we do not need (nor want) this observation to have any impact on an assimilation (it will only be used for spinning up the model and the ensemble), enter a very large value for the error variance. An observation with a very large error variance has essentially no impact on deterministic filter assimilations like the default variety implemented in DART. Finally, the location and type of the observation need to be defined. For all types of models, the most elementary form of synthetic observations are called 'identity' observations. These observations are generated simply by adding a random sample from a specified observational error distribution directly to the value of one of the state variables. This defines the observation as being an identity observation of the first state variable in the L63 model. The program will respond by terminating after generating a file (generally named `set_def.out`) that defines the single identity observation of the first state variable of the L63 model. The following is a screenshot (much of the verbose logging has been left off for clarity), the user input looks *like this*.

```
[unixprompt]$ ./create_obs_sequence
 Starting program create_obs_sequence
 Initializing the utilities module.
 Trying to log to unit   10
 Trying to open file dart_log.out

 Registering module :
 $url: http://squish/DART/trunk/utilities/utilities_mod.f90 $
 $revision: 2713 $
 $date: 2007-03-25 22:09:04 -0600 (Sun, 25 Mar 2007) $
 Registration complete.

 &UTILITIES_NML
 TERMLEVEL= 2,LOGFILENAME=dart_log.out

 /

 Registering module :
 $url: http://squish/DART/trunk/obs_sequence/create_obs_sequence.f90 $
 $revision: 2713 $
 $date: 2007-03-25 22:09:04 -0600 (Sun, 25 Mar 2007) $
 Registration complete.

 { ... }
```

(continues on next page)

(continued from previous page)

```
 Input upper bound on number of observations in sequence
10

 Input number of copies of data (0 for just a definition)
0

 Input number of quality control values per field (0 or greater)
0

 input a -1 if there are no more obs
0

 Registering module :
 $url: http://squish/DART/trunk/obs_def/DEFAULT_obs_def_mod.F90 $
 $revision: 2820 $
 $date: 2007-04-09 10:37:47 -0600 (Mon, 09 Apr 2007) $
 Registration complete.


 Registering module :
 $url: http://squish/DART/trunk/obs_kind/DEFAULT_obs_kind_mod.F90 $
 $revision: 2822 $
 $date: 2007-04-09 10:39:08 -0600 (Mon, 09 Apr 2007) $
 Registration complete.

 ------------------------------------------------------

 initialize_module obs_kind_nml values are

 -------------- ASSIMILATE_THESE_OBS_TYPES --------------
 RAW_STATE_VARIABLE
 -------------- EVALUATE_THESE_OBS_TYPES --------------
 ------------------------------------------------------

      Input -1 * state variable index for identity observations
      OR input the name of the observation kind from table below:
      OR input the integer index, BUT see documentation...
        1 RAW_STATE_VARIABLE

-1

 input time in days and seconds
1 0

 Input error variance for this observation definition
1000000

 input a -1 if there are no more obs
-1

 Input filename for sequence (  set_def.out   usually works well)
 set_def.out
 write_obs_seq  opening formatted file set_def.out
 write_obs_seq  closed file set_def.out
```

1.2 Create an observation sequence definition

`create_fixed_network_seq` creates an ‘observation sequence definition’ by extending the ‘observation set definition’ with the temporal attributes of the observations.

The first input is the name of the file created in the previous step, i.e. the name of the observation set definition that you’ve just created. It is possible to create sequences in which the observation sets are observed at regular intervals or irregularly in time. Here, all we need is a sequence that takes observations over a long period of time - indicated by entering a 1. Although the L63 system normally is defined as having a non-dimensional time step, the DART system arbitrarily defines the model timestep as being 3600 seconds. If we declare that we have one observation per day for 1000 days, we create an observation sequence definition spanning 24000 ‘model’ timesteps; sufficient to spin-up the model onto the attractor. Finally, enter a name for the ‘observation sequence definition’ file. Note again: there are no observation values present in this file. Just an observation type, location, time and the error characteristics. We are going to populate the observation sequence with the `perfect_model_obs` program.

```
[unixprompt]$ ./create_fixed_network_seq

 ...

 Registering module :
 $url: http://squish/DART/trunk/obs_sequence/obs_sequence_mod.f90 $
 $revision: 2749 $
 $date: 2007-03-30 15:07:33 -0600 (Fri, 30 Mar 2007) $
 Registration complete.

 static_init_obs_sequence obs_sequence_nml values are
 &OBS_SEQUENCE_NML
 WRITE_BINARY_OBS_SEQUENCE =  F,
 /
 Input filename for network definition sequence (usually  set_def.out  )
set_def.out

 ...

 To input a regularly repeating time sequence enter 1
 To enter an irregular list of times enter 2
1
 Input number of observations in sequence
1000
 Input time of initial ob in sequence in days and seconds
1, 0
 Input period of obs in days and seconds
1, 0
           1
           2
           3
...
         997
         998
         999
        1000
What is output file name for sequence (  obs_seq.in   is recommended )
obs_seq.in
 write_obs_seq  opening formatted file obs_seq.in
 write_obs_seq closed file obs_seq.in
```

1.3 Initialize the model onto the attractor

perfect_model_obs can now advance the arbitrary initial state for 24,000 timesteps to move it onto the attractor.

perfect_model_obs uses the Fortran90 namelist input mechanism instead of (admittedly gory, but temporary) interactive input. All of the DART software expects the namelists to found in a file called input.nml. When you built the executable, an example namelist was created input.nml.perfect_model_obs_default that contains all of the namelist input for the executable. If you followed the example, each namelist was saved to a unique name. We must now rename and edit the namelist file for perfect_model_obs. Copy input.nml.perfect_model_obs_default to input.nml and edit it to look like the following: (just worry about the highlighted stuff - and whitespace doesn't matter)

```
cp input.nml.perfect_model_obs_default input.nml
```

```
&perfect_model_obs_nml
   start_from_restart    = .false.,
   output_restart        = .true.,
   async                 = 0,
   init_time_days        = 0,
   init_time_seconds     = 0,
   first_obs_days        = -1,
   first_obs_seconds     = -1,
   last_obs_days         = -1,
   last_obs_seconds      = -1,
   output_interval       = 1,
   restart_in_file_name  = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   obs_seq_in_file_name  = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   adv_ens_command       = "./advance_ens.csh"  /

&ensemble_manager_nml
   single_restart_file_in  = .true.,
   single_restart_file_out = .true.,
   perturbation_amplitude  = 0.2  /

&assim_tools_nml
   filter_kind                     = 1,
   cutoff                          = 0.2,
   sort_obs_inc                    = .false.,
   spread_restoration              = .false.,
   sampling_error_correction       = .false.,
   adaptive_localization_threshold = -1,
   print_every_nth_obs             = 0  /

&cov_cutoff_nml
   select_localization = 1  /

&reg_factor_nml
   select_regression    = 1,
   input_reg_file       = "time_mean_reg",
   save_reg_diagnostics = .false.,
   reg_diagnostics_file = "reg_diagnostics"  /

&obs_sequence_nml
   write_binary_obs_sequence = .false.  /
```

(continues on next page)

(continued from previous page)

```
&obs_kind_nml
   assimilate_these_obs_types = 'RAW_STATE_VARIABLE'  /

&assim_model_nml
   write_binary_restart_files = .true. /

&model_nml
   sigma  = 10.0,
   r      = 28.0,
   b      = 2.6666666666667,
   deltat = 0.01,
   time_step_days = 0,
   time_step_seconds = 3600  /

&utilities_nml
   TERMLEVEL = 1,
   logfilename = 'dart_log.out'  /
```

For the moment, only two namelists warrant explanation. Each namelists is covered in detail in the html files accompanying the source code for the module.

perfect_model_obs_nml

namelist variable	description
start_from_restart	When set to ‘false’, perfect_model_obs generates an arbitrary initial condition (which cannot be guaranteed to be on the L63 attractor). When set to ‘true’, a restart file (specified by restart_in_file_name) is read.
output_restart	When set to ‘true’, perfect_model_obs will record the model state at the end of this integration in the file named by restart_out_file_name.
async	The lorenz_63 model is advanced through a subroutine call - indicated by async = 0. There is no other valid value for this model.
init_time_*xxx*	the start time of the integration.
first_obs_*xxx*	the time of the first observation of interest. While not needed in this example, you can skip observations if you want to. A value of -1 indicates to start at the beginning.
last_obs_*xxx*	the time of the last observation of interest. While not needed in this example, you do not have to assimilate all the way to the end of the observation sequence file. A value of -1 indicates to use all the observations.
output_interval	interval at which to save the model state (in True_State.nc).
restart_in_file_name	is ignored when ‘start_from_restart’ is ‘false’.
restart_out_file_name	if output_restart is ‘true’, this specifies the name of the file containing the model state at the end of the integration.
obs_seq_in_file_name	specifies the file name that results from running create_fixed_network_seq, i.e. the ‘observation sequence definition’ file.
obs_seq_out_file_name	specifies the output file name containing the ‘observation sequence’, finally populated with (perfect?) ‘observations’.
advance_ens_command	specifies the shell commands or script to execute when async /= 0.

utilities_nml

namelist variable	description
TERMLEVEL	When set to '1' the programs terminate when a 'warning' is generated. When set to '2' the programs terminate only with 'fatal' errors.
logfilename	Run-time diagnostics are saved to this file. This namelist is used by all programs, so the file is opened in APPEND mode. Subsequent executions cause this file to grow.

Executing perfect_model_obs will integrate the model 24,000 steps and output the resulting state in the file perfect_restart. Interested parties can check the spinup in the True_State.nc file.

```
$ perfect_model_obs
```

2. Generate a set of ensemble initial conditions

The set of initial conditions for a 'perfect model' experiment is created in several steps. 1) Starting from the spun-up state of the model (available in perfect_restart), run perfect_model_obs to generate the 'true state' of the experiment and a corresponding set of observations. 2) Feed the same initial spun-up state and resulting observations into filter.

The first step is achieved by changing a perfect_model_obs namelist parameter, copying perfect_restart to perfect_ics, and rerunning perfect_model_obs. This execution of perfect_model_obs will advance the model state from the end of the first 24,000 steps to the end of an additional 24,000 steps and place the final state in perfect_restart. The rest of the namelists in input.nml should remain unchanged.

```
&perfect_model_obs_nml
   start_from_restart    = .true.,
   output_restart        = .true.,
   async                 = 0,
   init_time_days        = 0,
   init_time_seconds     = 0,
   first_obs_days        = -1,
   first_obs_seconds     = -1,
   last_obs_days         = -1,
   last_obs_seconds      = -1,
   output_interval       = 1,
   restart_in_file_name  = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   obs_seq_in_file_name  = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   adv_ens_command       = "./advance_ens.csh"   /
```

```
$ cp perfect_restart perfect_ics
$ perfect_model_obs
```

A True_State.nc file is also created. It contains the 'true' state of the integration.

Generating the ensemble

This step (#2 from above) is done with the program `filter`, which also uses the Fortran90 namelist mechanism for input. It is now necessary to copy the `input.nml.filter_default` namelist to `input.nml`.

```
$ cp input.nml.filter_default
$ input.nml
```

You may also build one master namelist containting all the required namelists. Having unused namelists in the `input.nml` does not hurt anything, and it has been so useful to be reminded of what is possible that we made it an error to NOT have a required namelist. Take a peek at any of the other models for examples of a "fully qualified" `input.nml`.

HINT: if you used `svn` to get the project, try 'svn revert input.nml' to restore the namelist that was distributed with the project - which DOES have all the namelist blocks. Just be sure the values match the examples here.

```
&filter_nml
   async                    = 0,
   adv_ens_command          = "./advance_model.csh",
   ens_size                 = 100,
   start_from_restart       = .false.,
   output_restart           = .true.,
   obs_sequence_in_name     = "obs_seq.out",
   obs_sequence_out_name    = "obs_seq.final",
   restart_in_file_name     = "perfect_ics",
   restart_out_file_name    = "filter_restart",
   init_time_days           = 0,
   init_time_seconds        = 0,
   first_obs_days           = -1,
   first_obs_seconds        = -1,
   last_obs_days            = -1,
   last_obs_seconds         = -1,
   num_output_state_members = 20,
   num_output_obs_members   = 20,
   output_interval          = 1,
   num_groups               = 1,
   input_qc_threshold       =  4.0,
   outlier_threshold        = -1.0,
   output_forward_op_errors = .false.,
   output_timestamps        = .false.,
   output_inflation         = .true.,

   inf_flavor               = 0,                       0,
   inf_start_from_restart   = .false.,                 .false.,
   inf_output_restart       = .false.,                 .false.,
   inf_deterministic        = .true.,                  .true.,
   inf_in_file_name         = 'not_initialized',       'not_initialized',
   inf_out_file_name        = 'not_initialized',       'not_initialized',
   inf_diag_file_name       = 'not_initialized',       'not_initialized',
   inf_initial              = 1.0,                     1.0,
   inf_sd_initial           = 0.0,                     0.0,
   inf_lower_bound          = 1.0,                     1.0,
   inf_upper_bound          = 1000000.0,               1000000.0,
   inf_sd_lower_bound       = 0.0,                     0.0
/

&smoother_nml
   num_lags              = 0,
```

(continues on next page)

(continued from previous page)

```
   start_from_restart    = .false.,
   output_restart        = .false.,
   restart_in_file_name  = 'smoother_ics',
   restart_out_file_name = 'smoother_restart'  /

&ensemble_manager_nml
   single_restart_file_in  = .true.,
   single_restart_file_out = .true.,
   perturbation_amplitude  = 0.2  /

&assim_tools_nml
   filter_kind                     = 1,
   cutoff                          = 0.2,
   sort_obs_inc                    = .false.,
   spread_restoration              = .false.,
   sampling_error_correction       = .false.,
   adaptive_localization_threshold = -1,
   print_every_nth_obs             = 0  /

&cov_cutoff_nml
   select_localization = 1  /

&reg_factor_nml
   select_regression    = 1,
   input_reg_file       = "time_mean_reg",
   save_reg_diagnostics = .false.,
   reg_diagnostics_file = "reg_diagnostics"  /

&obs_sequence_nml
   write_binary_obs_sequence = .false.  /

&obs_kind_nml
   assimilate_these_obs_types = 'RAW_STATE_VARIABLE'  /

&assim_model_nml
   write_binary_restart_files = .true. /

&model_nml
   sigma  = 10.0,
   r      = 28.0,
   b      = 2.6666666666667,
   deltat = 0.01,
   time_step_days = 0,
   time_step_seconds = 3600  /

&utilities_nml
   TERMLEVEL = 1,
   logfilename = 'dart_log.out'  /
```

Only the non-obvious(?) entries for `filter_nml` will be discussed.

namelist variable	description
ens_size	Number of ensemble members. 100 is sufficient for most of the L63 exercises.
start_from_restart	when '.false.', filter will generate its own ensemble of initial conditions. It is important to note that the filter still makes use of the file named by restart_in_file_name (i.e. perfect_ics) by randomly perturbing these state variables.
num_output_state_members	specifies the number of state vectors contained in the netCDF diagnostic files. May be a value from 0 to ens_size.
num_output_obs_members	specifies the number of 'observations' (derived from applying the forward operator to the state vector) are contained in the obs_seq.final file. May be a value from 0 to ens_size
inf_flavor	A value of 0 results in no inflation.(spin-up)

The filter is told to generate its own ensemble initial conditions since start_from_restart is '.false.'. However, it is important to note that the filter still makes use of perfect_ics which is set to be the restart_in_file_name. This is the model state generated from the first 24,000 step model integration by perfect_model_obs. Filter generates its ensemble initial conditions by randomly perturbing the state variables of this state.

num_output_state_members are '.true.' so the state vector is output at every time for which there are observations (once a day here). Posterior_Diag.nc and Prior_Diag.nc then contain values for 20 ensemble members once a day. Once the namelist is set, execute filter to integrate the ensemble forward for 24,000 steps with the final ensemble state written to the filter_restart. Copy the perfect_model_obs restart file perfect_restart (the `true state') to perfect_ics, and the filter restart file filter_restart to filter_ics so that future assimilation experiments can be initialized from these spun-up states.

```
$ filter
$ cp perfect_restart perfect_ics
$ cp filter_restart filter_ics
```

The spin-up of the ensemble can be viewed by examining the output in the netCDF files True_State.nc generated by perfect_model_obs and Posterior_Diag.nc and Prior_Diag.nc generated by filter. To do this, see the detailed discussion of matlab diagnostics in Appendix I.

3. Simulate a particular observing system

Begin by using create_obs_sequence to generate an observation set in which each of the 3 state variables of L63 is observed with an observational error variance of 1.0 for each observation. To do this, use the following input sequence (the text including and after # is a comment and does not need to be entered):

4	# upper bound on num of observations in sequence
0	# number of copies of data (0 for just a definition)
0	# number of quality control values per field (0 or greater)
0	# -1 to exit/end observation definitions
-1	# observe state variable 1
0 0	# time – days, seconds
1.0	# observational variance
0	# -1 to exit/end observation definitions
-2	# observe state variable 2
0 0	# time – days, seconds
1.0	# observational variance
0	# -1 to exit/end observation definitions
-3	# observe state variable 3
0 0	# time – days, seconds
1.0	# observational variance
-1	# -1 to exit/end observation definitions
set_def.out	# Output file name

Now, generate an observation sequence definition by running `create_fixed_network_seq` with the following input sequence:

set_def.out	# Input observation set definition file
1	# Regular spaced observation interval in time
1000	# 1000 observation times
0, 43200	# First observation after 12 hours (0 days, 12 * 3600 seconds)
0, 43200	# Observations every 12 hours
obs_seq.in	# Output file for observation sequence definition

4. Generate a particular observing system and true state

An observation sequence file is now generated by running `perfect_model_obs` with the namelist values (unchanged from step 2):

```
&perfect_model_obs_nml
   start_from_restart    = .true.,
   output_restart        = .true.,
   async                 = 0,
   init_time_days        = 0,
   init_time_seconds     = 0,
   first_obs_days        = -1,
   first_obs_seconds     = -1,
   last_obs_days         = -1,
   last_obs_seconds      = -1,
   output_interval       = 1,
   restart_in_file_name  = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   obs_seq_in_file_name  = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   adv_ens_command       = "./advance_ens.csh"   /
```

This integrates the model starting from the state in `perfect_ics` for 1000 12-hour intervals outputting synthetic observations of the three state variables every 12 hours and producing a netCDF diagnostic file, `True_State.nc`.

5. Filtering

Finally, `filter` can be run with its namelist set to:

```
&filter_nml
   async                    = 0,
   adv_ens_command          = "./advance_model.csh",
   ens_size                 = 100,
   start_from_restart       = .true.,
   output_restart           = .true.,
   obs_sequence_in_name     = "obs_seq.out",
   obs_sequence_out_name    = "obs_seq.final",
   restart_in_file_name     = "filter_ics",
   restart_out_file_name    = "filter_restart",
   init_time_days           = 0,
   init_time_seconds        = 0,
   first_obs_days           = -1,
   first_obs_seconds        = -1,
   last_obs_days            = -1,
   last_obs_seconds         = -1,
   num_output_state_members = 20,
   num_output_obs_members   = 20,
   output_interval          = 1,
   num_groups               = 1,
   input_qc_threshold       =  4.0,
   outlier_threshold        = -1.0,
   output_forward_op_errors = .false.,
   output_timestamps        = .false.,
   output_inflation         = .true.,

   inf_flavor               = 0,                       0,
   inf_start_from_restart   = .false.,                 .false.,
   inf_output_restart       = .false.,                 .false.,
   inf_deterministic        = .true.,                  .true.,
   inf_in_file_name         = 'not_initialized',       'not_initialized',
   inf_out_file_name        = 'not_initialized',       'not_initialized',
   inf_diag_file_name       = 'not_initialized',       'not_initialized',
   inf_initial              = 1.0,                     1.0,
   inf_sd_initial           = 0.0,                     0.0,
   inf_lower_bound          = 1.0,                     1.0,
   inf_upper_bound          = 1000000.0,               1000000.0,
   inf_sd_lower_bound       = 0.0,                     0.0
 /
```

`filter` produces two output diagnostic files, `Prior_Diag.nc` which contains values of the ensemble mean, ensemble spread, and ensemble members for 12- hour lead forecasts before assimilation is applied and `Posterior_Diag.nc` which contains similar data for after the assimilation is applied (sometimes referred to as analysis values).

Now try applying all of the matlab diagnostic functions described in the Matlab® Diagnostics section.

6.16.13 The tutorial

The DART/tutorial documents are an excellent way to kick the tires on DART and learn about ensemble data assimilation. If you have gotten this far, you can run anything in the tutorial.

6.16.14 Matlab® diagnostics

The output files are netCDF files, and may be examined with many different software packages. We happen to use Matlab®, and provide our diagnostic scripts in the hopes that they are useful.

The diagnostic scripts and underlying functions reside in two places: DART/diagnostics/matlab and DART/matlab. They are reliant on the public-domain netcdf toolbox from http://woodshole.er.usgs.gov/staffpages/cdenham/public_html/MexCDF/nc4ml5.html as well as the public-domain CSIRO matlab/netCDF interface from http://www.marine.csiro.au/sw/matlab-netcdf.html. If you do not have them installed on your system and want to use Matlab to peruse netCDF, you must follow their installation instructions. The 'interested reader' may want to look at the DART/matlab/startup.m file I use on my system. If you put it in your $HOME/matlab directory, it is invoked every time you start up Matlab.

Once you can access the getnc function from within Matlab, you can use our diagnostic scripts. It is necessary to prepend the location of the DART/matlab scripts to the matlabpath. Keep in mind the location of the netcdf operators on your system WILL be different from ours ... and that's OK.

```
0[269]0 ghotiol:/<5>models/lorenz_63/work]$ matlab -nojvm

                                             < M A T L A B >
                                 Copyright 1984-2002 The MathWorks, Inc.
                                     Version 6.5.0.180913a Release 13
                                               Jun 18 2002

  Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.

  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

>> which getnc
/contrib/matlab/matlab_netcdf_5_0/getnc.m
>>ls *.nc

ans =

Posterior_Diag.nc  Prior_Diag.nc  True_State.nc


>>path('../../../matlab',path)
>>path('../../../diagnostics/matlab',path)
>>which plot_ens_err_spread
../../../matlab/plot_ens_err_spread.m
>>help plot_ens_err_spread

  DART : Plots summary plots of the ensemble error and ensemble spread.
                         Interactively queries for the needed information.
                         Since different models potentially need different
                         pieces of information ... the model types are
```

(continues on next page)

(continued from previous page)

```
                         determined and additional user input may be queried.

  Ultimately, plot_ens_err_spread will be replaced by a GUI.
  All the heavy lifting is done by PlotEnsErrSpread.

  Example 1 (for low-order models)

  truth_file = 'True_State.nc';
  diagn_file = 'Prior_Diag.nc';
  plot_ens_err_spread

>>plot_ens_err_spread
```

And the matlab graphics window will display the spread of the ensemble error for each state variable. The scripts are designed to do the "obvious" thing for the low-order models and will prompt for additional information if needed. The philosophy of these is that anything that starts with a lower-case *plot_some_specific_task* is intended to be user-callable and should handle any of the models. All the other routines in `DART/matlab` are called BY the high-level routines.

Matlab script	description
`plot_bins`	plots ensemble rank histograms
`plot_correl`	Plots space-time series of correlation between a given variable at a given time and other variables at all times in a n ensemble time sequence.
`plot_ens_err_spread`	Plots summary plots of the ensemble error and ensemble spread. Interactively queries for the needed information. Since different models potentially need different pieces of information . . . the model types are determined and additional user input may be queried.
`plot_ens_mean_time_series`	Queries for the state variables to plot.
`plot_ens_time_series`	Queries for the state variables to plot.
`plot_phase_space`	Plots a 3D trajectory of (3 state variables of) a single ensemble member. Additional trajectories may be superimposed.
`plot_total_err`	Summary plots of global error and spread.
`plot_var_var_correl`	Plots time series of correlation between a given variable at a given time and another variable at all times in an ensemble time sequence.

6.16.15 Bias, filter divergence and covariance inflation (with the l63 model)

One of the common problems with ensemble filters is filter divergence, which can also be an issue with a variety of other flavors of filters including the classical Kalman filter. In filter divergence, the prior estimate of the model state becomes too confident, either by chance or because of errors in the forecast model, the observational error characteristics, or approximations in the filter itself. If the filter is inappropriately confident that its prior estimate is correct, it will then tend to give less weight to observations than they should be given. The result can be enhanced overconfidence in the model's state estimate. In severe cases, this can spiral out of control and the ensemble can wander entirely away from the truth, confident that it is correct in its estimate. In less severe cases, the ensemble estimates may not diverge entirely from the truth but may still be too confident in their estimate. The result is that the truth ends up being farther away from the filter estimates than the spread of the filter ensemble would estimate. This type of behavior is commonly detected using rank histograms (also known as Talagrand diagrams). You can see the rank histograms for the L63 initial assimilation by using the matlab script `plot_bins`.

A simple, but surprisingly effective way of dealing with filter divergence is known as covariance inflation. In this method, the prior ensemble estimate of the state is expanded around its mean by a constant factor, effectively increasing the prior estimate of uncertainty while leaving the prior mean estimate unchanged. The program `filter` has a namelist parameter that controls the application of covariance inflation, `cov_inflate`. Up to this point, `cov_inflate` has been set to 1.0 indicating that the prior ensemble is left unchanged. Increasing `cov_inflate` to values greater than 1.0 inflates the ensemble before assimilating observations at each time they are available. Values smaller than 1.0 contract (reduce the spread) of prior ensembles before assimilating.

You can do this by modifying the value of `cov_inflate` in the namelist, (try 1.05 and 1.10 and other values at your discretion) and run the filter as above. In each case, use the diagnostic matlab tools to examine the resulting changes to the error, the ensemble spread (via rank histogram bins, too), etc. What kind of relation between spread and error is seen in this model?

6.16.16 Synthetic observations

Synthetic observations are generated from a `perfect' model integration, which is often referred to as the `truth' or a `nature run'. A model is integrated forward from some set of initial conditions and observations are generated as $y = H(x) + e$ where H is an operator on the model state vector, x, that gives the expected value of a set of observations, y, and e is a random variable with a distribution describing the error characteristics of the observing instrument(s) being simulated. Using synthetic observations in this way allows students to learn about assimilation algorithms while being isolated from the additional (extreme) complexity associated with model error and unknown observational error characteristics. In other words, for the real-world assimilation problem, the model has (often substantial) differences from what happens in the real system and the observational error distribution may be very complicated and is certainly not well known. Be careful to keep these issues in mind while exploring the capabilities of the ensemble filters with synthetic observations.

6.17 Iceland

6.17.1 DART Iceland release documentation

Attention: Iceland is a prior release of DART. Its source code is available via the DART repository on Github. This documentation is preserved merely for reference. See the DART homepage to learn about the latest release.

6.17.2 Overview of DART

The Data Assimilation Research Testbed (DART) is designed to facilitate the combination of assimilation algorithms, models, and observation sets to allow increased understanding of all three. The DART programs have been compiled with several Fortran 90 compilers and run on a linux compute-server and linux clusters. You should definitely read the Customizations section.

DART employs a modular programming approach to apply an Ensemble Kalman Filter which nudges models toward a state that is more consistent with information from a set of observations. Models may be swapped in and out, as can different algorithms in the Ensemble Kalman Filter. The method requires running multiple instances of a model to generate an ensemble of states. A forward operator appropriate for the type of observation being used is applied to each of the states to generate the model's estimate of the observation. Comparing these estimates and their uncertainty to the observation and its uncertainty ultimately results in the adjustments to the model states. Sort of. There's more to it, described in detail in the tutorial directory of the package.

DART ultimately creates a few netCDF files containing the model states just before the adjustment `Prior_Diag.nc` and just after the adjustment `Posterior_Diag.nc` as well as a file `obs_seq.final` with the model estimates of the observations. There is a suite of Matlab® functions that facilitate exploration of the results.

The Iceland release provides several new models and has a greatly expanded capability for **real** observations which required a fundamentally different implementation of the low-level routines. It is now required to run a preprocessor on several of the program units to construct the source code files which will be compiled by the remaining units. Due to the potentially large number of observations types possible and for portability reasons, the preprocessor is actually a F90 program that uses the namelist mechanism for specifying the observation types to be included. This also prevents having a gory set of compile flags that is different for every compiler. One very clever colleague also 'built a better mousetrap' and figured out how to effectively and robustly read namelists, detect errors, and generate meaningful error messages. HURRAY!

The Iceland release has also been tested with more compilers in an attempt to determine non-portable code elements. It is my experience that the largest impediment to portable code is the reliance on the compiler to autopromote `real` variables to one flavor or another. Using the F90 "kind" allows for much more flexible code, in that the use of interface procedures is possible only when two routines do not have identical sets of input arguments – something that happens when the compiler autopromotes 32bit reals to 64bit reals, for example.

DART programs can require three different types of input. First, some of the DART programs, those for creating synthetic observational datasets, require interactive input from the keyboard. For simple cases, this interactive input can be made directly from the keyboard. In more complicated cases, a file containing the appropriate keyboard input can be created and this file can be directed to the standard input of the DART program. Second, many DART programs expect one or more input files in DART specific formats to be available. For instance, `perfect_model_obs`, which creates a synthetic observation set given a particular model and a description of a sequence of observations, requires an input file that describes this observation sequence. At present, the observation files for DART are in a custom format in either human-readable ascii or more compact machine-specific binary. Third, many DART modules (including main programs) make use of the Fortan90 namelist facility to obtain values of certain parameters at run-time. All programs look for a namelist input file called `input.nml` in the directory in which the program is executed. The `input.nml` file can contain a sequence of individual Fortran90 namelists which specify values of particular parameters for modules that compose the executable program. DART provides a mechanism that automatically generates namelists with the default values for each program to be run.

DART uses the netCDF self-describing data format with a particular metadata convention to describe output that is used to analyze the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. A set of Matlab scripts, designed to produce graphical diagnostics from DART netCDF output files are available. DART users have also used ncview to create rudimentary graphical displays of output data fields. The NCO tools, produced by UCAR's Unidata group, are available to do operations like concatenating, slicing, and dicing of netCDF files.

6.17.3 Requirements: an F90 compiler

The DART software has been successfully built on several Linux/x86 platforms with several versions of the Intel Fortran Compiler for Linux, which (at one point) is/was free for individual scientific use. It has also been built and successfully run with several versions of each of the following: Portland Group Fortran Compiler, Lahey Fortran Compiler, Pathscale Fortran Compiler, Absoft Fortran 90/95 Compiler (Mac OSX). Since recompiling the code is a necessity to experiment with different models, there are no binaries to distribute.

DART uses the netCDF self-describing data format for the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. In particular, DART also makes use of the F90 interface to the library which is available through the `netcdf.mod` and `typesizes.mod` modules. *IMPORTANT*: different compilers create these modules with different "case" filenames, and sometimes they are not **both** installed into the expected directory. It is required that both modules be present. The normal place would be in the `netcdf/include` directory, as opposed to the `netcdf/lib` directory.

If the netCDF library does not exist on your system, you must build it (as well as the F90 interface modules). The library and instructions for building the library or installing from an RPM may be found at the netCDF home page: http://www.unidata.ucar.edu/packages/netcdf/ Pay particular attention to the compiler-specific patches that must be applied for the Intel Fortran Compiler. (Or the PG compiler, for that matter.)

The location of the netCDF library, `libnetcdf.a`, and the locations of both `netcdf.mod` and `typesizes.mod` will be needed by the makefile template, as described in the compiling section.

6.17.4 Unpacking the distribution

This release of the DART source code can be downloaded as a compressed zip or tar.gz file. When extracted, the source tree will begin with a directory named `DART` and will be approximately 103.1 Mb. Compiling the code in this tree (as is usually the case) will necessitate much more space.

```
$ gunzip DART-5.0.0.tar.gz
$ tar -xvf DART-5.0.0.tar
```

The code tree is very "bushy"; there are many directories of support routines, etc. but only a few directories involved with the customization and installation of the DART software. If you can compile and run ONE of the low-order models, you should be able to compile and run ANY of the low-order models. For this reason, we can focus on the Lorenz `63 model. Subsequently, the only directories with files to be modified to check the installation are: `DART/mkmf`, `DART/models/lorenz_63/work`, and `DART/matlab` (but only for analysis).

6.17.5 Customizing the build scripts – overview

DART executable programs are constructed using two tools: `make` and `mkmf`. The `make` utility is a relatively common piece of software that requires a user-defined input file that records dependencies between different source files. `make` then performs a hierarchy of actions when one or more of the source files is modified. The `mkmf` utility is a custom preprocessor that generates a `make` input file (named `Makefile`) and an example namelist *input.nml.program_default* with the default values. The `Makefile` is designed specifically to work with object-oriented Fortran90 (and other languages) for systems like DART.

`mkmf` requires two separate input files. The first is a `template' file which specifies details of the commands required for a specific Fortran90 compiler and may also contain pointers to directories containing pre-compiled utilities required by the DART system. **This template file will need to be modified to reflect your system**. The second input file is a `path_names' file which includes a complete list of the locations (either relative or absolute) of all Fortran90 source files that are required to produce a particular DART program. Each 'path_names' file must contain a path for exactly one Fortran90 file containing a main program, but may contain any number of additional paths pointing to files containing Fortran90 modules. An `mkmf` command is executed which uses the 'path_names' file and the mkmf template file to produce a `Makefile` which is subsequently used by the standard `make` utility.

Shell scripts that execute the mkmf command for all standard DART executables are provided as part of the standard DART software. For more information on `mkmf` see the FMS mkmf description.

One of the benefits of using `mkmf` is that it also creates an example namelist file for each program. The example namelist is called *input.nml.program_default*, so as not to clash with any exising `input.nml` that may exist in that directory.

Building and customizing the 'mkmf.template' file

A series of templates for different compilers/architectures exists in the `DART/mkmf/` directory and have names with extensions that identify either the compiler, the architecture, or both. This is how you inform the build process of the specifics of your system. Our intent is that you copy one that is similar to your system into `mkmf.template` and customize it. For the discussion that follows, knowledge of the contents of one of these templates (i.e. `mkmf.template.pgf90.ghotiol`) is needed: (note that only the LAST lines are shown here, the head of the file is just a big comment)

```
# Makefile template for PGI f90
FC = pgf90
LD = pgf90
CPPFLAGS =
LIST = -Mlist
```

(continues on next page)

(continued from previous page)

```
NETCDF = /contrib/netcdf-3.5.1-cc-c++-pgif90.5.2-4
FFLAGS = -O0 -Ktrap=fp -pc 64 -I$(NETCDF)/include
LIBS = -L$(NETCDF)/lib -lnetcdf
LDFLAGS = $(LIBS)

...
```

Essentially, each of the lines defines some part of the resulting `Makefile`. Since `make` is particularly good at sorting out dependencies, the order of these lines really doesn't make any difference. The `FC = pgf90` line ultimately defines the Fortran90 compiler to use, etc. The lines which are most likely to need site-specific changes start with `FFLAGS` and `NETCDF`, which indicate where to look for the netCDF F90 modules and the location of the netCDF library and modules.

Netcdf

Modifying the `NETCDF` value should be relatively straightforward.
Change the string to reflect the location of your netCDF installation containing `netcdf.mod` and `typesizes.mod`. The value of the `NETCDF` variable will be used by the `FFLAGS, LIBS,` and `LDFLAGS` variables.

FFLAGS

Each compiler has different compile flags, so there is really no way to exhaustively cover this other than to say the templates as we supply them should work – depending on the location of your netCDF. The low-order models can be compiled without a `-r8` switch, but the `bgrid_solo` model cannot.

Customizing the 'path_names_*' file

Several `path_names_*` files are provided in the `work` directory for each specific model, in this case: `DART/models/lorenz_63/work`.

1. `path_names_preprocess`
2. `path_names_create_obs_sequence`
3. `path_names_create_fixed_network_seq`
4. `path_names_perfect_model_obs`
5. `path_names_filter`
6. `path_names_obs_diag`

Since each model comes with its own set of files, no further customization is needed.

6.17.6 Building the Lorenz_63 DART project

Currently, DART executables are constructed in a work subdirectory under the directory containing code for the given model. In the top-level DART directory, change to the L63 work directory and list the contents:

```
$ cd DART/models/lorenz_63/work
$ ls -1
```

With the result:

```
filter_ics
filter_restart
input.nml
mkmf_create_fixed_network_seq
mkmf_create_obs_sequence
mkmf_filter
mkmf_obs_diag
mkmf_perfect_model_obs
mkmf_preprocess
obs_seq.final
obs_seq.in
obs_seq.out
obs_seq.out.average
obs_seq.out.x
obs_seq.out.xy
obs_seq.out.xyz
obs_seq.out.z
path_names_create_fixed_network_seq
path_names_create_obs_sequence
path_names_filter
path_names_obs_diag
path_names_perfect_model_obs
path_names_preprocess
perfect_ics
perfect_restart
Posterior_Diag.nc
Prior_Diag.nc
set_def.out
True_State.nc
workshop_setup.csh
```

There are six mkmf_*xxxxxx* files for the programs preprocess, create_obs_sequence, create_fixed_network_seq, perfect_model_obs, filter, and obs_diag along with the corresponding path_names_*xxxxxx* files. You can examine the contents of one of the path_names_*xxxxxx* files, for instance path_names_filter, to see a list of the relative paths of all files that contain Fortran90 modules required for the program filter for the L63 model. All of these paths are relative to your DART directory. The first path is the main program (filter.f90) and is followed by all the Fortran90 modules used by this program (after preprocessing).

The mkmf_*xxxxxx* scripts are cryptic but should not need to be modified – as long as you do not restructure the code tree (by moving directories, for example). The only function of the mkmf_*xxxxxx* script is to generate a Makefile and an *input.nml.program_default* file. It is not supposed to compile anything:

```
$ csh mkmf_preprocess
$ make
```

The first command generates an appropriate `Makefile` and the `input.nml.preprocess_default` file. The second command results in the compilation of a series of Fortran90 modules which ultimately produces an executable file: `preprocess`. Should you need to make any changes to the `DART/mkmf/mkmf.template`, you will need to regenerate the `Makefile`.

The `preprocess` program actually builds source code to be used by all the remaining modules. It is **imperative** to actually **run** `preprocess` before building the remaining executables. This is how the same code can assimilate state vector 'observations' for the Lorenz_63 model and real radar reflectivities for WRF without needing to specify a set of radar operators for the Lorenz_63 model!

`preprocess` reads the `&preprocess_nml` namelist to determine what observations and operators to incorporate. For this exercise, we will use the values in `input.nml`. `preprocess` is designed to abort if the files it is supposed to build already exist. For this reason, it is necessary to remove a couple files (if they exist) before you run the preprocessor. It is just a good habit to develop.

```
$ \rm -f ../../../obs_def/obs_def_mod.f90
$ \rm -f ../../../obs_kind/obs_kind_mod.f90
$ ./preprocess
$ ls -l ../../../obs_def/obs_def_mod.f90
$ ls -l ../../../obs_kind/obs_kind_mod.f90
```

This created `../../../obs_def/obs_def_mod.f90` from `../../../obs_kind/DEFAULT_obs_kind_mod.F90` and several other modules. `../../../obs_kind/obs_kind_mod.f90` was created similarly. Now we can build the rest of the project.

A series of object files for each module compiled will also be left in the work directory, as some of these are undoubtedly needed by the build of the other DART components. You can proceed to create the other five programs needed to work with L63 in DART as follows:

```
$ csh mkmf_create_obs_sequence
$ make
$ csh mkmf_create_fixed_network_seq
$ make
$ csh mkmf_perfect_model_obs
$ make
$ csh mkmf_filter
$ make
$ csh mkmf_obs_diag
$ make
```

The result (hopefully) is that six executables now reside in your work directory. The most common problem is that the netCDF libraries and include files (particularly `typesizes.mod`) are not found. Edit the `DART/mkmf/mkmf.template`, recreate the `Makefile`, and try again.

program	purpose
`preprocess`	creates custom source code for just the observations of interest
`create_obs_sequence`	specify a (set) of observation characteristics taken by a particular (set of) instruments
`create_fixed_network_seq`	specify the temporal attributes of the observation sets
`perfect_model_obs`	spinup, generate "true state" for synthetic observation experiments, ...
`filter`	perform experiments
`obs_diag`	creates observation-space diagnostic files to be explored by the Matlab® scripts.

6.17.7 Running Lorenz_63

This initial sequence of exercises includes detailed instructions on how to work with the DART code and allows investigation of the basic features of one of the most famous dynamical systems, the 3-variable Lorenz-63 model. The remarkable complexity of this simple model will also be used as a case study to introduce a number of features of a simple ensemble filter data assimilation system. To perform a synthetic observation assimilation experiment for the L63 model, the following steps must be performed (an overview of the process is given first, followed by detailed procedures for each step):

6.17.8 Experiment overview

1. Integrate the L63 model for a long time starting from arbitrary initial conditions to generate a model state that lies on the attractor. The ergodic nature of the L63 system means a 'lengthy' integration always converges to some point on the computer's finite precision representation of the model's attractor.
2. Generate a set of ensemble initial conditions from which to start an assimilation. Since L63 is ergodic, the ensemble members can be designed to look like random samples from the model's 'climatological distribution'. To generate an ensemble member, very small perturbations can be introduced to the state on the attractor generated by step 1. This perturbed state can then be integrated for a very long time until all memory of its initial condition can be viewed as forgotten. Any number of ensemble initial conditions can be generated by repeating this procedure.
3. Simulate a particular observing system by first creating an 'observation set definition' and then creating an 'observation sequence'. The 'observation set definition' describes the instrumental characteristics of the observations and the 'observation sequence' defines the temporal sequence of the observations.
4. Populate the 'observation sequence' with 'perfect' observations by integrating the model and using the information in the 'observation sequence' file to create simulated observations. This entails operating on the model state at the time of the observation with an appropriate forward operator (a function that operates on the model state vector to produce the expected value of the particular observation) and then adding a random sample from the observation error distribution specified in the observation set definition. At the same time, diagnostic output about the 'true' state trajectory can be created.
5. Assimilate the synthetic observations by running the filter; diagnostic output is generated.

1. Integrate the L63 model for a 'long' time

`perfect_model_obs` integrates the model for all the times specified in the 'observation sequence definition' file. To this end, begin by creating an 'observation sequence definition' file that spans a long time. Creating an 'observation sequence definition' file is a two-step procedure involving `create_obs_sequence` followed by `create_fixed_network_seq`. After they are both run, it is necessary to integrate the model with `perfect_model_obs`.

1.1 Create an observation set definition

`create_obs_sequence` creates an observation set definition, the time-independent part of an observation sequence. An observation set definition file only contains the `location, type`, and `observational error characteristics` (normally just the diagonal observational error variance) for a related set of observations. There are no actual observations, nor are there any times associated with the definition. For spin-up, we are only interested in integrating the L63 model, not in generating any particular synthetic observations. Begin by creating a minimal observation set definition.

In general, for the low-order models, only a single observation set need be defined. Next, the number of individual scalar observations (like a single surface pressure observation) in the set is needed. To spin-up an initial condition for the L63 model, only a single observation is needed. Next, the error variance for this observation must be entered. Since we do not need (nor want) this observation to have any impact on an assimilation (it will only be used for spinning up the model and the ensemble), enter a very large value for the error variance. An observation with a very large error variance has essentially no impact on deterministic filter assimilations like the default variety implemented in DART. Finally, the location and type of the observation need to be defined. For all types of models, the most elementary form of synthetic observations are called 'identity' observations. These observations are generated simply by adding a random sample from a specified observational error distribution directly to the value of one of the state variables. This defines the observation as being an identity observation of the first state variable in the L63 model. The program will respond by terminating after generating a file (generally named `set_def.out`) that defines the single identity observation of the first state variable of the L63 model. The following is a screenshot (much of the verbose logging has been left off for clarity), the user input looks *like this*.

```
[unixprompt]$ ./create_obs_sequence
 Initializing the utilities module.
 Trying to read from unit           10
 Trying to open file dart_log.out

 Registering module :
 $source: /home/dart/CVS.REPOS/DART/utilities/utilities_mod.f90,v $
 $revision: 1.18 $
 $date: 2004/06/29 15:16:40 $
 Registration complete.

 &UTILITIES_NML
 TERMLEVEL= 2,LOGFILENAME=dart_log.out

 /

 Registering module :
 $source: /home/dart/CVS.REPOS/DART/obs_sequence/create_obs_sequence.f90,v $
 $revision: 1.18 $
 $date: 2004/05/24 15:41:46 $
 Registration complete.

 { ... }

 Input upper bound on number of observations in sequence
10

 Input number of copies of data (0 for just a definition)
0

 Input number of quality control values per field (0 or greater)
0

 input a -1 if there are no more obs
0
```

```
initialize_module obs_kind_nml values are

-------------- ASSIMILATE_THESE_OBS_TYPES --------------
RAW_STATE_VARIABLE
-------------- EVALUATE_THESE_OBS_TYPES --------------
------------------------------------------------------


    Input -1 * state variable index for identity observations
    OR input the name of the observation kind from table below:
```

(continues on next page)

(continued from previous page)

```
    OR input the integer index, BUT see documentation...
                1 RAW_STATE_VARIABLE
```

```
-1

 input time in days and seconds
1 0

 Input error variance for this observation definition
1000000

 input a -1 if there are no more obs
-1

 Input filename for sequence (  set_def.out   usually works well)
 set_def.out
 write_obs_seq  opening formatted file set_def.out
 write_obs_seq  closed file set_def.out
```

1.2 Create an observation sequence definition

`create_fixed_network_seq` creates an ‘observation sequence definition’ by extending the ‘observation set definition’ with the temporal attributes of the observations.

The first input is the name of the file created in the previous step, i.e. the name of the observation set definition that you’ve just created. It is possible to create sequences in which the observation sets are observed at regular intervals or irregularly in time. Here, all we need is a sequence that takes observations over a long period of time - indicated by entering a 1. Although the L63 system normally is defined as having a non-dimensional time step, the DART system arbitrarily defines the model timestep as being 3600 seconds. If we declare that we have one observation per day for 1000 days, we create an observation sequence definition spanning 24000 ‘model’ timesteps; sufficient to spin-up the model onto the attractor. Finally, enter a name for the ‘observation sequence definition’ file. Note again: there are no observation values present in this file. Just an observation type, location, time and the error characteristics. We are going to populate the observation sequence with the `perfect_model_obs` program.

```
[unixprompt]$ ./create_fixed_network_seq

 ...

 Registering module :
 $source: /home/dart/CVS.REPOS/DART/obs_sequence/obs_sequence_mod.f90,v $
 $revision: 1.31 $
 $date: 2004/06/29 15:04:37 $
 Registration complete.

 Input filename for network definition sequence (usually  set_def.out  )
set_def.out

 ...

 To input a regularly repeating time sequence enter 1
 To enter an irregular list of times enter 2
1
 Input number of observations in sequence
```

(continues on next page)

(continued from previous page)

```
1000
 Input time of initial ob in sequence in days and seconds
1, 0
 Input period of obs in days and seconds
1, 0
           1
           2
           3
...
         997
         998
         999
        1000
What is output file name for sequence (  obs_seq.in   is recommended )
obs_seq.in
 write_obs_seq  opening formatted file obs_seq.in
 write_obs_seq closed file [blah blah blah]/work/obs_seq.in
```

1.3 Initialize the model onto the attractor

`perfect_model_obs` can now advance the arbitrary initial state for 24,000 timesteps to move it onto the attractor.

`perfect_model_obs` uses the Fortran90 namelist input mechanism instead of (admittedly gory, but temporary) interactive input. All of the DART software expects the namelists to found in a file called `input.nml`. When you built the executable, an example namelist was created `input.nml.perfect_model_obs_default` that contains all of the namelist input for the executable. If you followed the example, each namelist was saved to a unique name. We must now rename and edit the namelist file for `perfect_model_obs`. Copy `input.nml.perfect_model_obs_default` to `input.nml` and edit it to look like the following:

```
   &perfect_model_obs_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .false.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1 /

&ensemble_manager_nml
   in_core = .true.,
   single_restart_file_in = .true.,
   single_restart_file_out = .true. /

&assim_tools_nml
   filter_kind = 1,
   cutoff = 0.2,
   sort_obs_inc = .false.,
   cov_inflate = -1.0,
   cov_inflate_sd = 0.05,
   sd_lower_bound = 0.05,
   deterministic_cov_inflate = .true.,
```

(continues on next page)

(continued from previous page)

```
   start_from_assim_restart = .false.,
   assim_restart_in_file_name =
'assim_tools_ics',
   assim_restart_out_file_name =
'assim_tools_restart',
   do_parallel = 0,
   num_domains = 1
   parallel_command = "./assim_filter.csh",
   spread_restoration = .false.,
   cov_inflate_upper_bound = 10000000.0,
   internal_outlier_threshold = -1.0 /

&cov_cutoff_nml
   select_localization = 1 /

&reg_factor_nml
   select_regression = 1,
   input_reg_file = "time_mean_reg"
   save_reg_diagnostics = .false.,
   reg_diagnostics_file = 'reg_diagnostics' /

&obs_sequence_nml
   write_binary_obs_sequence = .false. /

&obs_kind_nml
   assimilate_these_obs_types = 'RAW_STATE_VARIABLE' /

&assim_model_nml
   write_binary_restart_files = .true. /

&model_nml
   sigma = 10.0,
   r = 28.0,
   b = 2.6666666666667,
   deltat = 0.01,
   time_step_days = 0,
   time_step_seconds = 3600 /

&utilities_nml
   TERMLEVEL = 1
   logfilename = 'dart_log.out' /
```

For the moment, only two namelists warrant explanation. Each namelists is covered in detail in the html files accompanying the source code for the module.

perfect_model_obs_nml

namelist variable	description
`async`	For the lorenz_63, simply ignore this. Leave it set to '0'
`advance_ens_command`	specifies the shell commands or script to execute when async /= 0
`obs_seq_in_file_name`	specifies the file name that results from running `create_fixed_network_seq`, i.e. the 'observation sequence definition' file.
`obs_seq_out_file_name`	specifies the output file name containing the 'observation sequence', finally populated with (perfect?) 'observations'.
`start_from_restart`	When set to 'false', `perfect_model_obs` generates an arbitrary initial condition (which cannot be guaranteed to be on the L63 attractor).
`output_restart`	When set to 'true', `perfect_model_obs` will record the model state at the end of this integration in the file named by `restart_out_file_name`.
`restart_in_file_name`	is ignored when 'start_from_restart' is 'false'.
`restart_out_file_name`	if `output_restart` is 'true', this specifies the name of the file containing the model state at the end of the integration.
`init_time_`*xxxx*	the start time of the integration.
`output_interval`	interval at which to save the model state.

utilities_nml

namelist variable	description
`TERMLEVEL`	When set to '1' the programs terminate when a 'warning' is generated. When set to '2' the programs terminate only with 'fatal' errors.
`logfilename`	Run-time diagnostics are saved to this file. This namelist is used by all programs, so the file is opened in APPEND mode. Subsequent executions cause this file to grow.

Executing `perfect_model_obs` will integrate the model 24,000 steps and output the resulting state in the file `perfect_restart`. Interested parties can check the spinup in the `True_State.nc` file.

```
$ perfect_model_obs
```

2. Generate a set of ensemble initial conditions

The set of initial conditions for a 'perfect model' experiment is created in several steps. 1) Starting from the spun-up state of the model (available in `perfect_restart`), run `perfect_model_obs` to generate the 'true state' of the experiment and a corresponding set of observations. 2) Feed the same initial spun-up state and resulting observations into `filter`.

The first step is achieved by changing a perfect_model_obs namelist parameter, copying `perfect_restart` to `perfect_ics`, and rerunning `perfect_model_obs`. This execution of `perfect_model_obs` will advance the model state from the end of the first 24,000 steps to the end of an additional 24,000 steps and place the final state in `perfect_restart`. The rest of the namelists in `input.nml` should remain unchanged.

```
&perfect_model_obs_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   obs_seq_in_file_name = "obs_seq.in",
```

(continues on next page)

(continued from previous page)

```
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .true.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1 /
```

```
$ cp perfect_restart perfect_ics
$ perfect_model_obs
```

A True_State.nc file is also created. It contains the 'true' state of the integration.

Generating the ensemble

This step (#2 from above) is done with the program filter, which also uses the Fortran90 namelist mechanism for input. It is now necessary to copy the input.nml.filter_default namelist to input.nml or you may simply insert the filter_nml namelist block into the existing input.nml. Having the perfect_model_obs namelist in the input.nml does not hurt anything. In fact, I generally create a single input.nml that has all the namelist blocks in it. I simply copied the filter namelist block from input.nml.filter_default and inserted it into our input.nml for the following example.

```
&perfect_model_obs_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .true.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1 /

&filter_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   ens_size = 100,
   cov_inflate = 1.0,
   start_from_restart = .false.,
   output_restart = .true.,
   obs_sequence_in_name = "obs_seq.out",
   obs_sequence_out_name = "obs_seq.final",
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "filter_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_state_ens_mean = .true.,
   output_state_ens_spread = .true.,
   output_obs_ens_mean = .true.,
   output_obs_ens_spread = .true.,
   num_output_state_members = 20,
```

(continues on next page)

(continued from previous page)

```
   num_output_obs_members = 20,
   output_interval = 1,
   num_groups = 1,
   outlier_threshold = -1.0 /

&ensemble_manager_nml
   in_core = .true.,
   single_restart_file_in = .true.,
   single_restart_file_out = .true. /

&assim_tools_nml
   filter_kind = 1,
   cutoff = 0.2,
   sort_obs_inc = .false.,
   cov_inflate = -1.0,
   cov_inflate_sd = 0.05,
   sd_lower_bound = 0.05,
   deterministic_cov_inflate = .true.,
   start_from_assim_restart = .false.,
   assim_restart_in_file_name =
'assim_tools_ics',
   assim_restart_out_file_name =
'assim_tools_restart',
   do_parallel = 0,
   num_domains = 1
   parallel_command = "./assim_filter.csh",
   spread_restoration = .false.,
   cov_inflate_upper_bound = 10000000.0,
   internal_outlier_threshold = -1.0 /

&cov_cutoff_nml
   select_localization = 1 /

&reg_factor_nml
   select_regression = 1,
   input_reg_file = "time_mean_reg"
   save_reg_diagnostics = .false.,
   reg_diagnostics_file = 'reg_diagnostics' /

&obs_sequence_nml
   write_binary_obs_sequence = .false. /

&obs_kind_nml
   assimilate_these_obs_types = 'RAW_STATE_VARIABLE'
/

&assim_model_nml
   write_binary_restart_files = .true. /

&model_nml
   sigma = 10.0,
   r = 28.0,
   b = 2.6666666666667,
   deltat = 0.01,
   time_step_days = 0,
   time_step_seconds = 3600 /
```

(continues on next page)

(continued from previous page)

```
&utilities_nml
   TERMLEVEL = 1
   logfilename = 'dart_log.out' /
```

Only the non-obvious(?) entries for `filter_nml` will be discussed.

namelist variable	description
`ens_size`	Number of ensemble members. 100 is sufficient for most of the L63 exercises.
`cutoff`	to limit the impact of an observation, set to 0.0 (i.e. spin-up)
`cov_inflate`	A value of 1.0 results in no inflation.(spin-up)
`start_from_restart`	when '.false.', `filter` will generate its own ensemble of initial conditions. It is important to note that the filter still makes use of `perfect_ics` by randomly perturbing these state variables.
`output_state_ens_mean`	when '.true.' the mean of all ensemble members is output.
`output_state_ens_spread`	when '.true.' the spread of all ensemble members is output.
`num_output_state_members`	may be a value from 0 to `ens_size`
`output_obs_ens_mean`	when '.true.' Output ensemble mean in observation output file.
`output_obs_ens_spread`	when '.true.' Output ensemble spread in observation output file.
`num_output_obs_members`	may be a value from 0 to `ens_size`
`output_interval`	The frequency with which output state diagnostics are written. Units are in assimilation times. Default value is 1 meaning output is written at every observation time

The filter is told to generate its own ensemble initial conditions since `start_from_restart` is '.false.'. However, it is important to note that the filter still makes use of `perfect_ics` which is set to be the `restart_in_file_name`. This is the model state generated from the first 24,000 step model integration by `perfect_model_obs`. `Filter` generates its ensemble initial conditions by randomly perturbing the state variables of this state.

The arguments `output_state_ens_mean` and `output_state_ens_spread` are '.true.' so that these quantities are output at every time for which there are observations (once a day here) and `num_output_ens_members` means that the same diagnostic files, `Posterior_Diag.nc` and `Prior_Diag.nc` also contain values for 20 ensemble members once a day. Once the namelist is set, execute `filter` to integrate the ensemble forward for 24,000 steps with the final ensemble state written to the `filter_restart`. Copy the `perfect_model_obs` restart file `perfect_restart` (the `true state') to `perfect_ics`, and the `filter` restart file `filter_restart` to `filter_ics` so that future assimilation experiments can be initialized from these spun-up states.

```
$ filter
$ cp perfect_restart perfect_ics
$ cp filter_restart filter_ics
```

The spin-up of the ensemble can be viewed by examining the output in the netCDF files `True_State.nc` generated by `perfect_model_obs` and `Posterior_Diag.nc` and `Prior_Diag.nc` generated by `filter`. To do this, see the detailed discussion of matlab diagnostics in Appendix I.

3. Simulate a particular observing system

Begin by using `create_obs_sequence` to generate an observation set in which each of the 3 state variables of L63 is observed with an observational error variance of 1.0 for each observation. To do this, use the following input sequence (the text including and after # is a comment and does not need to be entered):

4	# upper bound on num of observations in sequence
0	# number of copies of data (0 for just a definition)
0	# number of quality control values per field (0 or greater)
0	# -1 to exit/end observation definitions
-1	# observe state variable 1
0 0	# time – days, seconds
1.0	# observational variance
0	# -1 to exit/end observation definitions
-2	# observe state variable 2
0 0	# time – days, seconds
1.0	# observational variance
0	# -1 to exit/end observation definitions
-3	# observe state variable 3
0 0	# time – days, seconds
1.0	# observational variance
-1	# -1 to exit/end observation definitions
set_def.out	# Output file name

Now, generate an observation sequence definition by running `create_fixed_network_seq` with the following input sequence:

set_def.out	# Input observation set definition file
1	# Regular spaced observation interval in time
1000	# 1000 observation times
0, 43200	# First observation after 12 hours (0 days, 12 * 3600 seconds)
0, 43200	# Observations every 12 hours
obs_seq.in	# Output file for observation sequence definition

4. Generate a particular observing system and true state

An observation sequence file is now generated by running `perfect_model_obs` with the namelist values (unchanged from step 2):

```
&perfect_model_obs_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .true.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1 /
```

This integrates the model starting from the state in perfect_ics for 1000 12-hour intervals outputting synthetic observations of the three state variables every 12 hours and producing a netCDF diagnostic file, True_State.nc.

5. Filtering

Finally, filter can be run with its namelist set to:

```
&filter_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   ens_size = 100,
   cov_inflate = 1.0,
   start_from_restart = .true.,
   output_restart = .true.,
   obs_sequence_in_name = "obs_seq.out",
   obs_sequence_out_name = "obs_seq.final",
   restart_in_file_name = "filter_ics",
   restart_out_file_name = "filter_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_state_ens_mean = .true.,
   output_state_ens_spread = .true.,
   output_obs_ens_mean = .true.,
   output_obs_ens_spread = .true.,
   num_output_state_members = 20,
   num_output_obs_members = 20,
   output_interval = 1,
   num_groups = 1,
   outlier_threshold = -1.0 /
```

The large value for the cutoff allows each observation to impact all other state variables (see Appendix V for localization). filter produces two output diagnostic files, Prior_Diag.nc which contains values of the ensemble mean, ensemble spread, and ensemble members for 12- hour lead forecasts before assimilation is applied and Posterior_Diag.nc which contains similar data for after the assimilation is applied (sometimes referred to as analysis values).

Now try applying all of the matlab diagnostic functions described in the Matlab Diagnostics section.

6.17.9 Matlab® diagnostics

The output files are netCDF files, and may be examined with many different software packages. We happen to use Matlab®, and provide our diagnostic scripts in the hopes that they are useful.

The diagnostic scripts and underlying functions reside in two places: DART/diagnostics/matlab and DART/matlab. They are reliant on the public-domain netcdf toolbox from http://woodshole.er.usgs.gov/staffpages/cdenham/public_html/MexCDF/nc4ml5.html as well as the public-domain CSIRO matlab/netCDF interface from http://www.marine.csiro.au/sw/matlab-netcdf.html. If you do not have them installed on your system and want to use Matlab to peruse netCDF, you must follow their installation instructions. The 'interested reader' may want to look at the DART/matlab/startup.m file I use on my system. If you put it in your $HOME/matlab directory, it is invoked every time you start up Matlab.

Once you can access the getnc function from within Matlab, you can use our diagnostic scripts. It is necessary to prepend the location of the DART/matlab scripts to the matlabpath. Keep in mind the location of the netcdf operators on your system WILL be different from ours … and that's OK.

```
0[269]0 ghotiol:/<5>models/lorenz_63/work]$ matlab -nojvm

                                             < M A T L A B >
                                 Copyright 1984-2002 The MathWorks, Inc.
                                     Version 6.5.0.180913a Release 13
                                               Jun 18 2002

  Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.

  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

>> which getnc
/contrib/matlab/matlab_netcdf_5_0/getnc.m
>>ls *.nc

ans =

Posterior_Diag.nc  Prior_Diag.nc  True_State.nc


>>path('../../../matlab',path)
>>path('../../../diagnostics/matlab',path)
>>which plot_ens_err_spread
../../../matlab/plot_ens_err_spread.m
>>help plot_ens_err_spread

  DART : Plots summary plots of the ensemble error and ensemble spread.
                         Interactively queries for the needed information.
                         Since different models potentially need different
                         pieces of information ... the model types are
                         determined and additional user input may be queried.

  Ultimately, plot_ens_err_spread will be replaced by a GUI.
  All the heavy lifting is done by PlotEnsErrSpread.

  Example 1 (for low-order models)

  truth_file = 'True_State.nc';
  diagn_file = 'Prior_Diag.nc';
  plot_ens_err_spread

>>plot_ens_err_spread
```

And the matlab graphics window will display the spread of the ensemble error for each state variable. The scripts are designed to do the “obvious” thing for the low-order models and will prompt for additional information if needed. The philosophy of these is that anything that starts with a lower-case *plot_some_specific_task* is intended to be user-callable and should handle any of the models. All the other routines in `DART/matlab` are called BY the high-level routines.

Matlab script	description
`plot_bins`	plots ensemble rank histograms
`plot_correl`	Plots space-time series of correlation between a given variable at a given time and other variables at all times in a n ensemble time sequence.
`plot_ens_err_spread`	Plots summary plots of the ensemble error and ensemble spread. Interactively queries for the needed information. Since different models potentially need different pieces of information … the model types are determined and additional user input may be queried.
`plot_ens_mean_time_series`	Queries for the state variables to plot.
`plot_ens_time_series`	Queries for the state variables to plot.
`plot_phase_space`	Plots a 3D trajectory of (3 state variables of) a single ensemble member. Additional trajectories may be superimposed.
`plot_total_err`	Summary plots of global error and spread.
`plot_var_var_correl`	Plots time series of correlation between a given variable at a given time and another variable at all times in an ensemble time sequence.

6.17.10 Bias, filter divergence and covariance inflation (with the l63 model)

One of the common problems with ensemble filters is filter divergence, which can also be an issue with a variety of other flavors of filters including the classical Kalman filter. In filter divergence, the prior estimate of the model state becomes too confident, either by chance or because of errors in the forecast model, the observational error characteristics, or approximations in the filter itself. If the filter is inappropriately confident that its prior estimate is correct, it will then tend to give less weight to observations than they should be given. The result can be enhanced overconfidence in the model's state estimate. In severe cases, this can spiral out of control and the ensemble can wander entirely away from the truth, confident that it is correct in its estimate. In less severe cases, the ensemble estimates may not diverge entirely from the truth but may still be too confident in their estimate. The result is that the truth ends up being farther away from the filter estimates than the spread of the filter ensemble would estimate. This type of behavior is commonly detected using rank histograms (also known as Talagrand diagrams). You can see the rank histograms for the L63 initial assimilation by using the matlab script `plot_bins`.

A simple, but surprisingly effective way of dealing with filter divergence is known as covariance inflation. In this method, the prior ensemble estimate of the state is expanded around its mean by a constant factor, effectively increasing the prior estimate of uncertainty while leaving the prior mean estimate unchanged. The program `filter` has a namelist parameter that controls the application of covariance inflation, `cov_inflate`. Up to this point, `cov_inflate` has been set to 1.0 indicating that the prior ensemble is left unchanged. Increasing `cov_inflate` to values greater than 1.0 inflates the ensemble before assimilating observations at each time they are available. Values smaller than 1.0 contract (reduce the spread) of prior ensembles before assimilating.

You can do this by modifying the value of `cov_inflate` in the namelist, (try 1.05 and 1.10 and other values at your discretion) and run the filter as above. In each case, use the diagnostic matlab tools to examine the resulting changes to the error, the ensemble spread (via rank histogram bins, too), etc. What kind of relation between spread and error is seen in this model?

6.17.11 Synthetic observations

Synthetic observations are generated from a `perfect' model integration, which is often referred to as the `truth' or a `nature run'. A model is integrated forward from some set of initial conditions and observations are generated as $y = H(x) + e$ where H is an operator on the model state vector, x, that gives the expected value of a set of observations, y, and e is a random variable with a distribution describing the error characteristics of the observing instrument(s) being simulated. Using synthetic observations in this way allows students to learn about assimilation algorithms while being isolated from the additional (extreme) complexity associated with model error and unknown observational error characteristics. In other words, for the real-world assimilation problem, the model has (often substantial) differences from what happens in the real system and the observational error distribution may be very complicated and is certainly not well known. Be careful to keep these issues in mind while exploring the capabilities of the ensemble filters with synthetic observations.

6.18 Hawaii

6.18.1 DART Hawaii release documentation

> **Attention:** Hawaii is a prior release of DART. Its source code is available via the DART repository on Github. This documentation is preserved merely for reference. See the DART homepage to learn about the latest release.

6.18.2 Overview of DART

The Data Assimilation Research Testbed (DART) is designed to facilitate the combination of assimilation algorithms, models, and observation sets to allow increased understanding of all three. The DART programs have been compiled with the Intel 7.1 Fortran compiler and run on a linux compute-server. If your system is different, you will definitely need to read the Customizations section.

DART programs can require three different types of input. First, some of the DART programs, those for creating synthetic observational datasets, require interactive input from the keyboard. For simple cases, this interactive input can be made directly from the keyboard. In more complicated cases, a file containing the appropriate keyboard input can be created and this file can be directed to the standard input of the DART program. Second, many DART programs expect one or more input files in DART specific formats to be available. For instance, `perfect_model_obs` creates a synthetic observation set given a particular model and a description of a sequence of observations requires an input file that describes this observation sequence. At present, the observation files for DART are inefficient but human-readable ascii files in a custom format. Third, many DART modules (including main programs) make use of the Fortan90 namelist facility to obtain values of certain parameters at run-time. All programs look for a namelist input file called `input.nml` in the directory in which the program is executed. The `input.nml` file can contain a sequence of individual Fortran90 namelists which specify values of particular parameters for modules that compose the executable program. Unfortunately, the Fortran90 namelist interface is poorly defined in the language standard, leaving considerable leeway to compiler developers in implementing the facility. The Intel 7.1 compiler has some particularly unpleasant behavior when a namelist file contains an entry that is NOT defined in the program reading the namelist. Error behavior is unpredictable, but often results in read errors for other input files opened by DART programs. If you encounter run-time read errors, the first course of action should be to ensure the components of the namelist are actual components. Changing the names of the namelist components **will** create unpleasantries. DART provides a mechanism that automatically generates namelists with the default values for each program to be run.

DART uses the netCDF self-describing data format with a particular metadata convention to describe output that is used to analyze the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. A set of Matlab scripts, designed to produce graphical diagnostics from DART netCDF output files are available. DART users have also used ncview to create rudimentary graphical displays of output data fields. The NCO tools, produced by UCAR's Unidata group, are available to do operations like concatenating, slicing, and dicing of netCDF files.

6.18.3 Requirements: an F90 compiler

The DART software has been successfully built on the following:

machine	architecture	compiler
anchorage	2.4GHz Xeon running RH7.3	Intel Fortran CompilerV 7.1
dart,fisher,ocotillo	2.6GHz Xeon running Fedora Core 2	Intel Fortran Compiler V 8.0.046,
dart,fisher	2.6GHz Xeon running Fedora Core 2	Portland Group Fortran Compiler V 5.2.4
dart,fisher	2.6GHz Xeon running Fedora Core 2	Lahey LF95 Compiler V 5.2.4
tarpon	G4 PowerBook running OSX 10.3.8	Absoft Pro Fortran for Mac OSX V 9.0
bluesky	IBM running AIX	IBM XLF Compiler

Since recompiling the code is a necessity to experiment with different models, there are no binaries to distribute.

DART uses the netCDF self-describing data format for the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. In particular, DART also makes use of the F90 interface to the library which is available through the `netcdf.mod` and `typesizes.mod` modules. *IMPORTANT*: different compilers create these modules with different "case" filenames, and sometimes they are not **both** installed into the expected directory. It is required that both modules be present. The normal place would be in the `netcdf/include` directory, as opposed to the `netcdf/lib` directory.

If the netCDF library does not exist on your system, you must build it (as well as the F90 interface modules). The library and instructions for building the library or installing from an RPM may be found at the netCDF home page: http://www.unidata.ucar.edu/packages/netcdf/ Pay particular attention to the compiler-specific patches that must be applied for the Intel Fortran Compiler. (Or the PG compiler, for that matter.)

The location of the netCDF library, `libnetcdf.a`, and the locations of both `netcdf.mod` and `typesizes.mod` will be needed by the makefile template, as described in the compiling section.

Certain components of DART (i.e. the MPI version of the bgrid model) also use the **very** common udunits library for manipulating units of physical quantities. If, somehow, it is not installed on your system, you will need to install it (instructions are available from Unidata's Downloads page).

The location of the udunits library, `libudunits.a`, will be needed by the makefile template, as described in the compiling section. **If you are not using the MPI version of the bgrid model, you should remove the ``libudunits.a`` option from the makefile template.**

6.18.4 Unpacking the distribution

This release of the DART source code can be downloaded as a compressed zip or tar.gz file. When extracted, the source tree will begin with a directory named `DART` and will be approximately 30.7 Mb. Compiling the code in this tree (as is usually the case) will necessitate much more space.

```
$ gunzip DART-4.0.0.tar.gz
$ tar -xvf DART-4.0.0.tar
```

The code tree is very "bushy"; there are many directories of support routines, etc. but only a few directories involved with the customization and installation of the DART software. If you can compile and run ONE of the low-order models, you should be able to compile and run ANY of the low-order models. For this reason, we can focus on the Lorenz `63 model. Subsequently, the only directories with files to be modified to check the installation are: `DART_hawaii/mkmf`, `DART_hawaii/models/lorenz_63/work`, and `DART_hawaii/matlab` (but only for analysis).

6.18.5 Customizing the build scripts – overview

DART executable programs are constructed using two tools: `make` and `mkmf`. The `make` utility is a relatively common piece of software that requires a user-defined input file that records dependencies between different source files. `make` then performs a hierarchy of actions when one or more of the source files is modified. The `mkmf` utility is a custom preprocessor that generates a `make` input file (named `Makefile`) and an example namelist `input.nml.mkmf` with the default values. The `Makefile` is designed specifically to work with object-oriented Fortran90 (and other languages) for systems like DART.

`mkmf` requires two separate input files. The first is a `template' file which specifies details of the commands required for a specific Fortran90 compiler and may also contain pointers to directories containing pre-compiled utilities required by the DART system. **This template file will need to be modified to reflect your system**. The second input file is a `path_names' file which includes a complete list of the locations (either relative or absolute) of all Fortran90 source files that are required to produce a particular DART program. Each 'path_names' file must contain a path for exactly one Fortran90 file containing a main program, but may contain any number of additional paths pointing to files containing Fortran90 modules. An `mkmf` command is executed which uses the 'path_names' file and the mkmf template file to produce a `Makefile` which is subsequently used by the standard `make` utility.

Shell scripts that execute the mkmf command for all standard DART executables are provided as part of the standard DART software. For more information on `mkmf` see the FMS mkmf description.

One of the benefits of using `mkmf` is that it also creates an example namelist file for each program. The example namelist is called `input.nml.`*filter*`_default`, for example, so as not to clash with any exising `input.nml` that may exist in that directory.

Building and customizing the 'mkmf.template' file

A series of templates for different compilers/architectures exists in the `DART_hawaii/mkmf/` directory and have names with extensions that identify either the compiler, the architecture, or both. This is how you inform the build process of the specifics of your system. Our intent is that you copy one that is similar to your system into `mkmf.template` and customize it. For the discussion that follows, knowledge of the contents of one of these templates (i.e. `mkmf.template.pgf90.ghotiol`) is needed: (note that only the first few uncommented lines are shown here)

```
FC = pgf90
LD = pgf90
CPPFLAGS =
LIST = -Mlist
NETCDF = /contrib/netcdf-3.5.1-cc-c++-pgif90.5.2-4
FFLAGS = -O0 -Ktrap=fp -pc 64 -I$(NETCDF)/include
LIBS = -L$(NETCDF)/lib -lnetcdf
LDFLAGS = $(LIBS)

# you should never need to change any lines below.
...
```

Essentially, each of the lines defines some part of the resulting `Makefile`. Since `make` is particularly good at sorting out dependencies, the order of these lines really doesn't make any difference. The `FC = pgf90` line ultimately defines the Fortran90 compiler to use, etc.

FFLAGS

Each compiler has different compile flags, so there is really no way to exhaustively cover this other than to say the templates as we supply them should work – we usually turn the optimization off and try to use 64 bit arithmetic instead of 80 so we can more reasonably compare the results across architectures.

Netcdf

The variable which most likely needs a site-specific change is `NETCDF`. Configure your `NETCDF` variable such that you have a
`$(NETCDF)/include/typesizes.mod`
`$(NETCDF)/include/netcdf.mod`
`$(NETCDF)/lib/libnetcdf.a`
Depending on the compiler, the case of the modules might be different, i.e., your system might have a `TYPESIZES.mod`, or `Typesizes.mod...` anything goes.

Customizing the 'path_names_*' file

Several `path_names_*` files are provided in the `work` directory for each specific model, in this case: `DART_hawaii/models/lorenz_63/work`.

1. `path_names_create_obs_sequence`
2. `path_names_create_fixed_network_seq`
3. `path_names_perfect_model_obs`
4. `path_names_filter`

Since each model comes with its own set of files, no further customization is needed.

6.18.6 Building the Lorenz_63 DART project

Currently, DART executables are constructed in a `work` subdirectory under the directory containing code for the given model. In the top-level DART directory, change to the L63 work directory and list the contents:

```
$ cd DART_hawaii/models/lorenz_63/work
$ ls -1
```

With the result:

```
filter_ics
mkmf_create_fixed_network_seq
mkmf_create_obs_sequence
mkmf_filter
mkmf_perfect_model_obs
path_names_create_fixed_network_seq
path_names_create_obs_sequence
path_names_filter
path_names_perfect_model_obs
perfect_ics
```

There are four `mkmf_`*xxxxxx* files for the programs `create_obs_sequence`, `create_fixed_network_seq`, `perfect_model_obs`, and `filter` along with the corresponding `path_names_`*xxxxxx* files. You can examine the contents of one of the path_names_*xxxxxx* files, for instance path_names_filter, to see a list of the relative paths of all files that contain Fortran90 modules required for the program filter for the L63 model. All of these paths are relative to your DART_hawaii directory. The first path is the main program (filter.f90) and is followed by all the Fortran90 modules used by this program.

The mkmf_*xxxxxx* scripts are cryptic but should not need to be modified – as long as you do not restructure the code tree (by moving directories, for example). The only function of the mkmf_*xxxxxx* script is to generate a Makefile and an instance of the default namelist file: input.nml.*xxxxxx*_default. It is not supposed to compile anything.

```
$ csh mkmf_create_obs_sequence
$ make
```

The first command generates an appropriate Makefile and the input.nml.create_obs_sequence_default file. The make command results in the compilation of a series of Fortran90 modules which ultimately produces an executable file: create_obs_sequence. Should you need to make any changes to the DART_hawaii/mkmf/mkmf.template, (*i.e.* change compile options) you will need to regenerate the Makefile. A series of object files for each module compiled will also be left in the work directory, as some of these are undoubtedly needed by the build of the other DART components. You can proceed to create the other three programs needed to work with L63 in DART as follows:

```
$ csh mkmf_create_fixed_network_seq
$ make
$ csh mkmf_perfect_model_obs
$ make
$ csh mkmf_filter
$ make
```

The result (hopefully) is that four executables now reside in your work directory. The most common problem is that the netCDF libraries and include files (particularly typesizes.mod) are not found. If this is the case; edit the DART_hawaii/mkmf/mkmf.template, recreate the Makefile, and try again.

program	purpose
create_obs_sequence	specify a (set) of observation characteristics taken by a particular (set of) instruments
create_fixed_network_seq	specify the temporal attributes of the observation sets
perfect_model_obs	spinup, generate “true state” for synthetic observation experiments, . . .
filter	perform experiments

6.18.7 Running Lorenz_63

This initial sequence of exercises includes detailed instructions on how to work with the DART code and allows investigation of the basic features of one of the most famous dynamical systems, the 3-variable Lorenz-63 model. The remarkable complexity of this simple model will also be used as a case study to introduce a number of features of a simple ensemble filter data assimilation system. To perform a synthetic observation assimilation experiment for the L63 model, the following steps must be performed (an overview of the process is given first, followed by detailed procedures for each step):

6.18.8 Experiment overview

1. Integrate the L63 model for a long time starting from arbitrary initial conditions to generate a model state that lies on the attractor. The ergodic nature of the L63 system means a 'lengthy' integration always converges to some point on the computer's finite precision representation of the model's attractor.
2. Generate a set of ensemble initial conditions from which to start an assimilation. Since L63 is ergodic, the ensemble members can be designed to look like random samples from the model's 'climatological distribution'. To generate an ensemble member, very small perturbations can be introduced to the state on the attractor generated by step 1. This perturbed state can then be integrated for a very long time until all memory of its initial condition can be viewed as forgotten. Any number of ensemble initial conditions can be generated by repeating this procedure.
3. Simulate a particular observing system by first creating an 'observation set definition' and then creating an 'observation sequence'. The 'observation set definition' describes the instrumental characteristics of the observations and the 'observation sequence' defines the temporal sequence of the observations.
4. Populate the 'observation sequence' with 'perfect' observations by integrating the model and using the information in the 'observation sequence' file to create simulated observations. This entails operating on the model state at the time of the observation with an appropriate forward operator (a function that operates on the model state vector to produce the expected value of the particular observation) and then adding a random sample from the observation error distribution specified in the observation set definition. At the same time, diagnostic output about the 'true' state trajectory can be created.
5. Assimilate the synthetic observations by running the filter; diagnostic output is generated.

1. Integrate the L63 model for a 'long' time

`perfect_model_obs` integrates the model for all the times specified in the 'observation sequence definition' file. To this end, begin by creating an 'observation sequence definition' file that spans a long time. Creating an 'observation sequence definition' file is a two-step procedure involving `create_obs_sequence` followed by `create_fixed_network_seq`. After they are both run, it is necessary to integrate the model with `perfect_model_obs`.

1.1 Create an observation set definition

`create_obs_sequence` creates an observation set definition, the time-independent part of an observation sequence. An observation set definition file only contains the `location, type,` and `observational error characteristics` (normally just the diagonal observational error variance) for a related set of observations. There are no actual observations. For spin-up, we are only interested in integrating the L63 model, not in generating any particular synthetic observations. Begin by creating a minimal observation set definition.

In general, for the low-order models, only a single observation set need be defined. Next, the number of individual scalar observations (like a single surface pressure observation) in the set is needed. To spin-up an initial condition for the L63 model, only a single observation is needed. Next, the error variance for this observation must be entered. Since we do not need (nor want) this observation to have any impact on an assimilation (it will only be used for spinning up the model and the ensemble), enter a very large value for the error variance. An observation with a very large error variance has essentially no impact on deterministic filter assimilations like the default variety implemented in DART. Finally, the location and type of the observation need to be defined. For all types of models, the most elementary form of synthetic observations are called 'identity' observations. These observations are generated simply by adding a random sample from a specified observational error distribution directly to the value of one of the state variables. This defines the observation as being an identity observation of the first state variable in the L63 model. The program will respond by terminating after generating a file (generally named `set_def.out`) that defines the single identity observation of the first state variable of the L63 model. The following is a screenshot (much of the verbose logging has been left off for clarity), the user input looks *like this*.

```
[unixprompt]$ ./create_obs_sequence
 Initializing the utilities module.
 Trying to log to unit           10
 Trying to open file dart_log.out

 Registering module :
 $Source$
 $Revision$
 $Date$
 Registration complete.

 &UTILITIES_NML
 TERMLEVEL= 2,LOGFILENAME=dart_log.out
 /

{ ... }

 Registering module :
 $Source$
 $Revision$
 $Date$
 Registration complete.

 static_init_obs_sequence obs_sequence_nml values are
 &OBS_SEQUENCE_NML
 READ_BINARY_OBS_SEQUENCE= F,WRITE_BINARY_OBS_SEQUENCE= F
 /
 Input upper bound on number of observations in sequence
10000
 Input number of copies of data (0 for just a definition)
0
 Input number of quality control values per field (0 or greater)
0
 input a -1 if there are no more obs
0

 Registering module :
 $Source$
 $Revision$
 $Date$
 Registration complete.


 Registering module :
 $Source$
 $Revision$
 $Date$
 Registration complete.

 input obs kind: u =            1  v =            2  ps =            3  t =
           4  qv =            5  p =            6  w =            7  qr =
           8  Td =           10  rho =           11  Vr =          100  Ref =
         101  U10 =          200   V10 =         201  T2 =          202  Q2 =
         203
 input -1 times the state variable index for an identity observation
-2
 input time in days and seconds
```

(continues on next page)

(continued from previous page)

```
1 0
 input error variance for this observation definition
1000000
 calling insert obs in sequence
 back from insert obs in sequence
 input a -1 if there are no more obs
-1
 Input filename for sequence (  set_def.out   usually works well)
set_def.out
 write_obs_seq  opening formatted file set_def.out
 write_obs_seq  closed file set_def.out
```

Two files are created. `set_def.out` is the empty template containing the metadata for the observation(s). `dart_log.out` contains run-time diagnostics from `create_obs_sequence`.

1.2 Create a (temporal) network of observations

`create_fixed_network_seq` creates an 'observation network definition' by extending the 'observation set definition' with the temporal attributes of the observations.

The first input is the name of the file created in the previous step, *i.e.* the name of the observation set definition that you've just created. It is possible to create sequences in which the observation sets are observed at regular intervals or irregularly in time. Here, all we need is a sequence that takes observations over a long period of time - indicated by entering a 1. Although the L63 system normally is defined as having a non-dimensional time step, the DART system arbitrarily defines the model timestep as being 3600 seconds. By declaring we have 1000 observations taken once per day, we create an observation sequence definition spanning 24000 'model' timesteps; sufficient to spin-up the model onto the attractor. Finally, enter a name for the 'observation sequence definition' file. Note again: there are no observation values present in this file. Just an observation type, location, time and the error characteristics. We are going to populate the observation sequence with the `perfect_model_obs` program.

```
[thoar@ghotiol work]$ ./create_fixed_network_seq
 Initializing the utilities module.
 Trying to log to unit           10
 Trying to open file dart_log.out

 Registering module :
 $Source$
 $Revision$
 $Date$
 Registration complete.

 { ... }

 static_init_obs_sequence obs_sequence_nml values are
 &OBS_SEQUENCE_NML
 READ_BINARY_OBS_SEQUENCE= F,WRITE_BINARY_OBS_SEQUENCE= F
 /
 Input filename for network definition sequence (usually  set_def.out  )
set_def.out

 Registering module :
 $Source$
 $Revision$
```

(continues on next page)

(continued from previous page)

```
 $Date$
 Registration complete.


 Registering module :
 $Source$
 $Revision$
 $Date$
 Registration complete.

 To input a regularly repeating time sequence enter 1
 To enter an irregular list of times enter 2
1
 Input number of observation times in sequence
1000
 Input initial time in sequence
 input time in days and seconds (as integers)
1 0
 Input period of obs in sequence in days and seconds
1 0

       { ... }

         997
         998
         999
        1000
 What is output file name for sequence (  obs_seq.in   is recommended )
obs_seq.in
 write_obs_seq  opening formatted file obs_seq.in
 write_obs_seq  closed file obs_seq.in
```

1.3 Initialize the model onto the attractor

perfect_model_obs can now advance the arbitrary initial state for 24,000 timesteps to move it onto the attractor.

perfect_model_obs uses the Fortran90 namelist input mechanism instead of (admittedly gory, but temporary) interactive input. All of the DART software expects the namelists to found in a file called input.nml. When you built the executable, an example namelist was created input.nml.perfect_model_obs_default that contains all of the namelist input for the executable. We must now rename and customize the namelist file for perfect_model_obs. Copy input.nml.perfect_model_obs_default to input.nml and edit it to look like the following:

```
&perfect_model_obs_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .false.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
```

(continues on next page)

(continued from previous page)

```
   init_time_seconds = 0,
   output_interval = 1 /

&ensemble_manager_nml
   in_core = .true.,
   single_restart_file_in = .true.,
   single_restart_file_out = .true. /

&assim_tools_nml
   filter_kind = 1,
   cutoff = 0.2,
   sort_obs_inc = .false.,
   cov_inflate = -1.0,
   cov_inflate_sd = 0.05,
   sd_lower_bound = 0.05,
   deterministic_cov_inflate = .true.,
   start_from_assim_restart = .false.,
   assim_restart_in_file_name =
'assim_tools_ics'
   assim_restart_out_file_name =
'assim_tools_restart'
   do_parallel = 0,
   num_domains = 1,
   parallel_command = "./assim_filter.csh" /

&cov_cutoff_nml
   select_localization = 1 /

&reg_factor_nml
   select_regression = 1,
   input_reg_file = "time_mean_reg" /

&obs_sequence_nml
   read_binary_obs_sequence = .false.,
   write_binary_obs_sequence = .false. /

&assim_model_nml
   read_binary_restart_files = .true.,
   write_binary_restart_files = .true. /

&model_nml
   sigma = 10.0,
   r = 28.0,
   b = 2.6666666666667,
   deltat = 0.01,
   time_step_days = 0,
   time_step_days = 3600 /

&utilities_nml
   TERMLEVEL = 1,
   logfilename = 'dart_log.out' /
```

For the moment, only two namelists warrant explanation. Each namelists is covered in detail in the html files accompanying the source code for the module. `perfect_model_obs_nml`:

namelist variable	description
`async`	For the lorenz_63, simply ignore this. Leave it set to ‘0’
`obs_seq_in_file_name`	specifies the file name that results from running `create_fixed_network_seq`, i.e. the ‘observation sequence definition’ file.
`obs_seq_out_file_name`	specifies the output file name containing the ‘observation sequence’, finally populated with (perfect?) ‘observations’.
`start_from_restart`	When set to ‘false’, `perfect_model_obs` generates an arbitrary initial condition (which cannot be guaranteed to be on the L63 attractor).
`output_restart`	When set to ‘true’, `perfect_model_obs` will record the model state at the end of this integration in the file named by `restart_out_file_name`.
`restart_in_file_name`	is ignored when ‘start_from_restart’ is ‘false’.
`restart_out_file_name`	if `output_restart` is ‘true’, this specifies the name of the file containing the model state at the end of the integration.
`init_time_`*xxxx*	the start time of the integration.
`output_interval`	interval at which to save the model state.

`utilities_nml`:

namelist variable	description
`TERMLEVEL`	When set to ‘1’ the programs terminate when a ‘warning’ is generated. When set to ‘2’ the programs terminate only with ‘fatal’ errors.
`logfilename`	Run-time diagnostics are saved to this file. This namelist is used by all programs, so the file is opened in APPEND mode. Subsequent executions cause this file to grow. **Please make sure you always look at the bottom of the file for the most recent info.**

Executing `perfect_model_obs` will integrate the model 24,000 steps and output the resulting state in the file `perfect_restart`. Interested parties can check the spinup in the `True_State.nc` file.

./perfect_model_obs

Five files are created/updated:

`True_State.nc`		Contains the trajectory of the model
`perfect_restart`		Contains the model state at the end of the integration.
`obs_seq.out`		Contains the ‘perfect’ observations (since this is a spinup, they are of questionable value, at best).
`go_end_filter`		A ‘flag’ file that is not used by this model.
`dart_log.out`		**Appends** the run-time diagnostic output to an existing file, or creates a new file with the output.

2. Generate a set of ensemble initial conditions

The set of initial conditions for a ‘perfect model’ experiment is created by taking the spun-up state of the model (available in `perfect_restart`), running `perfect_model_obs` to generate the ‘true state’ of the experiment and a corresponding set of observations, and then feeding the same initial spun-up state and resulting observations into `filter`.

Generating ensemble initial conditions is achieved by changing a perfect_model_obs namelist parameter, copying `perfect_restart` to `perfect_ics`, and rerunning `perfect_model_obs`. This execution of `perfect_model_obs` will advance the model state from the end of the first 24,000 steps (i.e. the spun-up state) to the end of an additional 24,000 steps and place the final state in `perfect_restart`. The rest of the namelists in `input.nml` should remain unchanged.

```
&perfect_model_obs_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .true.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1 /
```

```
$ cp perfect_restart perfect_ics
$ ./perfect_model_obs
```

Five files are created/updated:

`True_State.nc`		Contains the trajectory of the model
`perfect_restart`		Contains the model state at the end of the integration.
`obs_seq.out`		Contains the ‘perfect’ observations.
`go_end_filter`		A ‘flag’ file that is not used by this model.
`dart_log.out`		**Appends** the run-time diagnostic output to an existing file, or creates a new file with the output.

Generating the ensemble

is done with the program `filter`, which also uses the Fortran90 namelist mechanism for input. It is now necessary to copy the `input.nml.filter_default` namelist to `input.nml`. Having the `perfect_model_obs` namelist in the `input.nml` does not hurt anything. In fact, I generally create a single `input.nml` that has all the namelist blocks in it by copying the `perfect_model_obs` block into the `input.nml.filter_default` and then rename it `input.nml`. This same namelist file may then also be used for `perfect_model_obs`.

```
   &filter_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   ens_size = 80,
   cov_inflate = 1.00,
   start_from_restart = .false.,
   output_restart = .true.,
   obs_sequence_in_name = "obs_seq.out",
   obs_sequence_out_name = "obs_seq.final",
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "filter_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_state_ens_mean = .true.,
   output_state_ens_spread = .true.,
   output_obs_ens_mean = .true.,
   output_obs_ens_spread = .true.,
```

(continues on next page)

(continued from previous page)

```
   num_output_state_members = 80,
   num_output_obs_members = 80,
   output_interval = 1,
   num_groups = 1,
   confidence_slope = 0.0,
   outlier_threshold = -1.0,
   save_reg_series = .false. /

&perfect_model_obs_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .true.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1 /

&ensemble_manager_nml
   in_core = .true.,
   single_restart_file_in = .true.,
   single_restart_file_out = .true. /

&assim_tools_nml
   filter_kind = 1,
   cutoff = 0.2,
   sort_obs_inc = .false.,
   cov_inflate = -1.0,
   cov_inflate_sd = 0.05,
   sd_lower_bound = 0.05,
   deterministic_cov_inflate = .true.,
   start_from_assim_restart = .false.,
   assim_restart_in_file_name =
'assim_tools_ics'
   assim_restart_out_file_name =
'assim_tools_restart'
   do_parallel = 0,
   num_domains = 1,
   parallel_command = "./assim_filter.csh" /

&cov_cutoff_nml
   select_localization = 1 /

&reg_factor_nml
   select_regression = 1,
   input_reg_file = "time_mean_reg" /

&obs_sequence_nml
   read_binary_obs_sequence = .false.,
   write_binary_obs_sequence = .false. /

&assim_model_nml
   read_binary_restart_files = .true.,
   write_binary_restart_files = .true. /
```

(continues on next page)

(continued from previous page)

```
&model_nml
   sigma = 10.0,
   r = 28.0,
   b = 2.6666666666667,
   deltat = 0.01
   time_step_days = 0
   time_step_days = 3600 /

&utilities_nml
   TERMLEVEL = 1
   logfilename = 'dart_log.out' /
```

Only the non-obvious(?) entries for `filter_nml` will be discussed.

namelist variable	description
`ens_size`	Number of ensemble members. 20 is sufficient for most of the L63 exercises.
`cutoff`	to limit the impact of an observation, set to 0.0 (i.e. spin-up)
`cov_inflate`	A value of 1.0 results in no inflation.(spin-up)
`start_from_restart`	when '.false.', `filter` will generate its own set of initial conditions. It is important to note that the filter still makes use of `perfect_ics` by randomly perturbing these state variables.
`num_output_state_members`	may be a value from 0 to `ens_size`
`num_output_obs_members`	may be a value from 0 to `ens_size`
`output_state_ens_mean`	when '.true.' the mean of all ensemble members is output.
`output_state_ens_spread`	when '.true.' the spread of all ensemble members is output.
`output_obs_ens_mean`	when '.true.' the mean of all ensemble members observations is output.
`output_obs_ens_spread`	when '.true.' the spread of all ensemble members observations is output.
`output_interval`	seconds

The filter is told to generate its own ensemble initial conditions since `start_from_restart` is '.false.'. However, it is important to note that the filter still makes use of `perfect_ics` which is set to be the `restart_in_file_name`. This is the model state generated from the first 24,000 step model integration by `perfect_model_obs`. `Filter` generates its ensemble initial conditions by randomly perturbing the state variables of this state.

The arguments `output_state_ens_mean` and `output_state_ens_spread` are '.true.' so that these quantities are output at every time for which there are observations (once a day here) and `num_output_state_members` means that the same diagnostic files, `Posterior_Diag.nc` and `Prior_Diag.nc` also contain values for all 20 ensemble members once a day. Once the namelist is set, execute `filter` to integrate the ensemble forward for 24,000 steps with the final ensemble state written to the `filter_restart`. Copy the `perfect_model_obs` restart file `perfect_restart` (the `true state') to `perfect_ics`, and the `filter` restart file `filter_restart` to `filter_ics` so that future assimilation experiments can be initialized from these spun-up states.

```
./filter
cp perfect_restart perfect_ics
cp filter_restart filter_ics
```

The spin-up of the ensemble can be viewed by examining the output in the netCDF files `True_State.nc` generated by `perfect_model_obs` and `Posterior_Diag.nc` and `Prior_Diag.nc` generated by `filter`. To do this, see the detailed discussion of matlab diagnostics in Appendix I.

3. Simulate a particular observing system

Begin by using `create_obs_sequence` to generate an observation set in which each of the 3 state variables of L63 is observed with an observational error variance of 1.0 for each observation. To do this, use the following input sequence (the text including and after # is a comment and does not need to be entered):

100

# upper bound on number of observations in this sequence

0

# number of copies of data (0 == define)

0

# number of quality control values per field

0

# anything to keep going … -1 exits program

-1

# identity observation for state variable 1

0 0

# relative time of observation

1.0

# Variance of first observation

0

# anything to keep going … -1 exits program

-2

# identity observation for state variable 2

0 0

# relative time of observation

1.0

# Variance of second observation

0

# anything to keep going … -1 exits program

-3

# identity observation for state variable 3

0 0

# relative time of observation

1.0

# Variance of third observation

-1

# … -1 exits program (finally)

set_def.out

# Output file name

Now, generate an observation sequence definition by running `create_fixed_network_seq` with the following input sequence:

set_def.out	# Input observation set definition file
1	# Regular spaced observation interval in time
1000	# 1000 observation times
0, 43200	# First observation after 12 hours (0 days, 3600 * 12 seconds)
0, 43200	# Observations every 12 hours
obs_seq.in	# Output file for observation sequence definition

4. Generate a particular observing system and true state

An observation sequence file is now generated by running `perfect_model_obs` with the namelist values (unchanged from step 2):

```
&perfect_model_obs_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .true.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1 /
```

This integrates the model starting from the state in `perfect_ics` for 1000 12-hour intervals outputting synthetic observations of the three state variables every 12 hours and producing a netCDF diagnostic file, `True_State.nc`.

5. Filtering

Finally, `filter` can be run with its namelist set to:

```
&filter_nml
   async = 0,
   ens_size = 20,
   cov_inflate = 1.00,
   start_from_restart = .true.,
   output_restart = .true.,
   obs_sequence_file_name = "obs_seq.out",
   restart_in_file_name = "filter_ics",
   restart_out_file_name = "filter_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_state_ens_mean = .true.,
   output_state_ens_spread = .true.,
   num_output_ens_members = 20,
   output_interval = 1,
   num_groups = 1,
```

(continues on next page)

(continued from previous page)

```
   confidence_slope = 0.0,
   output_obs_diagnostics = .false.,
   get_mean_reg = .false.,
   get_median_reg = .false.     /
...
&assim_tools_nml
   filter_kind = 1,
   cutoff = 22222222.0,
...
```

The large value for the cutoff allows each observation to impact all other state variables (see Appendix V for localization). `filter` produces two output diagnostic files, `Prior_Diag.nc` which contains values of the ensemble members, ensemble mean and ensemble spread for 12- hour lead forecasts before assimilation is applied and `Posterior_Diag.nc` which contains similar data for after the assimilation is applied (sometimes referred to as analysis values).

Now try applying all of the matlab diagnostic functions described in the Matlab Diagnostics section.

6.18.9 Matlab® diagnostics

The output files are netCDF files, and may be examined with many different software packages. We happen to use Matlab®, and provide our diagnostic scripts in the hopes that they are useful.

The Matlab diagnostic scripts and underlying functions reside in the `DART_hawaii/matlab` directory. They are reliant on the public-domain netcdf toolbox from `http://woodshole.er.usgs.gov/staffpages/cdenham/public_html/MexCDF/nc4ml5.html` as well as the public-domain CSIRO matlab/netCDF interface from `http://www.marine.csiro.au/sw/matlab-netcdf.html`. If you do not have them installed on your system and want to use Matlab to peruse netCDF, you must follow their installation instructions.

Once you can access the `getnc` function from within Matlab, you can use our diagnostic scripts. It is necessary to prepend the location of the DART_hawaii/matlab scripts to the matlabpath. Keep in mind the location of the netcdf operators on your system WILL be different from ours … and that's OK.

```
0[269]0 ghotiol:/<5>models/lorenz_63/work]$ matlab -nojvm

                                             < M A T L A B >
                                 Copyright 1984-2002 The MathWorks, Inc.
                                     Version 6.5.0.180913a Release 13
                                               Jun 18 2002

  Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.

  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

>> which getnc
/contrib/matlab/matlab_netcdf_5_0/getnc.m
>>ls *.nc

ans =

Posterior_Diag.nc  Prior_Diag.nc  True_State.nc


>>path('../../../matlab',path)
```

(continues on next page)

(continued from previous page)

```
>>which plot_ens_err_spread
../../../matlab/plot_ens_err_spread.m
>>help plot_ens_err_spread

  DART : Plots summary plots of the ensemble error and ensemble spread.
                         Interactively queries for the needed information.
                         Since different models potentially need different
                         pieces of information ... the model types are
                         determined and additional user input may be queried.

  Ultimately, plot_ens_err_spread will be replaced by a GUI.
  All the heavy lifting is done by PlotEnsErrSpread.

  Example 1 (for low-order models)

  truth_file = 'True_State.nc';
  diagn_file = 'Prior_Diag.nc';
  plot_ens_err_spread

>>plot_ens_err_spread
```

And the matlab graphics window will display the spread of the ensemble error for each state variable. The scripts are designed to do the "obvious" thing for the low-order models and will prompt for additional information if needed. The philosophy of these is that anything that starts with a lower-case *plot_some_specific_task* is intended to be user-callable and should handle any of the models. All the other routines in `DART_hawaii/matlab` are called BY the high-level routines.

Matlab script	description
`plot_bins`	plots ensemble rank histograms
`plot_correl`	Plots space-time series of correlation between a given variable at a given time and other variables at all times in a n ensemble time sequence.
`plot_ens_err_spread`	Plots summary plots of the ensemble error and ensemble spread. Interactively queries for the needed information. Since different models potentially need different pieces of information ... the model types are determined and additional user input may be queried.
`plot_ens_mean_time_series`	Queries for the state variables to plot.
`plot_ens_time_series`	Queries for the state variables to plot.
`plot_phase_space`	Plots a 3D trajectory of (3 state variables of) a single ensemble member. Additional trajectories may be superimposed.
`plot_total_err`	Summary plots of global error and spread.
`plot_var_var_correl`	Plots time series of correlation between a given variable at a given time and another variable at all times in an ensemble time sequence.

6.18.10 Bias, filter divergence and covariance inflation (with the l63 model)

One of the common problems with ensemble filters is filter divergence, which can also be an issue with a variety of other flavors of filters including the classical Kalman filter. In filter divergence, the prior estimate of the model state becomes too confident, either by chance or because of errors in the forecast model, the observational error characteristics, or approximations in the filter itself. If the filter is inappropriately confident that its prior estimate is correct, it will then tend to give less weight to observations than they should be given. The result can be enhanced overconfidence in the model's state estimate. In severe cases, this can spiral out of control and the ensemble can wander entirely away from the truth, confident that it is correct in its estimate. In less severe cases, the ensemble estimates may not diverge entirely from the truth but may still be too confident in their estimate. The result is that the truth ends up being farther away from the filter estimates than the spread of the filter ensemble would estimate. This type of behavior is commonly detected using rank histograms (also known as Talagrand diagrams). You can see the rank histograms for the L63 initial assimilation by using the matlab script `plot_bins`.

A simple, but surprisingly effective way of dealing with filter divergence is known as covariance inflation. In this method, the prior ensemble estimate of the state is expanded around its mean by a constant factor, effectively increasing the prior estimate of uncertainty while leaving the prior mean estimate unchanged. The program `filter` has a namelist parameter that controls the application of covariance inflation, `cov_inflate`. Up to this point, `cov_inflate` has been set to 1.0 indicating that the prior ensemble is left unchanged. Increasing `cov_inflate` to values greater than 1.0 inflates the ensemble before assimilating observations at each time they are available. Values smaller than 1.0 contract (reduce the spread) of prior ensembles before assimilating.

You can do this by modifying the value of `cov_inflate` in the namelist, (try 1.05 and 1.10 and other values at your discretion) and run the filter as above. In each case, use the diagnostic matlab tools to examine the resulting changes to the error, the ensemble spread (via rank histogram bins, too), etc. What kind of relation between spread and error is seen in this model?

6.18.11 Synthetic observations

Synthetic observations are generated from a `perfect' model integration, which is often referred to as the `truth' or a `nature run'. A model is integrated forward from some set of initial conditions and observations are generated as $y = H(x) + e$ where H is an operator on the model state vector, x, that gives the expected value of a set of observations, y, and e is a random variable with a distribution describing the error characteristics of the observing instrument(s) being simulated. Using synthetic observations in this way allows students to learn about assimilation algorithms while being isolated from the additional (extreme) complexity associated with model error and unknown observational error characteristics. In other words, for the real-world assimilation problem, the model has (often substantial) differences from what happens in the real system and the observational error distribution may be very complicated and is certainly not well known. Be careful to keep these issues in mind while exploring the capabilities of the ensemble filters with synthetic observations.

6.19 Guam

6.19.1 DART Guam release documentation

Attention: Guam is a prior release of DART. Its source code is available via the DART repository on Github. This documentation is preserved merely for reference. See the DART homepage to learn about the latest release.

6.19.2 Overview of DART

The Data Assimilation Research Testbed (DART) is designed to facilitate the combination of assimilation algorithms, models, and observation sets to allow increased understanding of all three. The DART programs have been compiled with the Intel 7.1 Fortran compiler and run on a linux compute-server. If your system is different, you will definitely need to read the Customizations section.

The Guam release provides several new models and has a fundamentally different implementation of the low-level routines that will now operate on a much broader class of observations. As such, the first two program units have changed somewhat. The strategy of these two programs units is still the same; declare the characteristics of the observations and then declare the temporal nature of the observations. These are still destined to be run as a back-end to some sort of GUI-driven interface, so virtually no work has been done to change the level of user interaction required to run these programs.

The Guam release has also been tested with more compilers in an attempt to determine non-portable code elements. It is my experience that the largest impediment to portable code is the reliance on the compiler to autopromote `real` variables to one flavor or another. Using the F90 "kind" allows for much more flexible code, in that the use of interface procedures is possible only when two routines do not have identical sets of input arguments – something that happens when the compiler autopromotes 32bit reals to 64bit reals, for example.

DART programs can require three different types of input. First, some of the DART programs, those for creating synthetic observational datasets, require interactive input from the keyboard. For simple cases, this interactive input can be made directly from the keyboard. In more complicated cases, a file containing the appropriate keyboard input can be created and this file can be directed to the standard input of the DART program. Second, many DART programs expect one or more input files in DART specific formats to be available. For instance, `perfect_model_obs` creates a synthetic observation set given a particular model and a description of a sequence of observations requires an input file that describes this observation sequence. At present, the observation files for DART are inefficient but human-readable ascii files in a custom format. Third, many DART modules (including main programs) make use of the Fortan90 namelist facility to obtain values of certain parameters at run-time. All programs look for a namelist input file called `input.nml` in the directory in which the program is executed. The `input.nml` file can contain a sequence of individual Fortran90 namelists which specify values of particular parameters for modules that compose the executable program. Unfortunately, the Fortran90 namelist interface is poorly defined in the language standard, leaving considerable leeway to compiler developers in implementing the facility. The Intel 7.1 compiler has some particularly unpleasant behavior when a namelist file contains an entry that is NOT defined in the program reading the namelist. Error behavior is unpredictable, but often results in read errors for other input files opened by DART programs. If you encounter run-time read errors, the first course of action should be to ensure the components of the namelist are actual components. Changing the names of the namelist components **will** create unpleasantries. DART provides a mechanism that automatically generates namelists with the default values for each program to be run.

DART uses the netCDF self-describing data format with a particular metadata convention to describe output that is used to analyze the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. A set of Matlab scripts, designed to produce graphical diagnostics from DART netCDF output files are available. DART users have also used ncview to create rudimentary graphical displays of output data fields. The NCO tools, produced by UCAR's Unidata group, are available to do operations like concatenating, slicing, and dicing of netCDF files.

6.19.3 Requirements: an F90 compiler

The DART software has been successfully built on several Linux/x86 platforms with the Intel Fortran Compiler 7.1 for Linux, which is free for individual scientific use. It has also been built and successfully run with the Portland Group Fortran Compiler (5.02), and again with the Intel 8.0.034 compiler. Since recompiling the code is a necessity to experiment with different models, there are no binaries to distribute.

DART uses the netCDF self-describing data format for the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. In particular, DART also makes use of the F90 interface to the library which are available through the `netcdf.mod` and `typesizes.mod` modules. *IMPORTANT*: different compilers create these modules with different "case" filenames, and sometimes they are not **both** installed into the expected directory. It is required that both modules be present. The normal place would be in the `netcdf/include` directory, as opposed to the `netcdf/lib` directory.

If the netCDF library does not exist on your system, you must build it (as well as the F90 interface modules). The library and instructions for building the library or installing from an RPM may be found at the netCDF home page: http://www.unidata.ucar.edu/packages/netcdf/ Pay particular attention to the compiler-specific patches that must be applied for the Intel Fortran Compiler. (Or the PG compiler, for that matter.)

The location of the netCDF library, `libnetcdf.a`, and the locations of both `netcdf.mod` and `typesizes.mod` will be needed by the makefile template, as described in the compiling section.

DART also uses the **very** common udunits library for manipulating units of physical quantities. If, somehow, it is not installed on your system, you will need to install it (instructions are available from Unidata's Downloads page).

The location of the udunits library, libudunits.a, will be needed by the makefile template, as described in the compiling section.

6.19.4 Unpacking the distribution

This release of the DART source code can be downloaded as a compressed zip or tar.gz file. When extracted, the source tree will begin with a directory named DART and will be approximately 28.6 Mb. Compiling the code in this tree (as is usually the case) will necessitate much more space.

```
$ gunzip DART-3.0.0.tar.gz
$ tar -xvf DART-3.0.0.tar
```

The code tree is very “bushy”; there are many directories of support routines, etc. but only a few directories involved with the customization and installation of the DART software. If you can compile and run ONE of the low-order models, you should be able to compile and run ANY of the low-order models. For this reason, we can focus on the Lorenz `63 model. Subsequently, the only directories with files to be modified to check the installation are: DART/mkmf, DART/models/lorenz_63/work, and DART/matlab (but only for analysis).

6.19.5 Customizing the build scripts – overview

DART executable programs are constructed using two tools: make and mkmf. The make utility is a relatively common piece of software that requires a user-defined input file that records dependencies between different source files. make then performs a hierarchy of actions when one or more of the source files is modified. The mkmf utility is a custom preprocessor that generates a make input file (named Makefile) and an example namelist input.nml.mkmf with the default values. The Makefile is designed specifically to work with object-oriented Fortran90 (and other languages) for systems like DART.

mkmf requires two separate input files. The first is a `template’ file which specifies details of the commands required for a specific Fortran90 compiler and may also contain pointers to directories containing pre-compiled utilities required by the DART system. **This template file will need to be modified to reflect your system**. The second input file is a `path_names’ file which includes a complete list of the locations (either relative or absolute) of all Fortran90 source files that are required to produce a particular DART program. Each ‘path_names’ file must contain a path for exactly one Fortran90 file containing a main program, but may contain any number of additional paths pointing to files containing Fortran90 modules. An mkmf command is executed which uses the ‘path_names’ file and the mkmf template file to produce a Makefile which is subsequently used by the standard make utility.

Shell scripts that execute the mkmf command for all standard DART executables are provided as part of the standard DART software. For more information on mkmf see the FMS mkmf description.

One of the benefits of using mkmf is that it also creates an example namelist file for each program. The example namelist is called input.nml.mkmf, so as not to clash with any exising input.nml that may exist in that directory.

Building and customizing the ‘mkmf.template’ file

A series of templates for different compilers/architectures exists in the DART/mkmf/ directory and have names with extensions that identify either the compiler, the architecture, or both. This is how you inform the build process of the specifics of your system. Our intent is that you copy one that is similar to your system into mkmf.template and customize it. For the discussion that follows, knowledge of the contents of one of these templates (i.e. mkmf.template.pgi) is needed: (note that only the first few lines are shown here).

```
# Makefile template for PGI f90
FC = pgf90
CPPFLAGS =
FFLAGS = -r8 -Ktrap=fp -pc 64 -I/usr/local/netcdf/include
LD = pgf90
LDFLAGS = $(LIBS)
LIBS = -L/usr/local/netcdf/lib -lnetcdf
-L/usr/local/udunits-1.11.7/lib -ludunits
LIST = -Mlist

# you should never need to change any lines below.
...
```

Essentially, each of the lines defines some part of the resulting `Makefile`. Since `make` is particularly good at sorting out dependencies, the order of these lines really doesn't make any difference. The `FC = pgf90` line ultimately defines the Fortran90 compiler to use, etc. The lines which are most likely to need site-specific changes start with `FFLAGS` and `LIBS`, which indicate where to look for the netCDF F90 modules and the location of the netCDF and udunits libraries.

FFLAGS

Each compiler has different compile flags, so there is really no way to exhaustively cover this other than to say the templates as we supply them should work – depending on the location of the netCDF modules `netcdf.mod` and `typesizes.mod`. Change the `/usr/local/netcdf/include` string to reflect the location of your modules. The low-order models can be compiled without the `-r8` switch, but the `bgrid_solo` model cannot.

Libs

Modifying the `LIBS` value should be relatively straightforward.
Change the `/usr/local/netcdf/lib` string to reflect the location of your `libnetcdf.a`.
Change the `/usr/local/udunits-1.11.7/lib` string to reflect the location of your `libudunits.a`.

Customizing the 'path_names_*' file

Several `path_names_*` files are provided in the `work` directory for each specific model, in this case: `DART/models/lorenz_63/work`.

1. `path_names_create_obs_sequence`
2. `path_names_create_fixed_network_seq`
3. `path_names_perfect_model_obs`
4. `path_names_filter`

Since each model comes with its own set of files, no further customization is needed.

6.19.6 Building the Lorenz_63 DART project

Currently, DART executables are constructed in a work subdirectory under the directory containing code for the given model. In the top-level DART directory, change to the L63 work directory and list the contents:

```
$ cd DART/models/lorenz_63/work
$ ls -1
```

With the result:

```
filter_ics
mkmf_create_obs_sequence
mkmf_create_fixed_network_seq
mkmf_filter
mkmf_perfect_model_obs
path_names_create_obs_sequence
path_names_create_fixed_network_seq
path_names_filter
path_names_perfect_model_obs
perfect_ics
```

There are four mkmf_*xxxxxx* files for the programs create_obs_sequence, create_obs_fixed_network_seq, perfect_model_obs, and filter along with the corresponding path_names_*xxxxxx* files. You can examine the contents of one of the path_names_*xxxxxx* files, for instance path_names_filter, to see a list of the relative paths of all files that contain Fortran90 modules required for the program filter for the L63 model. All of these paths are relative to your DART directory. The first path is the main program (filter.f90) and is followed by all the Fortran90 modules used by this program.

The mkmf_*xxxxxx* scripts are cryptic but should not need to be modified – as long as you do not restructure the code tree (by moving directories, for example).

The only function of the mkmf_*xxxxxx* script is to generate a Makefile and an input.nml.mkmf file. It is not supposed to compile anything:

```
$ csh mkmf_create_obs_sequence
$ make
```

The first command generates an appropriate Makefile and the input.nml.create_obs_sequence_default file. The second command results in the compilation of a series of Fortran90 modules which ultimately produces an executable file: create_obs_sequence. Should you need to make any changes to the DART/mkmf/mkmf.template, you will need to regenerate the Makefile. A series of object files for each module compiled will also be left in the work directory, as some of these are undoubtedly needed by the build of the other DART components. You can proceed to create the other three programs needed to work with L63 in DART as follows:

```
$ csh mkmf_create_fixed_network_seq
$ make
$ csh mkmf_perfect_model_obs
$ make
$ csh mkmf_filter
$ make
```

The result (hopefully) is that four executables now reside in your work directory. The most common problem is that the netCDF libraries and include files (particularly typesizes.mod) are not found. Edit the DART/mkmf/mkmf.template, recreate the Makefile, and try again.

program	purpose
`create_obs_sequence`	specify a (set) of observation characteristics taken by a particular (set of) instruments
`create_fixed_network_seq`	specify the temporal attributes of the observation sets
`perfect_model_obs`	spinup, generate "true state" for synthetic observation experiments, . . .
`filter`	perform experiments

6.19.7 Running Lorenz_63

This initial sequence of exercises includes detailed instructions on how to work with the DART code and allows investigation of the basic features of one of the most famous dynamical systems, the 3-variable Lorenz-63 model. The remarkable complexity of this simple model will also be used as a case study to introduce a number of features of a simple ensemble filter data assimilation system. To perform a synthetic observation assimilation experiment for the L63 model, the following steps must be performed (an overview of the process is given first, followed by detailed procedures for each step):

6.19.8 Experiment overview

1. Integrate the L63 model for a long time starting from arbitrary initial conditions to generate a model state that lies on the attractor. The ergodic nature of the L63 system means a 'lengthy' integration always converges to some point on the computer's finite precision representation of the model's attractor.
2. Generate a set of ensemble initial conditions from which to start an assimilation. Since L63 is ergodic, the ensemble members can be designed to look like random samples from the model's 'climatological distribution'. To generate an ensemble member, very small perturbations can be introduced to the state on the attractor generated by step 1. This perturbed state can then be integrated for a very long time until all memory of its initial condition can be viewed as forgotten. Any number of ensemble initial conditions can be generated by repeating this procedure.
3. Simulate a particular observing system by first creating an 'observation set definition' and then creating an 'observation sequence'. The 'observation set definition' describes the instrumental characteristics of the observations and the 'observation sequence' defines the temporal sequence of the observations.
4. Populate the 'observation sequence' with 'perfect' observations by integrating the model and using the information in the 'observation sequence' file to create simulated observations. This entails operating on the model state at the time of the observation with an appropriate forward operator (a function that operates on the model state vector to produce the expected value of the particular observation) and then adding a random sample from the observation error distribution specified in the observation set definition. At the same time, diagnostic output about the 'true' state trajectory can be created.
5. Assimilate the synthetic observations by running the filter; diagnostic output is generated.

1. Integrate the L63 model for a 'long' time

`perfect_model_obs` integrates the model for all the times specified in the 'observation sequence definition' file. To this end, begin by creating an 'observation sequence definition' file that spans a long time. Creating an 'observation sequence definition' file is a two-step procedure involving `create_obs_sequence` followed by `create_fixed_network_seq`. After they are both run, it is necessary to integrate the model with `perfect_model_obs`.

1.1 Create an observation set definition

create_obs_sequence creates an observation set definition, the time-independent part of an observation sequence. An observation set definition file only contains the location, type, and observational error characteristics (normally just the diagonal observational error variance) for a related set of observations. There are no actual observations, nor are there any times associated with the definition. For spin-up, we are only interested in integrating the L63 model, not in generating any particular synthetic observations. Begin by creating a minimal observation set definition.

In general, for the low-order models, only a single observation set need be defined. Next, the number of individual scalar observations (like a single surface pressure observation) in the set is needed. To spin-up an initial condition for the L63 model, only a single observation is needed. Next, the error variance for this observation must be entered. Since we do not need (nor want) this observation to have any impact on an assimilation (it will only be used for spinning up the model and the ensemble), enter a very large value for the error variance. An observation with a very large error variance has essentially no impact on deterministic filter assimilations like the default variety implemented in DART. Finally, the location and type of the observation need to be defined. For all types of models, the most elementary form of synthetic observations are called 'identity' observations. These observations are generated simply by adding a random sample from a specified observational error distribution directly to the value of one of the state variables. This defines the observation as being an identity observation of the first state variable in the L63 model. The program will respond by terminating after generating a file (generally named set_def.out) that defines the single identity observation of the first state variable of the L63 model. The following is a screenshot (much of the verbose logging has been left off for clarity), the user input looks *like this*.

```
[unixprompt]$ ./create_obs_sequence
 Initializing the utilities module.
 Trying to read from unit           10
 Trying to open file dart_log.out

 Registering module :
 $source: /home/dart/CVS.REPOS/DART/utilities/utilities_mod.f90,v $
 $revision: 1.18 $
 $date: 2004/06/29 15:16:40 $
 Registration complete.

 &UTILITIES_NML
 TERMLEVEL= 2,LOGFILENAME=dart_log.out

 /

 Registering module :
 $source: /home/dart/CVS.REPOS/DART/obs_sequence/create_obs_sequence.f90,v $
 $revision: 1.18 $
 $date: 2004/05/24 15:41:46 $
 Registration complete.

 { ... }

 Input upper bound on number of observations in sequence
10

 Input number of copies of data (0 for just a definition)
0

 Input number of quality control values per field (0 or greater)
0

```

(continues on next page)

(continued from previous page)

```
 input a -1 if there are no more obs
0

 Registering module :
 $source: /home/dart/CVS.REPOS/DART/obs_def/obs_def_mod.f90,v $
 $revision: 1.21 $
 $date: 2004/06/25 16:17:43 $
 Registration complete.

 Registering module :
 $source: /home/dart/CVS.REPOS/DART/obs_kind/obs_kind_mod.f90,v $
 $revision: 1.15 $
 $date: 2004/06/24 21:49:47 $
 Registration complete.

 input obs kind: u =            1  v =            2  ps =            3  t =
           4  qv =            5  p =            6  Td =           10  Vr =
         100  Ref =          101
 input -1 times the state variable index for an identity observation
-1

 input time in days and seconds
1 0

 input time in days and seconds
1 0

 Input error variance for this observation definition
1000000

 input a -1 if there are no more obs
-1

 Input filename for sequence (  set_def.out   usually works well)
 set_def.out
 write_obs_seq  opening formatted file set_def.out
 write_obs_seq  closed file set_def.out
```

1.2 Create an observation sequence definition

`create_fixed_network_seq` creates an ‘observation sequence definition’ by extending the ‘observation set definition’ with the temporal attributes of the observations.

The first input is the name of the file created in the previous step, i.e. the name of the observation set definition that you’ve just created. It is possible to create sequences in which the observation sets are observed at regular intervals or irregularly in time. Here, all we need is a sequence that takes observations over a long period of time - indicated by entering a 1. Although the L63 system normally is defined as having a non-dimensional time step, the DART system arbitrarily defines the model timestep as being 3600 seconds. By declaring we have 1000 observations taken once per day, we create an observation sequence definition spanning 24000 ‘model’ timesteps; sufficient to spin-up the model onto the attractor. Finally, enter a name for the ‘observation sequence definition’ file. Note again: there are no observation values present in this file. Just an observation type, location, time and the error characteristics. We are going to populate the observation sequence with the `perfect_model_obs` program.

```
[unixprompt]$ ./create_fixed_network_seq

 ...

 Registering module :
 $source: /home/dart/CVS.REPOS/DART/obs_sequence/obs_sequence_mod.f90,v $
 $revision: 1.31 $
 $date: 2004/06/29 15:04:37 $
 Registration complete.

 Input filename for network definition sequence (usually  set_def.out  )
set_def.out

 ...

 To input a regularly repeating time sequence enter 1
 To enter an irregular list of times enter 2
1
 Input number of observations in sequence
1000
 Input time of initial ob in sequence in days and seconds
1, 0
 Input period of obs in days and seconds
1, 0
 time             1  is             0            1
 time             2  is             0            2
 time             3  is             0            3
...
 time           998  is             0          998
 time           999  is             0          999
 time          1000  is             0         1000
What is output file name for sequence (  obs_seq.in   is recommended )
obs_seq.in
```

1.3 Initialize the model onto the attractor

perfect_model_obs can now advance the arbitrary initial state for 24,000 timesteps to move it onto the attractor.

perfect_model_obs uses the Fortran90 namelist input mechanism instead of (admittedly gory, but temporary) interactive input. All of the DART software expects the namelists to found in a file called input.nml. When you built the executable, an example namelist was created input.nml.mkmf that contains all of the namelist input for the executable. If you followed the example, each namelist was saved to a unique name. We must now rename and edit the namelist file for perfect_model_obs. Copy input.nml.perfect_model_obs to input.nml and edit it to look like the following:

```
&perfect_model_obs_nml
   async = 0,
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .false.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
```

(continues on next page)

(continued from previous page)

```
   init_time_seconds = 0,
   output_interval = 1 /
&assim_tools_nml
   prior_spread_correction = .false.,
   filter_kind = 1,
   slope_threshold = 1.0 /
&cov_cutoff_nml
   select_localization = 1 /
&assim_model_nml
   binary_restart_files = .true. /
&model_nml
   sigma = 10.0,
   r = 28.0,
   b = 2.6666666666667,
   deltat = 0.01 /
&utilities_nml
   TERMLEVEL = 1
   logfilename = 'dart_log.out' /
```

For the moment, only two namelists warrant explanation. Each namelists is covered in detail in the html files accompanying the source code for the module.

perfect_model_obs_nml

namelist variable	description
`async`	For the lorenz_63, simply ignore this. Leave it set to '0'
`obs_seq_in_file_name`	specifies the file name that results from running `create_fixed_network_seq`, i.e. the 'observation sequence definition' file.
`obs_seq_out_file_name`	specifies the output file name containing the 'observation sequence', finally populated with (perfect?) 'observations'.
`start_from_restart`	When set to 'false', `perfect_model_obs` generates an arbitrary initial condition (which cannot be guaranteed to be on the L63 attractor).
`output_restart`	When set to 'true', `perfect_model_obs` will record the model state at the end of this integration in the file named by `restart_out_file_name`.
`restart_in_file_name`	is ignored when 'start_from_restart' is 'false'.
`restart_out_file_name`	if `output_restart` is 'true', this specifies the name of the file containing the model state at the end of the integration.
`init_time_`*xxxx*	the start time of the integration.
`output_interval`	interval at which to save the model state.

utilities_nml

namelist variable	description
`TERMLEVEL`	When set to '1' the programs terminate when a 'warning' is generated. When set to '2' the programs terminate only with 'fatal' errors.
`logfilename`	Run-time diagnostics are saved to this file. This namelist is used by all programs, so the file is opened in APPEND mode. Subsequent executions cause this file to grow.

Executing `perfect_model_obs` will integrate the model 24,000 steps and output the resulting state in the file `perfect_restart`. Interested parties can check the spinup in the `True_State.nc` file.

```
$ perfect_model_obs
```

2. Generate a set of ensemble initial conditions

The set of initial conditions for a 'perfect model' experiment is created by taking the spun-up state of the model (available in `perfect_restart`), running `perfect_model_obs` to generate the 'true state' of the experiment and a corresponding set of observations, and then feeding the same initial spun-up state and resulting observations into `filter`.
Generating ensemble initial conditions is achieved by changing a perfect_model_obs namelist parameter, copying `perfect_restart` to `perfect_ics`, and rerunning `perfect_model_obs`. This execution of `perfect_model_obs` will advance the model state from the end of the first 24,000 steps to the end of an additional 24,000 steps and place the final state in `perfect_restart`. The rest of the namelists in `input.nml` should remain unchanged.

```
&perfect_model_obs_nml
   async = 0,
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .true.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1 /
```

```
$ cp perfect_restart perfect_ics
$ perfect_model_obs
```

A `True_State.nc` file is also created. It contains the 'true' state of the integration.

Generating the ensemble

is done with the program `filter`, which also uses the Fortran90 namelist mechanism for input. It is now necessary to copy the `input.nml.filter` namelist to `input.nml` or you may simply insert the `filter_nml` namelist into the existing `input.nml`. Having the `perfect_model_obs` namelist in the input.nml does not hurt anything. In fact, I generally create a single `input.nml` that has all the namelist blocks in it.

```
&perfect_model_obs_nml
   async = 0,
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .true.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1 /
&assim_tools_nml
   prior_spread_correction = .false.,
```

(continues on next page)

(continued from previous page)

```
   filter_kind = 1,
   slope_threshold = 1.0 /
&cov_cutoff_nml
   select_localization = 1 /
&assim_model_nml
   binary_restart_files = .true. /
&model_nml
   sigma = 10.0,
   r = 28.0,
   b = 2.6666666666667
   deltat = 0.01 /
&utilities_nml
   TERMLEVEL = 1
   logfilename = 'dart_log.out' /
&reg_factor_nml
   select_regression = 1,
   input_reg_file = "time_mean_reg" /
&filter_nml
   async = 0,
   ens_size = 20,
   cutoff = 0.20,
   cov_inflate = 1.00,
   start_from_restart = .false.,
   output_restart = .true.,
   obs_sequence_file_name = "obs_seq.out",
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "filter_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_state_ens_mean = .true.,
   output_state_ens_spread = .true.,
   num_output_ens_members = 20,
   output_interval = 1,
   num_groups = 1,
   confidence_slope = 0.0,
   output_obs_diagnostics = .false.,
   get_mean_reg = .false.,
   get_median_reg = .false. /
```

Only the non-obvious(?) entries for `filter_nml` will be discussed.

namelist variable	description
`ens_size`	Number of ensemble members. 20 is sufficient for most of the L63 exercises.
`cutoff`	to limit the impact of an observation, set to 0.0 (i.e. spin-up)
`cov_inflate`	A value of 1.0 results in no inflation.(spin-up)
`start_from_restart`	when '.false.', `filter` will generate its own set of initial conditions. It is important to note that the filter still makes use of `perfect_ics` by randomly perturbing these state variables.
`num_output_ens_members`	may be a value from 0 to `ens_size`
`output_state_ens_mean`	when '.true.' the mean of all ensemble members is output.
`output_state_ens_spread`	when '.true.' the spread of all ensemble members is output.
`output_interval`	Jeff - units for interval?

The filter is told to generate its own ensemble initial conditions since `start_from_restart` is '.false.'. However, it is important to note that the filter still makes use of `perfect_ics` which is set to be the `restart_in_file_name`. This is the model state generated from the first 24,000 step model integration by `perfect_model_obs`. `Filter` generates its ensemble initial conditions by randomly perturbing the state variables of this state.

The arguments `output_state_ens_mean` and `output_state_ens_spread` are '.true.' so that these quantities are output at every time for which there are observations (once a day here) and `num_output_ens_members` means that the same diagnostic files, `Posterior_Diag.nc` and `Prior_Diag.nc` also contain values for all 20 ensemble members once a day. Once the namelist is set, execute `filter` to integrate the ensemble forward for 24,000 steps with the final ensemble state written to the `filter_restart`. Copy the `perfect_model_obs` restart file `perfect_restart` (the `true state') to `perfect_ics`, and the `filter` restart file `filter_restart` to `filter_ics` so that future assimilation experiments can be initialized from these spun-up states.

```
$ filter
$ cp perfect_restart perfect_ics
$ cp filter_restart filter_ics
```

The spin-up of the ensemble can be viewed by examining the output in the netCDF files `True_State.nc` generated by `perfect_model_obs` and `Posterior_Diag.nc` and `Prior_Diag.nc` generated by `filter`. To do this, see the detailed discussion of matlab diagnostics in Appendix I.

3. Simulate a particular observing system

Begin by using `create_obs_sequence` to generate an observation set in which each of the 3 state variables of L63 is observed with an observational error variance of 1.0 for each observation. To do this, use the following input sequence (the text including and after # is a comment and does not need to be entered):

4	# upper bound on num of observations in sequence
0	# number of copies of data (0 for just a definition)
0	# number of quality control values per field (0 or greater)
0	# -1 to exit/end observation definitions
-1	# observe state variable 1
0 0	# time – days, seconds
1.0	# observational variance
0	# -1 to exit/end observation definitions
-2	# observe state variable 2
0 0	# time – days, seconds
1.0	# observational variance
0	# -1 to exit/end observation definitions
-3	# observe state variable 3
0 0	# time – days, seconds
1.0	# observational variance
-1	# -1 to exit/end observation definitions
set_def.out	# Output file name

Now, generate an observation sequence definition by running `create_fixed_network_seq` with the following input sequence:

set_def.out	# Input observation set definition file
1	# Regular spaced observation interval in time
1000	# 1000 observation times
0, 43200	# First observation after 12 hours (0 days, 3600 * 12 seconds)
0, 43200	# Observations every 12 hours
obs_seq.in	# Output file for observation sequence definition

4. Generate a particular observing system and true state

An observation sequence file is now generated by running `perfect_model_obs` with the namelist values (unchanged from step 2):

```
&perfect_model_obs_nml
   async = 0,
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .true.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1 /
```

This integrates the model starting from the state in `perfect_ics` for 1000 12-hour intervals outputting synthetic observations of the three state variables every 12 hours and producing a netCDF diagnostic file, `True_State.nc`.

5. Filtering

Finally, `filter` can be run with its namelist set to:

```
&filter_nml
   async = 0,
   ens_size = 20,
   cutoff = 22222222.0,
   cov_inflate = 1.00,
   start_from_restart = .true.,
   output_restart = .true.,
   obs_sequence_file_name = "obs_seq.out",
   restart_in_file_name = "filter_ics",
   restart_out_file_name = "filter_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_state_ens_mean = .true.,
   output_state_ens_spread = .true.,
   num_output_ens_members = 20,
   output_interval = 1,
   num_groups = 1,
   confidence_slope = 0.0,
   output_obs_diagnostics = .false.,
   get_mean_reg = .false.,
   get_median_reg = .false. /
```

The large value for the cutoff allows each observation to impact all other state variables (see Appendix V for localization). `filter` produces two output diagnostic files, `Prior_Diag.nc` which contains values of the ensemble members, ensemble mean and ensemble spread for 12- hour lead forecasts before assimilation is applied and `Posterior_Diag.nc` which contains similar data for after the assimilation is applied (sometimes referred to as analysis values).

Now try applying all of the matlab diagnostic functions described in the Matlab Diagnostics section.

6.19.9 Matlab diagnostics

The output files are netCDF files, and may be examined with many different software packages. We happen to use Matlab, and provide our diagnostic scripts in the hopes that they are useful.

The Matlab diagnostic scripts and underlying functions reside in the `DART/matlab` directory. They are reliant on the public-domain netcdf toolbox from `http://woodshole.er.usgs.gov/staffpages/cdenham/public_html/MexCDF/nc4ml5.html` as well as the public-domain CSIRO matlab/netCDF interface from `http://www.marine.csiro.au/sw/matlab-netcdf.html`. If you do not have them installed on your system and want to use Matlab to peruse netCDF, you must follow their installation instructions.

Once you can access the `getnc` function from within Matlab, you can use our diagnostic scripts. It is necessary to prepend the location of the DART/matlab scripts to the matlabpath. Keep in mind the location of the netcdf operators on your system WILL be different from ours . . . and that's OK.

```
0[269]0 ghotiol:/<5>models/lorenz_63/work]$ matlab -nojvm

                                             < M A T L A B >
                                 Copyright 1984-2002 The MathWorks, Inc.
                                     Version 6.5.0.180913a Release 13
                                               Jun 18 2002

  Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.

  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

>> which getnc
/contrib/matlab/matlab_netcdf_5_0/getnc.m
>>ls *.nc

ans =

Posterior_Diag.nc  Prior_Diag.nc  True_State.nc


>>path('../../../matlab',path)
>>which plot_ens_err_spread
../../../matlab/plot_ens_err_spread.m
>>help plot_ens_err_spread

  DART : Plots summary plots of the ensemble error and ensemble spread.
                         Interactively queries for the needed information.
                         Since different models potentially need different
                         pieces of information ... the model types are
                         determined and additional user input may be queried.

  Ultimately, plot_ens_err_spread will be replaced by a GUI.
  All the heavy lifting is done by PlotEnsErrSpread.

  Example 1 (for low-order models)

  truth_file = 'True_State.nc';
  diagn_file = 'Prior_Diag.nc';
  plot_ens_err_spread

>>plot_ens_err_spread
```

And the matlab graphics window will display the spread of the ensemble error for each state variable. The scripts are designed to do the "obvious" thing for the low-order models and will prompt for additional information if needed. The philosophy of these is that anything that starts with a lower-case *plot_some_specific_task* is intended to be user-callable and should handle any of the models. All the other routines in `DART/matlab` are called BY the high-level routines.

Matlab script	description
`plot_bins`	plots ensemble rank histograms
`plot_correl`	Plots space-time series of correlation between a given variable at a given time and other variables at all times in a n ensemble time sequence.
`plot_ens_err_spread`	Plots summary plots of the ensemble error and ensemble spread. Interactively queries for the needed information. Since different models potentially need different pieces of information ... the model types are determined and additional user input may be queried.
`plot_ens_mean_time_series`	Queries for the state variables to plot.
`plot_ens_time_series`	Queries for the state variables to plot.
`plot_phase_space`	Plots a 3D trajectory of (3 state variables of) a single ensemble member. Additional trajectories may be superimposed.
`plot_total_err`	Summary plots of global error and spread.
`plot_var_var_correl`	Plots time series of correlation between a given variable at a given time and another variable at all times in an ensemble time sequence.

6.19.10 Bias, filter divergence and covariance inflation (with the l63 model)

One of the common problems with ensemble filters is filter divergence, which can also be an issue with a variety of other flavors of filters including the classical Kalman filter. In filter divergence, the prior estimate of the model state becomes too confident, either by chance or because of errors in the forecast model, the observational error characteristics, or approximations in the filter itself. If the filter is inappropriately confident that its prior estimate is correct, it will then tend to give less weight to observations then they should be given. The result can be enhanced overconfidence in the model's state estimate. In severe cases, this can spiral out of control and the ensemble can wander entirely away from the truth, confident that it is correct in its estimate. In less severe cases, the ensemble estimates may not diverge entirely from the truth but may still be too confident in their estimate. The result is that the truth ends up being farther away from the filter estimates than the spread of the filter ensemble would estimate. This type of behavior is commonly detected using rank histograms (also known as Talagrand diagrams). You can see the rank histograms for the L63 initial assimilation by using the matlab script `plot_bins`.

A simple, but surprisingly effective way of dealing with filter divergence is known as covariance inflation. In this method, the prior ensemble estimate of the state is expanded around its mean by a constant factor, effectively increasing the prior estimate of uncertainty while leaving the prior mean estimate unchanged. The program `filter` has a namelist parameter that controls the application of covariance inflation, `cov_inflate`. Up to this point, `cov_inflate` has been set to 1.0 indicating that the prior ensemble is left unchanged. Increasing `cov_inflate` to values greater than 1.0 inflates the ensemble before assimilating observations at each time they are available. Values smaller than 1.0 contract (reduce the spread) of prior ensembles before assimilating.

You can do this by modifying the value of `cov_inflate` in the namelist, (try 1.05 and 1.10 and other values at your discretion) and run the filter as above. In each case, use the diagnostic matlab tools to examine the resulting changes to the error, the ensemble spread (via rank histogram bins, too), etc. What kind of relation between spread and error is seen in this model?

6.19.11 Synthetic observations

Synthetic observations are generated from a `perfect' model integration, which is often referred to as the `truth' or a `nature run'. A model is integrated forward from some set of initial conditions and observations are generated as $y = H(x) + e$ where H is an operator on the model state vector, x, that gives the expected value of a set of observations, y, and e is a random variable with a distribution describing the error characteristics of the observing instrument(s) being simulated. Using synthetic observations in this way allows students to learn about assimilation algorithms while being isolated from the additional (extreme) complexity associated with model error and unknown observational error characteristics. In other words, for the real-world assimilation problem, the model has (often substantial) differences from what happens in the real system and the observational error distribution may be very complicated and is certainly not well known. Be careful to keep these issues in mind while exploring the capabilities of the ensemble filters with synthetic observations.

6.20 Fiji

6.20.1 DART Fiji release documentation

Attention: Fiji is a prior release of DART. Its source code is available via the DART repository on Github. This documentation is preserved merely for reference. See the DART homepage for information on the latest release.

6.20.2 Overview of DART

The Data Assimilation Research Testbed (DART) is designed to facilitate the combination of assimilation algorithms, models, and observation sets to allow increased understanding of all three. The DART programs have been compiled with the Intel 7.1 Fortran compiler and run on a linux compute-server. If your system is different, you will definitely need to read the Customizations section.

DART programs can require three different types of input. First, some of the DART programs, those for creating synthetic observational datasets, require interactive input from the keyboard. For simple cases, this interactive input can be made directly from the keyboard. In more complicated cases, a file containing the appropriate keyboard input can be created and this file can be directed to the standard input of the DART program. Second, many DART programs expect one or more input files in DART specific formats to be available. For instance, `perfect_model_obs` creates a synthetic observation set given a particular model and a description of a sequence of observations requires an input file that describes this observation sequence. At present, the observation files for DART are inefficient but human-readable ascii files in a custom format. Third, many DART modules (including main programs) make use of the Fortan90 namelist facility to obtain values of certain parameters at run-time. All programs look for a namelist input file called `input.nml` in the directory in which the program is executed. The `input.nml` file can contain a sequence of individual Fortran90 namelists which specify values of particular parameters for modules that compose the executable program. Unfortunately, the Fortran90 namelist interface is poorly defined in the language standard, leaving considerable leeway to compiler developers in implementing the facility. The Intel 7.1 compiler has some particularly unpleasant behavior when a namelist file contains an entry that is NOT defined in the program reading the namelist. Error behavior is unpredictable, but often results in read errors for other input files opened by DART programs. If you encounter run-time read errors, the first course of action should be to ensure the components of the namelist are actual components. Changing the names of the namelist components **will** create unpleasantries. DART provides a mechanism that automatically generates namelists with the default values for each program to be run.

DART uses the netCDF self-describing data format with a particular metadata convention to describe output that is used to analyze the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. A set of Matlab scripts, designed to produce graphical diagnostics from DART netCDF output files are available. DART users have also used ncview to create rudimentary graphical displays of output data fields. The NCO tools, produced by UCAR's Unidata group, are available to do operations like concatenating, slicing, and dicing of netCDF files.

6.20.3 Requirements: an F90 compiler

The DART software has been successfully built on several Linux/x86 platforms with the Intel Fortran Compiler 7.1 for Linux, which is free for individual scientific use. It has also been built and successfully run with the Portland Group Fortran Compiler (5.02), and again with the Intel 8.0.034 compiler. Since recompiling the code is a necessity to experiment with different models, there are no binaries to distribute.

DART uses the netCDF self-describing data format for the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. In particular, DART also makes use of the F90 interface to the library which are available through the `netcdf.mod` and `typesizes.mod` modules. *IMPORTANT*: different compilers create these modules with different "case" filenames, and sometimes they are not **both** installed into the expected directory. It is required that both modules be present. The normal place would be in the `netcdf/include` directory, as opposed to the `netcdf/lib` directory.

If the netCDF library does not exist on your system, you must build it (as well as the F90 interface modules). The library and instructions for building the library or installing from an RPM may be found at the netCDF home page: http://www.unidata.ucar.edu/packages/netcdf/ Pay particular attention to the compiler-specific patches that must be applied for the Intel Fortran Compiler. (Or the PG compiler, for that matter.)

The location of the netCDF library, `libnetcdf.a`, and the locations of both `netcdf.mod` and `typesizes.mod` will be needed by the makefile template, as described in the compiling section.

DART also uses the **very** common udunits library for manipulating units of physical quantities. If, somehow, it is not installed on your system, you will need to install it (instructions are available from Unidata's Downloads page).

The location of the udunits library, `libudunits.a`, will be needed by the makefile template, as described in the compiling section.

6.20.4 Unpacking the distribution

This release of the DART source code can be downloaded as a compressed zip or tar.gz file. When extracted, the source tree will begin with a directory named `DART` and will be approximately 14.2 Mb. Compiling the code in this tree (as is usually the case) will necessitate much more space.

```
$ gunzip DART-2.0.0.tar.gz
$ tar -xvf DART-2.0.0.tar
```

The code tree is very "bushy"; there are many directories of support routines, etc. but only a few directories involved with the customization and installation of the DART software. If you can compile and run ONE of the low-order models, you should be able to compile and run ANY of the low-order models. For this reason, we can focus on the Lorenz `63 model. Subsequently, the only directories with files to be modified to check the installation are: `DART/mkmf`, `DART/models/lorenz_63/work`, and `DART/matlab` (but only for analysis).

6.20.5 Customizing the build scripts – overview

DART executable programs are constructed using two tools: make and mkmf. The make utility is a relatively common piece of software that requires a user-defined input file that records dependencies between different source files. make then performs a hierarchy of actions when one or more of the source files is modified. The mkmf utility is a custom preprocessor that generates a make input file (named Makefile) and an example namelist input.nml.mkmf with the default values. The Makefile is designed specifically to work with object-oriented Fortran90 (and other languages) for systems like DART.

mkmf requires two separate input files. The first is a \`template' file which specifies details of the commands required for a specific Fortran90 compiler and may also contain pointers to directories containing pre-compiled utilities required by the DART system. **This template file will need to be modified to reflect your system**. The second input file is a \`path_names' file which includes a complete list of the locations (either relative or absolute) of all Fortran90 source files that are required to produce a particular DART program. Each 'path_names' file must contain a path for exactly one Fortran90 file containing a main program, but may contain any number of additional paths pointing to files containing Fortran90 modules. An mkmf command is executed which uses the 'path_names' file and the mkmf template file to produce a Makefile which is subsequently used by the standard make utility.

Shell scripts that execute the mkmf command for all standard DART executables are provided as part of the standard DART software. For more information on mkmf see the FMS mkmf description.

One of the benefits of using mkmf is that it also creates an example namelist file for each program. The example namelist is called input.nml.mkmf, so as not to clash with any exising input.nml that may exist in that directory.

A series of templates for different compilers/architectures exists in the DART/mkmf/ directory and have names with extensions that identify either the compiler, the architecture, or both. This is how you inform the build process of the specifics of your system. Our intent is that you copy one that is similar to your system into mkmf.template and customize it. For the discussion that follows, knowledge of the contents of one of these templates (i.e. mkmf.template.pgi) is needed: (note that only the first few lines are shown here)

```
# Makefile template for PGI f90
FC = pgf90
CPPFLAGS =
FFLAGS = -r8 -Ktrap=fp -pc 64 -I/usr/local/netcdf/include
LD = pgf90
LDFLAGS = $(LIBS)
LIBS = -L/usr/local/netcdf/lib -lnetcdf
-L/usr/local/udunits-1.11.7/lib -ludunits
LIST = -Mlist
# you should never need to change any lines below.
...
```

Essentially, each of the lines defines some part of the resulting Makefile. Since make is particularly good at sorting out dependencies, the order of these lines really doesn't make any difference. The FC = pgf90 line ultimately defines the Fortran90 compiler to use, etc. The lines which are most likely to need site-specific changes start with FFLAGS and LIBS, which indicate where to look for the netCDF F90 modules and the location of the netCDF and udunits libraries.

FFLAGS

Each compiler has different compile flags, so there is really no way to exhaustively cover this other than to say the templates as we supply them should work – depending on the location of the netCDF modules netcdf.mod and typesizes.mod. Change the /usr/local/netcdf/include string to reflect the location of your modules. The low-order models can be compiled without the -r8 switch, but the bgrid_solo model cannot.

Libs

Modifying the LIBS value should be relatively straightforward.
Change the /usr/local/netcdf/lib string to reflect the location of your libnetcdf.a.
Change the /usr/local/udunits-1.11.7/lib string to reflect the location of your libudunits.a.

Customizing the 'path_names_*' file

Several path_names_* files are provided in the work directory for each specific model, in this case: DART/models/lorenz_63/work.

1. path_names_create_obs_set_def
2. path_names_create_obs_sequence
3. path_names_perfect_model_obs
4. path_names_filter

Since each model comes with its own set of files, no further customization is needed.

6.20.6 Building the Lorenz_63 DART project

Currently, DART executables are constructed in a work subdirectory under the directory containing code for the given model. In the top-level DART directory, change to the L63 work directory and list the contents:

```
$ cd DART/models/lorenz_63/work
$ ls -1
```

With the result:

```
filter_ics
mkmf_create_obs_sequence
mkmf_create_obs_set_def
mkmf_filter
mkmf_perfect_model_obs
path_names_create_obs_sequence
path_names_create_obs_set_def
path_names_filter
path_names_perfect_model_obs
perfect_ics
```

There are four mkmf_*xxxxxx* files for the programs create_obs_set_def, create_obs_sequence, perfect_model_obs, and filter along with the corresponding path_names_*xxxxxx* files. You can examine the contents of one of the path_names_*xxxxxx* files, for instance path_names_filter, to see a list of the relative paths of all files that contain Fortran90 modules required for the program filter for the L63 model. All of these paths are relative to your DART directory. The first path is the main program (filter.f90) and is followed by all the Fortran90 modules used by this program.

The mkmf_*xxxxxx* scripts are cryptic but should not need to be modified – as long as you do not restructure the code tree (by moving directories, for example). The only function of the mkmf_*xxxxxx* script is to generate a Makefile and an input.nml.mkmf file. It is not supposed to compile anything:

```
$ csh mkmf_create_obs_set_def
$ mv input.nml.mkmf input.nml.create_obs_set_def
$ make
```

The first command generates an appropriate Makefile and the input.nml.mkmf file. The second saves the example namelist to a unique name (the next DART release will do this automatically – no harm is done by omitting this step) and the last command results in the compilation of a series of Fortran90 modules which ultimately produces an executable file: create_obs_set_def. Should you need to make any changes to the DART/mkmf/mkmf.template, you will need to regenerate the Makefile. A series of object files for each module compiled will also be left in the work directory, as some of these are undoubtedly needed by the build of the other DART components. You can proceed to create the other three programs needed to work with L63 in DART as follows:

```
$ csh mkmf_create_obs_sequence
$ mv input.nml.mkmf input.nml.create_obs_sequence
$ make
$ csh mkmf_perfect_model_obs
$ mv input.nml.mkmf input.nml.perfect_model_obs
$ make
$ csh mkmf_filter
$ mv input.nml.mkmf input.nml.filter
$ make
```

The result (hopefully) is that four executables now reside in your work directory. The most common problem is that the netCDF libraries and include files (particularly typesizes.mod) are not found. Edit the DART/mkmf/mkmf.template, recreate the Makefile, and try again.

program	purpose
create_obs_set_def	specify a (set) of observation characteristics taken by a particular (set of) instruments
create_obs_sequence	specify the temporal attributes of the observation sets
perfect_model_obs	spinup, generate "true state" for synthetic observation experiments, . . .
filter	perform experiments

6.20.7 Running Lorenz_63

This initial sequence of exercises includes detailed instructions on how to work with the DART code and allows investigation of the basic features of one of the most famous dynamical systems, the 3-variable Lorenz-63 model. The remarkable complexity of this simple model will also be used as a case study to introduce a number of features of a simple ensemble filter data assimilation system. To perform a synthetic observation assimilation experiment for the L63 model, the following steps must be performed (an overview of the process is given first, followed by detailed procedures for each step):

6.20.8 Experiment overview

1. Integrate the L63 model for a long time starting from arbitrary initial conditions to generate a model state that lies on the attractor. The ergodic nature of the L63 system means a 'lengthy' integration always converges to some point on the computer's finite precision representation of the model's attractor.
2. Generate a set of ensemble initial conditions from which to start an assimilation. Since L63 is ergodic, the ensemble members can be designed to look like random samples from the model's 'climatological distribution'. To generate an ensemble member, very small perturbations can be introduced to the state on the attractor generated by step 1. This perturbed state can then be integrated for a very long time until all memory of its initial condition can be viewed as forgotten. Any number of ensemble initial conditions can be generated by repeating this procedure.
3. Simulate a particular observing system by first creating an 'observation set definition' and then creating an 'observation sequence'. The 'observation set definition' describes the instrumental characteristics of the observations and the 'observation sequence' defines the temporal sequence of the observations.
4. Populate the 'observation sequence' with 'perfect' observations by integrating the model and using the information in the 'observation sequence' file to create simulated observations. This entails operating on the model state at the time of the observation with an appropriate forward operator (a function that operates on the model state vector to produce the expected value of the particular observation) and then adding a random sample from the observation error distribution specified in the observation set definition. At the same time, diagnostic output about the 'true' state trajectory can be created.
5. Assimilate the synthetic observations by running the filter; diagnostic output is generated.

1. Integrate the L63 model for a 'long' time

`perfect_model_obs` integrates the model for all the times specified in the 'observation sequence definition' file. To this end, begin by creating an 'observation sequence definition' file that spans a long time. Creating an 'observation sequence definition' file is a two-step procedure involving `create_obs_set_def` followed by `create_obs_sequence`. After they are both run, it is necessary to integrate the model with `perfect_model_obs`.

1.1 Create an observation set definition

`create_obs_set_def` creates an observation set definition, the time-independent part of an observation sequence. An observation set definition file only contains the `location, type,` and `observational error characteristics` (normally just the diagonal observational error variance) for a related set of observations. There are no actual observations, nor are there any times associated with the definition. For spin-up, we are only interested in integrating the L63 model, not in generating any particular synthetic observations. Begin by creating a minimal observation set definition.

In general, for the low-order models, only a single observation set need be defined. Next, the number of individual scalar observations (like a single surface pressure observation) in the set is needed. To spin-up an initial condition for the L63 model, only a single observation is needed. Next, the error variance for this observation must be entered. Since we do not need (nor want) this observation to have any impact on an assimilation (it will only be used for spinning up the model and the ensemble), enter a very large value for the error variance. An observation with a very large error variance has essentially no impact on deterministic filter assimilations like the default variety implemented in DART. Finally, the location and type of the observation need to be defined. For all types of models, the most elementary form of synthetic observations are called 'identity' observations. These observations are generated simply by adding a random sample from a specified observational error distribution directly to the value of one of the state variables. This defines the observation as being an identity observation of the first state variable in the L63 model. The program will respond by terminating after generating a file (generally named `set_def.out`) that defines the single identity observation of the first state variable of the L63 model. The following is a screenshot (much of the verbose logging has been left off for clarity), the user input looks *like this*.

```
[unixprompt]$ ./create_obs_set_def
 Initializing the utilities module.
 Registering module :
 $Source$
 $Revision$
 $Date$
 Registration complete.

 &UTILITIES_NML
 TERMLEVEL =            2,
 LOGFILENAME = dart_log.out
 /

 Registering module :
 $Source$
 $Revision$
 $Date$
 Registration complete.

 Input the filename for output of observation set_def_list? [set_def.out]
set_def.out

{ ... }

 Input the number of unique observation sets you might define
1
 How many observations in set             1
1
 Defining observation             1
 Input error variance for this observation definition
1000000
 Input an integer index if this is identity observation, else -1
1

 Registering module :
 $Source$
 $Revision$
 $Date$
 Registration complete.

 set_def.out successfully created.
 Terminating normally.
```

1.2 Create an observation sequence definition

`create_obs_sequence` creates an 'observation sequence definition' by extending the 'observation set definition' with the temporal attributes of the observations.

The first input is the name of the file created in the previous step, i.e. the name of the observation set definition that you've just created. It is possible to create sequences in which the observation sets are observed at regular intervals or irregularly in time. Here, all we need is a sequence that takes observations over a long period of time - indicated by entering a 1. Although the L63 system normally is defined as having a non-dimensional time step, the DART system arbitrarily defines the model timestep as being 3600 seconds. By declaring we have 1000 observations taken once per day, we create an observation sequence definition spanning 24000 'model' timesteps; sufficient to spin-up the model onto the attractor. Finally, enter a name for the 'observation sequence definition' file. Note again: there are no observation values present in this file. Just an observation type, location, time and the error characteristics. We are going to populate the observation sequence with the `perfect_model_obs` program.

```
[thoar@ghotiol work]$ ./create_obs_sequence
 Registering module :
 $Source$
 $Revision$
 $Date$
 Registration complete.

 &UTILITIES_NML
 TERMLEVEL =            2,
 LOGFILENAME = dart_log.out
 /

 Registering module :
 $Source$
 $Revision$
 $Date$
 Registration complete.

 What is name of set_def_list? [set_def.out]
set_def.out

 { ... }

 Setting times for obs_def             1
 To input a regularly repeating time sequence enter 1
 To enter an irregular list of times enter 2
1
 Input number of observations in sequence
1000
 Input time of initial ob in sequence in days and seconds
1, 0
 Input period of obs in days and seconds
1, 0
 time             1  is             0            1
 time             2  is             0            2
 time             3  is             0            3
...
 time           998  is             0          998
 time           999  is             0          999
 time          1000  is             0         1000
 Input file name for output of obs_sequence? [obs_seq.in]
```

(continues on next page)

(continued from previous page)

```
obs_seq.in
```

1.3 Initialize the model onto the attractor

`perfect_model_obs` can now advance the arbitrary initial state for 24,000 timesteps to move it onto the attractor.

`perfect_model_obs` uses the Fortran90 namelist input mechanism instead of (admittedly gory, but temporary) interactive input. All of the DART software expects the namelists to found in a file called `input.nml`. When you built the executable, an example namelist was created `input.nml.mkmf` that contains all of the namelist input for the executable. If you followed the example, each namelist was saved to a unique name. We must now rename and edit the namelist file for `perfect_model_obs`. Copy `input.nml.perfect_model_obs` to `input.nml` and edit it to look like the following:

```
&perfect_model_obs_nml
   async = 0,
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .false.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1
&end
&assim_tools_nml
   prior_spread_correction = .false.,
   filter_kind = 1,
   slope_threshold = 1.0
&end
&cov_cutoff_nml
   select_localization = 1
&end
&assim_model_nml
   binary_restart_files = .true.
&end
&model_nml
   sigma = 10.0,
   r = 28.0,
   b = 2.6666666666667,
   deltat = 0.01
&end
&utilities_nml
   TERMLEVEL = 1
   logfilename = 'dart_log.out'
&end
```

For the moment, only two namelists warrant explanation. Each namelists is covered in detail in the html files accompanying the source code for the module.

perfect_model_obs_nml

namelist variable	description
`async`	For the lorenz_63, simply ignore this. Leave it set to '0'
`obs_seq_in_file_name`	specifies the file name that results from running `create_obs_sequence`, i.e. the 'observation sequence definition' file.
`obs_seq_out_file_name`	specifies the output file name containing the 'observation sequence', finally populated with (perfect?) 'observations'.
`start_from_restart`	When set to 'false', `perfect_model_obs` generates an arbitrary initial condition (which cannot be guaranteed to be on the L63 attractor).
`output_restart`	When set to 'true', `perfect_model_obs` will record the model state at the end of this integration in the file named by `restart_out_file_name`.
`restart_in_file_name`	is ignored when 'start_from_restart' is 'false'.
`restart_out_file_name`	if `output_restart` is 'true', this specifies the name of the file containing the model state at the end of the integration.
`init_time_`*xxxx*	the start time of the integration.
`output_interval`	interval at which to save the model state.

utilities_nml

namelist variable	description
`TERMLEVEL`	When set to '1' the programs terminate when a 'warning' is generated. When set to '2' the programs terminate only with 'fatal' errors.
`logfilename`	Run-time diagnostics are saved to this file. This namelist is used by all programs, so the file is opened in APPEND mode. Subsequent executions cause this file to grow.

Executing `perfect_model_obs` will integrate the model 24,000 steps and output the resulting state in the file `perfect_restart`. Interested parties can check the spinup in the `True_State.nc` file.

```
$ perfect_model_obs
```

2. Generate a set of ensemble initial conditions

The set of initial conditions for a 'perfect model' experiment is created by taking the spun-up state of the model (available in `perfect_restart`), running `perfect_model_obs` to generate the 'true state' of the experiment and a corresponding set of observations, and then feeding the same initial spun-up state and resulting observations into `filter`.

Generating ensemble initial conditions is achieved by changing a perfect_model_obs namelist parameter, copying `perfect_restart` to `perfect_ics`, and rerunning `perfect_model_obs`. This execution of `perfect_model_obs` will advance the model state from the end of the first 24,000 steps to the end of an additional 24,000 steps and place the final state in `perfect_restart`. The rest of the namelists in `input.nml` should remain unchanged.

```
&perfect_model_obs_nml
   async = 0,
   obs_seq_in_file_name = "obs_seq.in",
```

(continues on next page)

(continued from previous page)

```
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .true.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1 /
```

Then run:

```
$ cp perfect_restart perfect_ics
$ perfect_model_obs
```

A True_State.nc file is also created. It contains the 'true' state of the integration.

Generating the ensemble

is done with the program filter, which also uses the Fortran90 namelist mechanism for input. It is now necessary to copy the input.nml.filter namelist to input.nml or you may simply insert the filter_nml namelist into the existing input.nml. Having the perfect_model_obs namelist in the input.nml does not hurt anything. In fact, I generally create a single input.nml that has all the namelist blocks in it.

```
&perfect_model_obs_nml
   async = 0,
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .true.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1 /
&assim_tools_nml
   prior_spread_correction = .false.,
   filter_kind = 1,
   slope_threshold = 1.0 /
&cov_cutoff_nml
   select_localization = 1 /
&assim_model_nml
   binary_restart_files = .true. /
&model_nml
   sigma = 10.0,
   r = 28.0,
   b = 2.6666666666667
   deltat = 0.01 /
&utilities_nml
   TERMLEVEL = 1
   logfilename = 'dart_log.out' /
&reg_factor_nml
   select_regression = 1,
   input_reg_file = "time_mean_reg" /
&filter_nml
   async = 0,
```

(continues on next page)

(continued from previous page)

```
   ens_size = 20,
   cutoff = 0.20,
   cov_inflate = 1.00,
   start_from_restart = .false.,
   output_restart = .true.,
   obs_sequence_file_name = "obs_seq.out",
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "filter_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_state_ens_mean = .true.,
   output_state_ens_spread = .true.,
   num_output_ens_members = 20,
   output_interval = 1,
   num_groups = 1,
   confidence_slope = 0.0,
   output_obs_diagnostics = .false.,
   get_mean_reg = .false.,
   get_median_reg = .false. /
```

Only the non-obvious(?) entries for `filter_nml` will be discussed.

namelist variable	description
`ens_size`	Number of ensemble members. 20 is sufficient for most of the L63 exercises.
`cutoff`	to limit the impact of an observation, set to 0.0 (i.e. spin-up)
`cov_inflate`	A value of 1.0 results in no inflation.(spin-up)
`start_from_restart`	when ‘.false.’, `filter` will generate its own set of initial conditions. It is important to note that the filter still makes use of `perfect_ics` by randomly perturbing these state variables.
`num_output_ens_members`	may be a value from 0 to `ens_size`
`output_state_ens_mean`	when ‘.true.’ the mean of all ensemble members is output.
`output_state_ens_spread`	when ‘.true.’ the spread of all ensemble members is output.
`output_interval`	seconds

The filter is told to generate its own ensemble initial conditions since `start_from_restart` is ‘.false.’. However, it is important to note that the filter still makes use of `perfect_ics` which is set to be the `restart_in_file_name`. This is the model state generated from the first 24,000 step model integration by `perfect_model_obs`. `Filter` generates its ensemble initial conditions by randomly perturbing the state variables of this state.

The arguments `output_state_ens_mean` and `output_state_ens_spread` are ‘.true.’ so that these quantities are output at every time for which there are observations (once a day here) and `num_output_ens_members` means that the same diagnostic files, `Posterior_Diag.nc` and `Prior_Diag.nc` also contain values for all 20 ensemble members once a day. Once the namelist is set, execute `filter` to integrate the ensemble forward for 24,000 steps with the final ensemble state written to the `filter_restart`. Copy the `perfect_model_obs` restart file `perfect_restart` (the `true state’) to `perfect_ics`, and the `filter` restart file `filter_restart` to `filter_ics` so that future assimilation experiments can be initialized from these spun-up states.

```
$ filter
$ cp perfect_restart perfect_ics
$ cp filter_restart filter_ics
```

The spin-up of the ensemble can be viewed by examining the output in the netCDF files `True_State.nc` generated by `perfect_model_obs` and `Posterior_Diag.nc` and `Prior_Diag.nc` generated by `filter`. To do this, see the detailed discussion of matlab diagnostics in Appendix I.

3. Simulate a particular observing system

Begin by using `create_obs_set_def` to generate an observation set in which each of the 3 state variables of L63 is observed with an observational error variance of 1.0 for each observation. To do this, use the following input sequence (the text including and after # is a comment and does not need to be entered):

set_def.out	# Output file name
1	# Number of sets
3	# Number of observations in set (x, y, and z)
1.0	# Variance of first observation
1	# First ob is identity observation of state variable 1 (x)
1.0	# Variance of second observation
2	# Second is identity observation of state variable 2 (y)
1.0	# Variance of third ob
3	# Identity ob of third state variable (z)

Now, generate an observation sequence definition by running `create_obs_sequence` with the following input sequence:

set_def.out	# Input observation set definition file
1	# Regular spaced observation interval in time
1000	# 1000 observation times
0, 43200	# First observation after 12 hours (0 days, 3600 * 12 seconds)
0, 43200	# Observations every 12 hours
obs_seq.in	# Output file for observation sequence definition

4. Generate a particular observing system and true state

An observation sequence file is now generated by running `perfect_model_obs` with the namelist values (unchanged from step 2):

```
&perfect_model_obs_nml
   async = 0,
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .true.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1 /
```

This integrates the model starting from the state in `perfect_ics` for 1000 12-hour intervals outputting synthetic observations of the three state variables every 12 hours and producing a netCDF diagnostic file, `True_State.nc`.

5. Filtering

Finally, `filter` can be run with its namelist set to:

```
&filter_nml
   async = 0,
   ens_size = 20,
   cutoff = 22222222.0,
   cov_inflate = 1.00,
   start_from_restart = .true.,
   output_restart = .true.,
   obs_sequence_file_name = "obs_seq.out",
   restart_in_file_name = "filter_ics",
   restart_out_file_name = "filter_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_state_ens_mean = .true.,
   output_state_ens_spread = .true.,
   num_output_ens_members = 20,
   output_interval = 1,
   num_groups = 1,
   confidence_slope = 0.0,
   output_obs_diagnostics = .false.,
   get_mean_reg = .false.,
   get_median_reg = .false. /
```

The large value for the cutoff allows each observation to impact all other state variables (see Appendix V for localization). `filter` produces two output diagnostic files, `Prior_Diag.nc` which contains values of the ensemble members, ensemble mean and ensemble spread for 12- hour lead forecasts before assimilation is applied and `Posterior_Diag.nc` which contains similar data for after the assimilation is applied (sometimes referred to as analysis values).

Now try applying all of the matlab diagnostic functions described in the Matlab Diagnostics section.

6.20.9 Matlab® diagnostics

The output files are netCDF files, and may be examined with many different software packages. We happen to use Matlab, and provide our diagnostic scripts in the hopes that they are useful.

The Matlab diagnostic scripts and underlying functions reside in the `DART/matlab` directory. They are reliant on the public-domain netcdf toolbox from `http://woodshole.er.usgs.gov/staffpages/cdenham/public_html/MexCDF/nc4ml5.html` as well as the public-domain CSIRO matlab/netCDF interface from `http://www.marine.csiro.au/sw/matlab-netcdf.html`. If you do not have them installed on your system and want to use Matlab to peruse netCDF, you must follow their installation instructions.

Once you can access the `getnc` function from within Matlab, you can use our diagnostic scripts. It is necessary to prepend the location of the DART/matlab scripts to the matlabpath. Keep in mind the location of the netcdf operators on your system WILL be different from ours … and that's OK.

```
0[269]0 ghotiol:/<5>models/lorenz_63/work]$ matlab -nojvm

                                             < M A T L A B >
                                 Copyright 1984-2002 The MathWorks, Inc.
                                     Version 6.5.0.180913a Release 13
                                               Jun 18 2002

```

(continues on next page)

(continued from previous page)

```
  Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.

  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

>> which getnc
/contrib/matlab/matlab_netcdf_5_0/getnc.m
>>ls *.nc

ans =

Posterior_Diag.nc  Prior_Diag.nc  True_State.nc


>>path('../../../matlab',path)
>>which plot_ens_err_spread
../../../matlab/plot_ens_err_spread.m
>>help plot_ens_err_spread

  DART : Plots summary plots of the ensemble error and ensemble spread.
                         Interactively queries for the needed information.
                         Since different models potentially need different
                         pieces of information ... the model types are
                         determined and additional user input may be queried.

  Ultimately, plot_ens_err_spread will be replaced by a GUI.
  All the heavy lifting is done by PlotEnsErrSpread.

  Example 1 (for low-order models)

  truth_file = 'True_State.nc';
  diagn_file = 'Prior_Diag.nc';
  plot_ens_err_spread

>>plot_ens_err_spread
```

And the matlab graphics window will display the spread of the ensemble error for each state variable. The scripts are designed to do the “obvious” thing for the low-order models and will prompt for additional information if needed. The philosophy of these is that anything that starts with a lower-case *plot_some_specific_task* is intended to be user-callable and should handle any of the models. All the other routines in `DART/matlab` are called BY the high-level routines.

Matlab script	description
`plot_bins`	plots ensemble rank histograms
`plot_correl`	Plots space-time series of correlation between a given variable at a given time and other variables at all times in a n ensemble time sequence.
`plot_ens_err_spread`	Plots summary plots of the ensemble error and ensemble spread. Interactively queries for the needed information. Since different models potentially need different pieces of information ... the model types are determined and additional user input may be queried.
`plot_ens_mean_time_series`	Queries for the state variables to plot.
`plot_ens_time_series`	Queries for the state variables to plot.
`plot_phase_space`	Plots a 3D trajectory of (3 state variables of) a single ensemble member. Additional trajectories may be superimposed.
`plot_total_err`	Summary plots of global error and spread.
`plot_var_var_correl`	Plots time series of correlation between a given variable at a given time and another variable at all times in an ensemble time sequence.

6.20.10 Bias, filter divergence and covariance inflation (with the l63 model)

One of the common problems with ensemble filters is filter divergence, which can also be an issue with a variety of other flavors of filters including the classical Kalman filter. In filter divergence, the prior estimate of the model state becomes too confident, either by chance or because of errors in the forecast model, the observational error characteristics, or approximations in the filter itself. If the filter is inappropriately confident that its prior estimate is correct, it will then tend to give less weight to observations than they should be given. The result can be enhanced overconfidence in the model's state estimate. In severe cases, this can spiral out of control and the ensemble can wander entirely away from the truth, confident that it is correct in its estimate. In less severe cases, the ensemble estimates may not diverge entirely from the truth but may still be too confident in their estimate. The result is that the truth ends up being farther away from the filter estimates than the spread of the filter ensemble would estimate. This type of behavior is commonly detected using rank histograms (also known as Talagrand diagrams). You can see the rank histograms for the L63 initial assimilation by using the matlab script `plot_bins`.

A simple, but surprisingly effective way of dealing with filter divergence is known as covariance inflation. In this method, the prior ensemble estimate of the state is expanded around its mean by a constant factor, effectively increasing the prior estimate of uncertainty while leaving the prior mean estimate unchanged. The program `filter` has a namelist parameter that controls the application of covariance inflation, `cov_inflate`. Up to this point, `cov_inflate` has been set to 1.0 indicating that the prior ensemble is left unchanged. Increasing `cov_inflate` to values greater than 1.0 inflates the ensemble before assimilating observations at each time they are available. Values smaller than 1.0 contract (reduce the spread) of prior ensembles before assimilating.

You can do this by modifying the value of `cov_inflate` in the namelist, (try 1.05 and 1.10 and other values at your discretion) and run the filter as above. In each case, use the diagnostic matlab tools to examine the resulting changes to the error, the ensemble spread (via rank histogram bins, too), etc. What kind of relation between spread and error is seen in this model?

6.20.11 Synthetic observations

Synthetic observations are generated from a `perfect’ model integration, which is often referred to as the `truth’ or a `nature run’. A model is integrated forward from some set of initial conditions and observations are generated as $y = H(x) + e$ where H is an operator on the model state vector, x, that gives the expected value of a set of observations, y, and e is a random variable with a distribution describing the error characteristics of the observing instrument(s) being simulated. Using synthetic observations in this way allows students to learn about assimilation algorithms while being isolated from the additional (extreme) complexity associated with model error and unknown observational error characteristics. In other words, for the real-world assimilation problem, the model has (often substantial) differences from what happens in the real system and the observational error distribution may be very complicated and is certainly not well known. Be careful to keep these issues in mind while exploring the capabilities of the ensemble filters with synthetic observations.

6.21 9-variable

6.21.1 Overview

The 9-variable model is described in Lorenz (1980).[1] Lorenz developed this primitive-equation model using shallow-water equations as a starting point and manipulating the divergence equations so that the model exhibits quasi-geostrophic behavior and transient gravity waves that dissipate with time. Gent and McWilliams (1982)[2] explore the behavior of this model extensively. For an introduction to shallow-water equations, we recommend consulting the relevant section of a meteorology textbook such as section 4.5 of Holton and Hakim (2013).[3]

The model’s three X variables are at 0, 1/9, and 2/9, three Y variables are at 3/9, 4/9 and 5/9, and three Z variables are at 6/9, 7/9, and 8/9 on a cyclic [0, 1] domain.

In the 9-variable model, DART advances the model, gets the model state and metadata describing this state. The model can be configured by altering the `&model_nml` *namelist* in the `input.nml` file. The details of the `&model_nml` namelist are always model-specific (there are no generic namelist values). The model time step defaults to 1 hour (3600 seconds) but is settable by altering the namelist.

The 9-variable model has a `work/workshop_setup.csh` script that compiles and runs an example. This example is referenced in Sections 7 and 10 of the DART_tutorial and is intended to provide insight into model/assimilation behavior. The example **may or may not** result in good (*or even decent!*) results!

6.21.2 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   g                  = 8.0,
   deltat             = 0.0833333333333333,
   time_step_days     = 0,
   time_step_seconds  = 3600
/
```

[1] Lorenz, Edward N., 1980: Attractor Sets and Quasi-Geostrophic Equilibrium. *Journal of the Atmospheric Sciences*, **37**, 1685-1699. doi:10.1175/1520-0469(1980)037<1685:ASAQGE>2.0.CO;2

[2] Gent, Peter R., and James C. McWilliams, 1982: Intermediate Model Solutions to the Lorenz Equations: Strange Attractors and Other Phenomena. *Journal of the Atmospheric Sciences*, **39**, 3-13. doi:10.1175/1520-0469(1982)039<0003:IMSTTL>2.0.CO;2

[3] Holton, James R., and Gregory J. Hakim, 2013: *An Introduction to Dynamic Meteorology – Fifth Edition.* Academic Press, 532 pp.

Description of each namelist entry

Item	Type	Description
g	real(r8)	Model parameter, see comp_dt in code for equations.
delta_t	real(r8)	Non-dimensional timestep. This is mapped to the dimensional timestep specified by time_step_days and time_step_seconds.
time_step_days	real(r8)	Number of days for dimensional timestep, mapped to delta_t.
time_step_seconds	real(r8)	Number of seconds for dimensional timestep, mapped to delta_t.

6.21.3 References

6.22 AM2

6.22.1 Overview

AM2 is an atmospheric model developed as part of a coupled atmosphere-ocean general circulation system developed at NOAA's Geophysical Fluid Dynamics Laboratory.

If you are interested in running DART with this model please contact the DART group at dart@ucar.edu for more information.

6.23 bgrid_solo

6.23.1 Overview

DART interface module for the dynamical core of the GFDL AM2 Bgrid model. This model is subroutine callable from DART and can be run in a similar fashion to low-order models that produce diagnostic output files with multiple assimilation times per file.

The Bgrid model was originally configured as a comprehensive atmospheric model as described in Anderson et al. (2004).[1]

All of that code remains in the directories under the `DARTHOME/models/bgrid_solo` directory, however, much of the capability has been disabled by code modification. What is left is a dry dynamical core for a model with no dirunal cycle at equinox with forcing described in Held and Suarez (1994).[2]

The default settings are for a model with a 60x30 horizontal grid and 5 vertical levels. This is close to the smallest version that has somewhat realistic baroclinic instability resulting in mid-latitude 'storm tracks'. The model resolution can be changed with the entries in the `bgrid_cold_start_nml` namelist described in the *Namelist* section. It may be necessary to change the model time step to maintain stability for larger model grids. The model state variables are the gridded surface pressure, temperature, and u and v wind components.

The `bgrid_solo` directory has a `work/workshop_setup.csh` script that compiles and runs an example. This example is intended to demonstrate that the same process used for a low-order model may be used for a much more complex model and generates output for state-space or observation-space diagnostics.

[1] Anderson, J. L. and Coauthors, 2004: The new GFDL global atmosphere and land model AM2-LM2: Evaluation with prescribed SST simulations. *Journal of Climate*, **17**, 4641-4673. doi:10.1175/JCLI-3223.1

[2] Held, I. M., and M. J. Suarez, 1994: A proposal for the intercomparison of the dynamical cores of atmospheric general circulation models, *Bulletin of the American Meteorological Society*, **75(10)**, 1825-1830. doi:10.1175/1520-0477(1994)075<1825:APFTIO>2.0.CO;2

Some examples of ways in which this model can be configured and modified to test DART assimilation capabilities are documented in Anderson et al. (2005).[3]

Several programs that generate interesting observation sequences are available in the DARTHOME/models/bgrid_solo directory. These programs take interactive user input and create a text file that can be piped into program create_obs_sequence to create obs_sequence files. These can serve as examples for users who are interested in designing their own custom obs_sequence files.

Program column_rand creates an obs_sequence with randomly located columns of observations (essentially synthetic radiosondes) that observe surface pressure along with temperature and wind components at all model levels.

Program id_set_def_stdin generates an obs_sequence file that observes every state variable with error variance of 10000 for surface pressure and 1.0 for temperature and wind components.

Program ps_id_stdin generates an obs_sequence that observes every surface pressure variable for the default model size (30x60) with an error variance of 100.

Program ps_rand_local generates a set of randomly located surface pressure observations with an interactively specified error variance. It also allows the observations to be confined to a rectangular subdomain.

6.23.2 Namelist

The &model_nml namelist is read from the input.nml file. Namelists start with an ampersand & and terminate with a slash /. Character strings that contain a / must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   current_time =  0, 0, 0, 0
   override = .false.,
   dt_atmos = 3600,
   days     = 10,
   hours    = 0,
   minutes  = 0,
   seconds  = 0,
   noise_sd = 0.0,
   dt_bias  = -1,
   state_variables = 'ps', 'QTY_SURFACE_PRESSURE',
                     't',  'QTY_TEMPERATURE',
                     'u',  'QTY_U_WIND_COMPONENT',
                     'v',  'QTY_V_WIND_COMPONENT',
   template_file = 'perfect_input.nc'
/
# only used if initial conditions file not specified in run
&bgrid_cold_start_nml
   nlon = 60,
   nlat = 30,
   nlev = 5,
   equal_vert_spacing = .true.
/
# Values in hs_forcing_nml are described in Held and Suarez (1994)
&hs_forcing_nml
   delh      =  60.,
   t_zero    = 315.,
   t_strat   = 200.,
   delv      =  10.,
```

(continues on next page)

[3] Anderson, J. L., Wyman, B., Zhang, S. & Hoar, T., 2005: Assimilation of surface pressure observations using an ensemble filter in an idealized global atmospheric prediction system, *Journal of the Atmospheric Sciences*, **62**, 2925-2938. doi:10.1175/JAS3510.1

(continued from previous page)

```
   eps        =   0.,
   ka         = -40.,
   ks         =  -4.,
   kf         =  -1.,
   sigma_b    =  .7,
   do_conserve_energy = .false.
/
&bgrid_core_driver_nml
   damp_coeff_wind   = 0.10,
   damp_coeff_temp   = 0.10,
   damp_coeff_tracer = 0.10,
   advec_order_wind   = 4,
       advec_order_temp   = 2,
       advec_order_tracer = 2,
       num_sponge_levels = 1,
       sponge_coeff_wind   = 1.00,
       sponge_coeff_temp   = 1.00,
       sponge_coeff_tracer = 1.00,
       num_fill_pass = 2,
       decomp = 0,0,
       num_adjust_dt = 3,
       num_advec_dt  = 3,
       halo = 1,
       do_conserve_energy = .false.
/
&bgrid_integrals_nml
   file_name  = 'dynam_integral.out',
   time_units = 'days',
   output_interval = 1.00
/
```

Description of each namelist entry

The following values are specified in `model_nml`.

Item	Type	Description
current_time	integer(4)	Specifies the initial time of the Bgrid model internal clock. The four integer values are the day, hour, minutes, and seconds. The default version of the Bgrid model has neither a diurnal or seasonal cycle, so these can all be set to 0, the default value.
override	logical	If true, then the initial model date is taken from namelist entry current_time, even if an atmos_model.res file is found in directory INPUT. For most DART applications, atmospheric restart values are coming from DART files and no INPUT directory is used.
dt_atmos	integer	Model timestep in seconds.
noise_sd	real(r8)	Standard deviation of random perturbations to the time tendency of temperature applied at each timestep. Each gridpoint value of the computed temperature tendency is multiplied by 1+N(0, noise_sd) before the updated values of temperature are computed.
dt_bias	integer	Allows a simple mechanism to simulate model error. If dt_bias is non-zero, the assimilation programs believe that each model advance changes the time by dt_bias. However, internally the bgrid model is moving things forward by dt_atmos. By running perfect_model_obs with one time step for the internal bgrid clock (for instance dt_atmos = 3600, dt_bias = 3600), and filter with another (dt_atmos = 3000, and dt_bias = 3600) model error is simulated.
state_variables	character(len=129)(:,2)	Strings that identify the bgrid_solo variables that should be part of the DART state vector. The first column is the netCDF variable name, the second column is the corresponding DART quantity.
template_file	character(len=256)	This is the name of the file that specifies the resolution of the variables DART uses to create the DART state vector. If *template_file = "null"* the *&bgrid_cold_start_nml* namelist variables are used to specify the resolution. The actual input filenames for *filter* and *perfect_model_obs* come from their respective namelists. The resolutions in the file specified in *template_file* must match the resolutions of the variables in the input filenames. To start an experiment with a new model resolution, set template_file to "null" and set the resolutions in bgrid_cold_start_nml.

The following values are specified in `bgrid_cold_start_nml`.

Item	Type	Description
nlon	integer	The number of longitudes on the model grid.
nlat	integer	The number of latitudes on the model grid.
nlev	integer	The number of model levels.
equal_vertical_spacing	logical	Model levels are equally spaced in pressure if true.

The Held-Suarez forcing details can be modified with the `hs_forcing_nml` namelist using the documentation in Held and Suarez (1994).

Model dynamics can be adjusted with the bgrid_core_driver_nml following the documentation in the references and internal documentation in the bgrid code.

6.23.3 References

6.24 CAM-FV

6.24.1 Overview

The DART system supports data assimilation into the Community Atmosphere Model (CAM) which is the atmospheric component of the Community Earth System Model (CESM). This DART interface is being used by graduate students, post-graduates, and scientists at universities and research labs to conduct data assimilation reseearch. Others are using the products of data assimilation (analyses), which were produced here at NCAR using CESM+DART, to conduct related research. The variety of research can be sampled on the DART Publications page.

"CAM" refers to a family of related atmospheric components, which can be built with two independent main characteristics. CESM labels these as:

resolution

where *resolution* refers to both the horizontal resolution of the grid (rather than the vertical resolution) **and** the dynamical core run on the specified grid. The dynamical core refers to the fluid dynamical equations run on the specified grid.

compset

where *compset* refers to the vertical grid **and** the parameterizations – the formulation of the subgrid-scale physics. These parameterizations encompass the equations describing physical processes such as convection, radiation, chemistry.

- The vertical grid is determined by the needs of the chosen parameterizations, thus the vertical spacing and the top level of the model domain, specified by a variable known as `ptop`, vary.
- The combinations of parameterizations and vertical grids are named: CAM3.5, CAM5, CAM#, . . . WACCM, WACCM#, WACCM-X, CAM-Chem.

Setup Variations describes the differences in the namelists, build scripts assimilation setup.

This DART+CAM interface has the following features.

- Assimilate within the CESM software framework by using the multi-instance capability of CESM1.1.1 (and later). This enables assimilation of suitable observations into multiple CESM components. The ability to assimilate in the previous mode, where DART called 'stand-alone' CAMs when needed, is not being actively supported for these CESM versions.
- Support for the finite-volume (FV) dynamical core.
- Use any resolution of CAM.
- Assimilate a variety of observations; to date the observations successfully assimilated include the NCEP reanalysis BUFR obs (T,U,V,Q), Global Positioning System radio occultation obs, and MOPITT carbon monoxide (when a chemistry model is incorporated into CAM-FV). Research has also explored assimilating surface observations, cloud liquid water, and aerosols. SABER and AURA observations have been assimilated into WACCM.
- Specify, via namelist entries, the CAM (initial file) variables which will be directly affected by the observations, that is, the state vector. This allows users to change the model state without recompiling (but other restrictions remain).
- Generate analyses on the CAM grid which have only CAM model error in them, rather than another model's.
- Generate such analyses with as few as 20 ensemble members.

In addition to the standard DART package there are ensembles of initial condition files at the large file website http://www.image.ucar.edu/pub/DART/CAM/ that are helpful for interfacing CAM with DART. In the current (2015) mode, CESM+DART can easily be started from a single model state, which is perturbed to create an ensemble of the desired size. A spin-up period is then required to allow the ensemble members to diverge.

Sample sets of observations, which can be used with CESM+DART assimilations, can be found at http://www.image.ucar.edu/pub/DART/Obs_sets/ of which the NCEP BUFR observations are the most widely used.

Reanalyses

There have been two large-scale reanalysis efforts using CAM and DART. The NCAR Research Archive dataset **"CAM6 Data Assimilation Research Testbed (DART) Reanalysis"** DS345.0 contains just under 120Tb (yes Tb) of data:

> These CAM6 Data Assimilation Research Testbed (DART) Reanalysis data products are designed to facilitate a broad variety of research using NCAR's CESM2 models, ranging from model evaluation to (ensemble) hindcasting, data assimilation experiments, and sensitivity studies. They come from an 80 member ensemble reanalysis of the global troposphere and stratosphere using DART and CAM6 from CESM2.1. The data products represent the actual states of the atmosphere during 2 recent decades at a 1 degree horizontal resolution and 6 hourly frequency. Each ensemble member is an equally likely description of the atmosphere, and is also consistent with dynamics and physics of CAM6.

The NCAR Research Archive dataset **"An Ensemble of Atmospheric Forcing Files from a CAM Reanalysis"** DS199.1 contains about 1.5Tb of data:

> This dataset contains files that are an ensemble of 'coupler history' files from an 80-member reanalysis performed with the Data Assimilation Research Testbed (DART) using the Community Atmosphere Model Version 4 with the finite volume core (CAM4 FV) at 1.9 degree by 2.5 degree resolution. The observations assimilated include all those used in the NCEP/NCAR reanalysis (temperature and wind components from radiosondes, aircraft, and satellite drift winds) plus radio occultation observations from the COSMIC satellites starting in late 2006. These files are intended to be used as 'DATM stream files' for CESM component sets that require a data atmosphere. Some example stream text files are included to illustrate how to use these data.

Guidance

Experience on a variety of machines has shown that it is a very good idea to make sure your run-time environment has the following:

```
limit stacksize unlimited
limit datasize unlimited
```

This page contains the documentation for the DART interface module for the CAM and WACCM models, using the dynamical cores listed above. This implementation uses the CAM initial files (**not** restart files) for transferring the model state to/from the filter. This may change in future versions, but probably only for CAM-SE. The reasons for this include:

1. The contents of the restart files vary depending on both the model release version and the physics packages selected.
2. There is no metadata describing the variables in the restart files. Some information can be tracked down in the `atm.log` file, but not all of it.
3. The restart files (for non-chemistry model versions) are much larger than the initial files (and we need to deal with an ensemble of them).
4. The temperature on the restart files is virtual equivalent potential temperature, which requires (at least) surface pressure, specific humidity, and sensible temperature to calculate.
5. CAM does not call the initialization routines when restart files are used, so fields which are not modified by DART may be inconsistent with fields which are.
6. If DART modifies the contents of the `.r.` restart file, it might also need to modify the contents of the `.rs.` restart file, which has similar characteristics (1-3 above) to the `.r.` file.

The DART interfaces to CAM and many of the other CESM components have been integrated with the CESM set-up and run scripts.

6.24.2 Setup Scripts

Unlike previous versions of DART-CAM, CESM runs using its normal scripts, then stops and calls a DART script, which runs a single assimilation step, then returns to the CESM run script to continue the model advances. See the CESM interface documentation in `$DARTROOT/models/CESM` for more information on running DART with CESM. Due to the complexity of the CESM software environment, the versions of CESM which can be used for assimilation are more restricted than previously. Each supported CESM version has similar, but unique, sets of set-up scripts and CESM SourceMods. Those generally do not affect the `cam-fv/model_mod.f90` interface. Current (April, 2015) set-up scripts are:

- `CESM1_2_1_setup_pmo`: sets up a perfect_model_mod experiment, which creates synthetic observations from a free model run, based on the user's somewhat restricted choice of model, dates, etc. The restrictions are made in order to streamline the script, which will shorten the learning curve for new users.
- `CESM1_2_1_setup_pmo_advanced`: same as `CESM1_2_1_setup_pmo`, but can handle more advanced set-ups: recent dates (non-default forcing files), refined-grid CAM-SE, etc.
- `CESM1_2_1_setup_hybrid`: streamlined script (see `CESM1_2_1_setup_pmo`) which sets up an ensemble assimilation using CESM's multi-instance capability.
- `CESM1_2_1_setup_advanced`: like `CESM1_2_1_setup_pmo_advanced`, but for setting up an assimilation.

The DART state vector should include all prognostic variables in the CAM initial files which cannot be calculated directly from other prognostic variables. In practice the state vector sometimes contains derived quantities to enable DART to compute forward operators (expected observation values) efficiently. The derived quantities are often overwritten when the model runs the next timestep, so the work DART does to update them is wasted work.

Expected observation values on pressure, scale height, height or model levels can be requested from `model_interpolate`. Surface observations can not yet be interpolated, due to the difference between the model surface and the earth's surface where the observations are made. Model_interpolate can be queried for any (non-surface) variable in the state vector (which are variables native to CAM) plus pressure on height levels. The default state vector is PS, T, U, V, Q, CLDLIQ, CLDICE and any tracers or chemicals needed for a given study. Variables which are not in the initial file can be added (see the `./doc` directory but minor modifications to `model_mod.f90` and CAM may be necessary.

The 19 public interfaces in `model_mod` are standardized for all DART compliant models. These interfaces allow DART to get the model state and metadata describing this state, find state variables that are close to a given location, and do spatial interpolation for a variety of variables required by observational operators.

6.24.3 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   cam_template_filename               = 'caminput.nc'
   cam_phis_filename                   = 'cam_phis.nc'
   vertical_localization_coord         = 'PRESSURE'
   use_log_vertical_scale              = .false.
   no_normalization_of_scale_heights   = .true.
   no_obs_assim_above_level            = -1,
   model_damping_ends_at_level         = -1,
   state_variables                     = ''
   assimilation_period_days            = 0
   assimilation_period_seconds         = 21600
```

(continues on next page)

(continued from previous page)

```
   suppress_grid_info_in_output         = .false.
   custom_routine_to_generate_ensemble = .true.
   fields_to_perturb                    = ''
   perturbation_amplitude               = 0.0_r8
   using_chemistry                      = .false.
   use_variable_mean_mass               = .false.
   debug_level                          = 0
/
```

The names of the fields to put into the state vector must match the CAM initial NetCDF file variable names.

Item	Type	Description
cam_template_file	character(len=128)	CAM initial file used to provide configuration information, such as the grid resolution, number of vertical levels, whether fields are staggered or not, etc.
cam_phis	character(len=128)	CAM topography file. Reads the "PHIS" NetCDF variable from this file. Typically this is a CAM History file because this field is not normally found in a CAM initial file.
vertical_localization_coord	character(len=128)	The vertical coordinate to which all vertical locations are converted in model_mod. Valid options are "pressure", "height", "scaleheight" or "level".
no_normalization_of_scale_heights	logical	If true the scale height is computed as the log of the pressure at the given location. If false the scale height is computed as a ratio of the log of the surface pressure and the log of the pressure aloft. In limited areas of high topography the ratio version might be advantageous, and in previous versions of filter this was the default. For global CAM the recommendation is to set this to .true. so the scale height is simply the log of the pressure at any location.
no_obs_assim_above_level	integer	Because the top of the model is highly damped it is recommended to NOT assimilate observations in the top model levels. The units here are CAM model level numbers. Set it to equal or below the lowest model level (the highest number) where damping is applied in the model.
model_damping_ends_at_level	integer	Set this to the lowest model level (the highest number) where model damping is applied. Observations below the 'no_obs_assim_above_level' cutoff but close enough to the model top to have an impact during the assimilation will have their impacts decreased smoothly to 0 at this given model level. The assimilation should make no changes to the model state above the given level.
state_variables	character(len=64), dimension(100)	Character string table that includes: Names of fields (NetCDF variable names) to be read into the state vector, the corresponding DART Quantity for that variable, if a bounded quantity the minimum and maximum valid values, and finally the string 'UPDATE' to indicate the updated values should be written back to the output file. 'NOUPDATE' will skip writing this field at the end of the assimilation.
assimilation_period_days	integer	Sets the assimilation window width, and should match the model advance time when cycling. The scripts distributed with DART always set this to 0 days, 21600 seconds (6 hours).
assimilation_period_seconds	integer	Sets the assimilation window width, and should match the model advance time when cycling. The scripts distributed with DART always set this to 0 days, 21600 seconds (6 hours).
suppress_grid_info_in_output	logical	Filter can update fields in existing files or create diagnostic/output files from scratch. By default files created from scratch include a full set of CAM grid information to make the file fully self-contained and plottable. However, to save disk space the grid variables can be suppressed in files created by filter by setting this to true.
custom_routine_to_generate_ensemble	logical	The default perturbation routine in filter adds gaussian noise equally to all fields in the state vector. It is recommended to set this option to true so code in the model_mod is called instead. This allows only a limited number of fields to be perturbed. For example, only perturbing the temperature field T with a small amount of noise and then running the model forward for a few days is often a recommended way to generate an ensemble from a single state.
fields_to_perturb	character(len=32), dimension(100)	If perturbing a single state to generate an ensemble, set 'custom_routine_to_generate_ensemble = .true.' and list list the field(s) to be perturbed here.
perturbation_amplitude	real(r8), dimension(100)	For each field name in the 'fields_to_perturb' list give the standard deviation for the gaussian noise to add to each field being perturbed.
pert_base_vals	real(r8), dimension(100)	If pert_sd is positive, this the list of values to which the field(s) listed in pert_names will be reset if filter is told to create an ensemble from a single state vector. Otherwise, it's is the list of values to use for each ensemble member when perturbing the single field named in pert_names. Unused unless pert_names is set and pert_base_vals is not the DART missing value.
using_chemistry	logical	If using CAM-CHEM, set this to .true.
us-	logical	If using any variant of WACCM with a very high model top, set this to .true.

Item	Type	Description
cam_template_file	character(len=128)	CAM initial file used to provide configuration information, such as the grid resolution, number of vertical levels, whether fields are staggered or not, etc.
cam_phis	character(len=128)	CAM topography file. Reads the "PHIS" NetCDF variable from this file. Typically this is a CAM History file because this field is not normally found in a CAM initial file.
vertical_localization_coord	character(len=128)	The vertical coordinate to which all vertical locations are converted in model_mod. Valid options are "pressure", "height", "scaleheight" or "level".
no_normalization_of_scale_heights	logical	If true the scale height is computed as the log of the pressure at the given location. If false the scale height is computed as a ratio of the log of the surface pressure and the log of the pressure aloft. In limited areas of high topography the ratio version might be advantageous, and in previous versions of filter this was the default. For global CAM the recommendation is to set this to .true. so the scale height is simply the log of the pressure at any location.
no_obs_assim_above_level	integer	Because the top of the model is highly damped it is recommended to NOT assimilate observations in the top model levels. The units here are CAM model level numbers. Set it to equal or below the lowest model level (the highest number) where damping is applied in the model.
model_damping_ends_at_level	integer	Set this to the lowest model level (the highest number) where model damping is applied. Observations below the 'no_obs_assim_above_level' cutoff but close enough to the model top to have an impact during the assimilation will have their impacts decreased smoothly to 0 at this given model level. The assimilation should make no changes to the model state above the given level.
state_variables	character(len=64), dimension(100)	Character string table that includes: Names of fields (NetCDF variable names) to be read into the state vector, the corresponding DART Quantity for that variable, if a bounded quantity the minimum and maximum valid values, and finally the string 'UPDATE' to indicate the updated values should be written back to the output file. 'NOUPDATE' will skip writing this field at the end of the assimilation.
assimilation_period_days	integer	Sets the assimilation window width, and should match the model advance time when cycling. The scripts distributed with DART always set this to 0 days, 21600 seconds (6 hours).
assimilation_period_seconds	integer	Sets the assimilation window width, and should match the model advance time when cycling. The scripts distributed with DART always set this to 0 days, 21600 seconds (6 hours).
suppress_grid_info_in_output	logical	Filter can update fields in existing files or create diagnostic/output files from scratch. By default files created from scratch include a full set of CAM grid information to make the file fully self-contained and plottable. However, to save disk space the grid variables can be suppressed in files created by filter by setting this to true.
custom_routine_to_generate_ensemble	logical	The default perturbation routine in filter adds gaussian noise equally to all fields in the state vector. It is recommended to set this option to true so code in the model_mod is called instead. This allows only a limited number of fields to be perturbed. For example, only perturbing the temperature field T with a small amount of noise and then running the model forward for a few days is often a recommended way to generate an ensemble from a single state.
fields_to_perturb	character(len=32), dimension(100)	If perturbing a single state to generate an ensemble, set 'custom_routine_to_generate_ensemble = .true.' and list list the field(s) to be perturbed here.
perturbation_amplitude	real(r8), dimension(100)	For each field name in the 'fields_to_perturb' list give the standard deviation for the gaussian noise to add to each field being perturbed.
pert_base_vals	real(r8), dimension(100)	If pert_sd is positive, this the list of values to which the field(s) listed in pert_names will be reset if filter is told to create an ensemble from a single state vector. Otherwise, it's is the list of values to use for each ensemble member when perturbing the single field named in pert_names. Unused unless pert_names is set and pert_base_vals is not the DART missing value.
using_chemistry	logical	If using CAM-CHEM, set this to .true.
us-	logical	If using any variant of WACCM with a very high model top, set this to .true.

6.24.4 Setup Variations

The variants of CAM require slight changes to the setup scripts (in `$DARTROOT/models/cam-fv/shell_scripts`) and in the namelists (in `$DARTROOT/models/cam-fv/work/input.nml`). From the DART side, assimilations can be started from a pre-existing ensemble, or an ensemble can be created from a single initial file before the first assimilation. In addition, there are setup differences between 'perfect model' runs, which are used to generate synthetic observations, and assimilation runs. Those differences are extensive enough that they've been coded into separate *Setup Scripts*.

Since the CESM compset and resolution, and the initial ensemble source are essentially independent of each other, changes for each of those may need to be combined to perform the desired setup.

Perturbed Ensemble

The default values in `work/input.nml` and `shell_scripts/CESM1_2_1_setup_pmo` and `shell_scripts/CESM1_2_1_setup_hybrid` are set up for a CAM-FV, single assimilation cycle using the default values as found in `model_mod.f90` and starting from a single model state, which must be perturbed into an ensemble. The following are suggestions for setting it up for other assimilations. Namelist variables listed here might be in any namelist within `input.nml`.

CAM-FV

If built with the FV dy-core, the number of model top levels with extra diffusion in CAM is controlled by `div24del2flag`. The recommended minium values of `highest_state_pressure_Pa` come from that variable, and `cutoff*vert_normalization_X`:

```
2    ("div2") -> 2 levels  -> highest_state_pressure_Pa =  9400. Pa
4,24 ("del2") -> 3 levels  -> highest_state_pressure_Pa = 10500. Pa
```

and:

```
vert_coord           = 'pressure'
state_num_1d         = 0,
state_num_2d         = 1,
state_num_3d         = 6,
state_names_1d       = ''
state_names_2d       = 'PS'
state_names_3d       = 'T', 'US', 'VS', 'Q', 'CLDLIQ', 'CLDICE'
which_vert_1d        = 0,
which_vert_2d        = -1,
which_vert_3d        = 6*1,
highest_state_pressure_Pa = 9400. or 10500.
```

WACCM

WACCM[#][-X] has a much higher top than the CAM versions, which requires the use of scale height as the vertical coordinate, instead of pressure, during assimilation. One impact of the high top is that the number of top model levels with extra diffusion in the FV version is different than in the low-topped CAM-FV, so the `div24del2flag` options lead to the following minimum values for `highest_state_pressure_Pa`:

```
2    ("div2") -> 3 levels  -> highest_state_pressure_Pa = 0.01 Pa
4,24 ("del2") -> 4 levels  -> highest_state_pressure_Pa = 0.02 Pa
```

The best choices of `vert_normalization_scale_height`, `cutoff`, and `highest_state_pressure_Pa` are still being investigated (April, 2015), and may depend on the observation distribution being assimilated.

WACCM is also typically run with coarser horizontal resolution. There's an existing 2-degree ensemble, so see the *Continuing after the first cycle* section to start from it, instead of a single state. If you use this, ignore any existing inflation restart file and tell DART to make its own in the first cycle in `input.nml`:

```
inf_initial_from_restart    = .false.,                 .false.,
inf_sd_initial_from_restart = .false.,                 .false.,
```

In any case, make the following changes (or similar) to convert from a CAM setup to a WACCM setup. `CESM1_2_1_setup_hybrid`:

```
setenv compset     F_2000_WACCM
setenv resolution  f19_f19
setenv refcase     FV1.9x2.5_WACCM4
setenv refyear     2008
setenv refmon      12
setenv refday      20
```

and the settings within `input.nml`:

```
vert_normalization_scale_height = 2.5
vert_coord                = 'log_invP'
highest_obs_pressure_Pa   = .001,
highest_state_pressure_Pa = .01,
```

If built with the SE dy-core (warning; experimental), then 4 levels will have extra diffusion.

If there are problems with instability in the WACCM foreasts, try changing some of the following parameters in either the user_nl_cam section of the setup script or input.nml.

- The default div24del2flag in WACCM is 4. Change it in the setup script to

  ```
  echo " div24del2flag         = 2 "                      >> ${fname}
  ```

  which will use the `cd_core.F90` in SourceMods, which has doubled diffusion in the top layers compared to CAM.

- Use a smaller dtime (1800 s is the default for 2-degree) in the setup script. This can also be changed in the ensemble of `user_nl_cam_####` in the `$CASEROOT` directory.

  ```
  echo " dtime         = 600 "                             >> ${fname}
  ```

- Increase highest_state_pressure_Pa in input.nml:

  ```
  div24del2flag = 2    ("div2") -> highest_state_pressure_Pa = 0.1 Pa
  div24del2flag = 4,24 ("del2") -> highest_state_pressure_Pa = 0.2 Pa
  ```

- Use a larger nsplit and/or nspltvrm in the setup script:

  ```
  echo " nsplit         = 16 "                             >> ${fname}
  echo " nspltvrm       =  4 "                             >> ${fname}
  ```

- Reduce `inf_damping` from the default value of `0.9` in `input.nml`:

  ```
  inf_damping           = 0.5,                    0,
  ```

6.24.5 Notes for Continuing an Integration

Continuing after the first cycle

After the first forecast+assimilation cycle, using an ensemble created from a single file, it is necessary to change to the 'continuing' mode, where CAM will not perform all of its startup procedures and DART will use the most recent ensemble. This example applies to an assimiation using prior inflation (`inf_...= .true.`). If posterior inflation were needed, then the 2nd column of `infl_...` would be set to `.true...` Here is an example snippet from `input.nml`:

```
start_from_restart       = .true.,
restart_in_file_name     = "filter_ics",
single_restart_file_in   = .false.,

inf_initial_from_restart    = .true.,                 .false.,
inf_sd_initial_from_restart = .true.,                 .false.,
```

Combining multiple cycles into one job

`CESM1_2_1_setup_hybrid` and `CESM1_2_1_setup_pmo` are set up in the default cycling mode, where each submitted job performs one model advance and one assimilation, then resubmits the next cycle as a new job. For long series of cycles, this can result in a lot of time waiting in the queue for short jobs to run. This can be prevented by using the 'cycles' scripts generated by `CESM1_2_1_setup_advanced` (instead of `CESM1_2_1_setup_hybrid`). This mode is described in `$DARTROOT/models/cam-fv/doc/README_cam-fv`.

6.24.6 Discussion

Many CAM initial file variables are already handled in the `model_mod`. Here is a list of others, which may be used in the future. Each would need to have a DART `*KIND*` associated with it in `model_mod`.

```
Atmos
   CLOUD:       "Cloud fraction" ;
   QCWAT:       "q associated with cloud water" ;
   TCWAT:       "T associated with cloud water" ;
   CWAT:        "Total Grid box averaged Condensate Amount (liquid + ice)" ;
   also? LCWAT

pbl
   PBLH:        "PBL height" ;
   QPERT:       "Perturbation specific humidity (eddies in PBL)" ;
   TPERT:       "Perturbation temperature (eddies in PBL)" ;

Surface
   LANDFRAC:    "Fraction of sfc area covered by land" ;
   LANDM:       "Land ocean transition mask: ocean (0), continent (1), transition (0-
↪1)" ;
      also LANDM_COSLAT
   ICEFRAC:     "Fraction of sfc area covered by sea-ice" ;
   SGH:         "Standard deviation of orography" ;
   Z0FAC:       "factor relating z0 to sdv of orography" ;
   TS:          "Surface temperature (radiative)" ;
   TSOCN:       "Ocean tempertare" ;
   TSICE:       "Ice temperature" ;
   TSICERAD:    "Radiatively equivalent ice temperature" ;
```

(continues on next page)

(continued from previous page)

```
Land/under surface
   SICTHK:       "Sea ice thickness" ;
   SNOWHICE:     "Water equivalent snow depth" ;
   TS1:          "subsoil temperature" ;
   TS2:          "subsoil temperature" ;
   TS3:          "subsoil temperature" ;
   TS4:          "subsoil temperature" ;

Other fields are not included because they look more CLM oriented.

Other fields which users may add to the CAM initial files are not listed here.
```

6.24.7 Files

- `model_nml` in `input.nml`
- `cam_phis.nc` (CAM surface height file, often CAM's .h0. file in the CESM run environment)
- `caminput.nc` (CAM initial file)
- `clminput.nc` (CLM restart file)
- `iceinput.nc` (CICE restart file) by model_mod at the start of each assimilation)
- netCDF output state diagnostics files

6.24.8 Nitty gritty: Efficiency possibilities

- index_from_grid (and others?) could be more efficient by calculating and globally storing the beginning index of each cfld and/or the size of each cfld. Get_state_meta_data too. See `clm/model_mod.f90`.
- Global storage of height fields? but need them on staggered grids (only sometimes) Probably not; machines going to smaller memory and more recalculation.
- ! Some compilers can't handle passing a section of an array to a subroutine/function; I do this in `nc_write_model_vars(?)` and/or `write_cam_init(?)`; replace with an exactly sized array?
- Is the testing of resolution in read_cam_coord overkill in the line that checks the size of `(resol_n - resol_1)*resol`?
- Replace some do loops with forall (constructs)
- Subroutine `write_cam_times(model_time, adv_time)` is not needed in CESM+DART framework? Keep anyway?
- Remove the code that accommodates old CAM coordinate order (`lon,lev,lat`).
- Cubed sphere: Convert lon,lat refs into dim1,dim2 in more subroutines. get_val_heights is called with (`column_ind,1`) by CAM-SE code, and (`lon_ind, lat_ind`) otherwise).
- `cam_to_dart_kinds` and `dart_to_cam_types` are dimensioned 300, regardless of the number of fields in the state vector and/or *KIND*s .
- Describe:
  - The coordinate orders and translations; CAM initial file, `model_mod`, and `DART_Diag.nc`.
  - Motivations

    - There need to be 2 sets of arrays for dimensions and dimids;
        - one describing the caminput file (`f_...`)
        - and one for the state (`s_...`) (storage in this module).
        - Call them `f_dim_Nd`, `f_dimid_Nd`
        - `s_dim_Nd`, `s_dimid_Nd`

- Change (private only) subroutine argument lists; structures first, regardless of in/out then output, and input variables.
- Change declarations to have dummy argument integers used as dimensions first
- Implement a `grid_2d_type`? Convert phis to a `grid_2d_type`? ps, and staggered ps fields could also be this type.
- Deallocate `grid_1d_arrays` using `end_1d_grid_instance` in end_model. `end_model` is called by subroutines `pert_model_state`, `nc_write_model_vars`; any problem?
- ISSUE; In `P[oste]rior_Diag.nc` ensemble members are written out *between* the field mean/spread pair and the inflation mean/sd pair. Would it make more sense to put members after both pairs? Easy to do?
- ISSUE?; `model_interpolate` assumes that obs with a vertical location have 2 horizontal locations too. The state vector may have fields for which this isn't true, but no obs we've seen so far violate this assumption. It would have to be a synthetic/perfect_model obs, like some sort of average or parameter value.
- ISSUE; In convert_vert, if a 2D field has dimensions (lev, lat) then how is `p_surf` defined? Code would be needed to set the missing dimension to 1, or make different calls to `coord_ind`, etc.
- ISSUE; The `QTY_` list from obs_def_mod must be updated when new fields are added to state vector. This could be done by the preprocessor when it inserts the code bits corresponding to the lists of observation types, but it currently (10/06) does not. Document accordingly.
- ISSUE: The CCM code (and Hui's packaging) for geopotentials and heights use different values of the physical constants than DART's. In one case Shea changed g from 9.81 to 9.80616, to get agreement with CCM(?…), so it may be important. Also, matching with Hui's tests may require using his values; change to DART after verifying?
- ISSUE: It's possible to figure out the model_version from the NetCDF file itself, rather than have that be user-provided (sometimes incorrect and hard to debug) meta-data. model_version is also misnamed; it's really the `caminput.nc` model version. The actual model might be a different version(?). The problem with removing it from the namelist is that the scripts need it too, so some rewriting there would be needed.
- ISSUE: `max_neighbors` is set to 6, but could be set to 4 for non-refined grids. Is there a good mechanism for this? Is it worth the file space savings?
- ISSUE: `x_planar` and `y_planar` could be reduced in rank, if no longer needed for testing and debugging.
- "Pobs" marks changes for providing expected obs of P break from past philosophy; P is not a native CAM variable (but is already calced here)
- NOVERT marks modifications for fields with no vertical location, i.e. GWD parameters.

6.24.9 References and Acknowledgements

- CAM homepage

Ave Arellano did the first work with CAM-Chem, assimilating MOPPITT CO observations into CAM-Chem. Jerome Barre and Benjamin Gaubert took up the development work from Ave, and prompted several additions to DART, as well as `model_mod.f90`.

Nick Pedatella developed the first `vert_coord = 'log_invP'` capability to enable assimilation using WACCM and scale height vertical locations.

6.25 CESM

6.25.1 Overview

This is the start of an interface for assimilating observations into the fully-coupled CESM whole-system model. It makes use of the existing POP (ocean), CLM (land), and CAM (atmosphere) model_mod codes.

Note: See ./doc/setup_guidelines.html for much more information.

We have adopted some terminology to help us keep things straight.

1. CESM already uses the term 'fully-coupled', so we use that in reference to CESM components only. It means that CESM has an active atmosphere and ocean, ignoring other components. In CESM an active atmosphere almost always implies an active land, but that is not necessary for it to be called 'fully coupled', and, by itself, is not 'fully coupled'.
2. We use the term 'single-component' to denote the situation in which the assimilations are performed for ONE active model component. Atmospheric obs directly impact the atmosphere, OR land obs directly impact the land, etc.. Any impact from the atmosphere to the land happens through interaction with the CESM coupler.
3. We use the term 'multi-component' to denote the situation in which the assimilations are performed separately for more than one active model component. Atmospheric obs directly impact the atmosphere AND ocean obs directly impact the ocean, etc.. Any impact from the atmosphere to the ocean happens through interaction with the CESM coupler.
4. 'cross-component' is used to specify the case when observations of one component can directly impact any/all of the other active components without going through the coupler.

Prior to 9 Nov 2015, models/CESM had programs that were an attempt to achieve the cross-component, fully-coupled data assimilation. Since this is being implemented with the Remote Memory Access (RMA) strategy that is not consistent with the current SVN trunk, the files that allow that usage pattern are being removed from the SVN trunk. However, there are scripts in ./shell_scripts which enable the 'multi-component' assimilation mode, which does not require a models/CESM/model_mod.f90, since it uses a separate filter for each component (cam-fv, pop, . . .).

The models/CESM/work directory has nothing of use in it, since there are no programs to interact with a cross-component DART state vector (a DART state that consists of atmosphere and/or ocean and/or land).

6.26 CICE

6.26.1 Overview

The Community Ice CodE (CICE) is a sea ice model that was first developed by Elizabeth Hunke as the Los Alamos Sea Ice Model. Its code base and capabilities have grown as a result of continued development by the broader geosciences community, an effort organized by the CICE Consortium.

The DART model interface was developed to work with CICE's dynamical core on an Arakawa B-grid.[1] When CICE is coupled to POP in CESM, the ocean and sea ice grids are identical.

According to the CICE manual:

> The spatial discretization is specialized for a generalized orthogonal B-grid as in Murray (1996)[2] or Smith et al. (1995).[3] The ice and snow area, volume and energy are given at the center of the cell, velocity is defined at the corners, and the internal ice stress tensor takes four different values within a grid cell; bilinear approximations are used for the stress tensor and the ice velocity across the cell, as described in Hunke and Dukowicz (2002).[4] This tends to avoid the grid decoupling problems associated with the B-grid.

Hence, in the DART interface:

- U, V are at grid cell corners
- T, h, hs, and the various scalar quantities are at grid cell centers

6.26.2 Development Notes

CICE is under active development to work with other grids, such as the unstructured grid in MPAS and the C-grid in MOM. Due to this activity, this README contains development notes chronicling the development of the model interface. We ask that developers continue to document the development both in this README and with descriptive comments in the source code.

- Possible bug found in `model_mod.f90` for POP where set_date is sent sec this day. The routine wants sec this min.

Notes from Cecilia M. Bitz on 14 May 2016

- Created `../../obs_def/obs_def_cice_mod.f90` to make new obs_kinds used in `model_mod.f90` and `input.nml`.
- Not sure about `QTY_TRACERARRAY_CATS`
- Model mod assumes that the grid is identical to POP
- Leaving this part but it may be unneeded in CICE `INTERFACE vector_to_prog_var MODULE PROCEDURE vector_to_2d_prog_var MODULE PROCEDURE vector_to_3d_prog_var END INTERFACE`

[1] Arakawa, Akio and Vivian R. Lamb, 1977: Computational Design of the Basic Dynamical Processes of the UCLA General Circulation Model. *Methods in Computational Physics: Advances in Research and Applications*, **17**, 173–265, doi:10.1016/B978-0-12-460817-7.50009-4

[2] Murray, Ross J., 1996: Explicit Generation of Orthogonal Grids for Ocean Models. *Journal of Computational Physics*, **126**, 251–273, doi:10.1006/jcph.1996.0136

[3] Smith, Richard D., Samuel Kortas and Bertrand Meltz, 1995: Curvilinear Coordinates for Global Ocean Models. Technical Report LA-UR95-1146, Los Alamos National Laboratory.

[4] Hunke, Elizabeth C., and John K. Dukowicz, 2002: The Elastic–Viscous–Plastic Sea Ice Dynamics Model in General Orthogonal Curvilinear Coordinates on a Sphere—Incorporation of Metric Terms. *Monthly Weather Review*, **130**, 1848–1865, doi:10.1175/1520-0493(2002)130%3C1848:TEVPSI%3E2.0.CO;2

- Not used in pop so of course not used now in CICE either, why? `subroutine vector_to_3d_prog_var(x, varindex, data_3d_array) subroutine get_gridsize(num_x, num_y, num_z)`
- Come back here, some changes made below but need to look line-by-line still `subroutine get_state_meta_data(state_handle, index_in, location, var_type)`

Fortran Files

- `dart_to_cice.f90` Think it is done
- `cice_to_dart.f90` Is trivial so hope it's done too
- `model_mod_check.f90`
- `dart_cice_mod.f90` Should it have a get_cat_dim?
- `model_mod.f90` I do not understand this part, but appears in clm too: `INTERFACE vector_to_prog_var MODULE PROCEDURE vector_to_1d_prog_var ! this is in clm MODULE PROCEDURE vector_to_2d_prog_var ! this is in pop MODULE PROCEDURE vector_to_3d_prog_var ! this is in pop END INTERFACE`
- `test_dipole_interp.f90` Also trivial, nothing to change?
- `rma-cice/location/` Has a bunch of subdirs each with a location_mod.f90

```
-rw-r--r--  1 bitz  staff  142664 May 26 17:12 model_mod.f90
-rw-r--r--  1 bitz  staff    4439 May 21 07:55 cice_to_dart.f90
-rw-r--r--  1 bitz  staff    5676 May 21 07:49 dart_to_cice.f90
-rw-r--r--  1 bitz  staff   24008 May 18 21:55 dart_cice_mod.f90
-rw-r--r--  1 bitz  staff   24294 May 14 16:30 model_mod_check.f90
-rw-r--r--  1 bitz  staff    2270 May 14 16:30 test_dipole_interp.f90
```

Code Snippet from model_mod.f90

```
! these routines must be public and you cannot change
! the arguments - they will be called *from* the DART code.
public :: get_model_size,                &
          adv_1step,                     &
          get_state_meta_data,           &
          model_interpolate,             &
          get_model_time_step,           &
          static_init_model,             &
          end_model,                     &
          init_time,                     &
          init_conditions,               &
          nc_write_model_atts,           &
          nc_write_model_vars,           &
          pert_model_copies,             &
          get_close_maxdist_init,        &
          get_close_obs_init,            &
          get_close_obs,                 &
          query_vert_localization_coord, &
          vert_convert,                  &
          construct_file_name_in,        &
          read_model_time,               &
          write_model_time
```

6.26.3 Namelist

```
&model_nml
   assimilation_period_days      = 1
   assimilation_period_seconds   = 0
   model_perturbation_amplitude = 0.00002
   binary_grid_file_format       = 'big_endian'
   debug                         = 1
   model_state_variables         = 'aicen', 'QTY_SEAICE_CONCENTR',    'UPDATE',
                                   'vicen', 'QTY_SEAICE_VOLUME',      'UPDATE',
                                   ...
                                   'vsnon', 'QTY_SEAICE_SNOWVOLUME', 'UPDATE',
/
```

Description of each namelist entry

Item	Type	Description
time_step_days	integer	Number of days for dimensional timestep, mapped to deltat.
time_step_seconds	integer	Number of seconds for dimensional timestep, mapped to deltat.
model_perturbation_amplitude	real(r8)	Perturbation amplitude
binary_grid_file_format	character(64)	Byte sequence for the binary grid. Valid values are native, big_endian & little_endian.
debug	integer	When set to 0, debug statements are not printed. Higher numbers mean more debug reporting.
model_state_variables	character(*)	List of model state variables

References

6.27 CLM

6.27.1 Overview

This is the DART interface to the Community Land Model (CLM). It is run as part of the Community Earth System Model (CESM) framework. It is **strongly** recommended that you become familiar with running a multi-instance experiment in CESM **before** you try to run DART/CLM. The DART/CLM facility uses language and concepts that should be familiar to CESM users. The DART/CLM capability is entirely dependent on the multi-instance capability of CESM, first supported in its entirety in CESM1.1.1. Consequently, this version or newer is required to run CLM/DART. The CLM User's Guide is an excellent reference for CLM. *As of (V7195) 3 October 2014, CESM1.2.1 is also supported.*

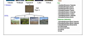
DART uses the multi-instance capability of CESM, which means that DART is not responsible for advancing the model. This GREATLY simplifies the traditional DART workflow, but it means *CESM has to stop and write out a restart file every time an assimilation is required*. The multi-instance capability is very new to CESM and we are in close collaboration with the CESM developers to make using DART with CESM as easy as possible. While we strive to keep DART requirements out of the model code, there are a few SourceMods needed to run DART from within CESM. Appropriate SourceMods for each CESM version are available at http://www.image.ucar.edu/pub/DART/CESM and should be unpacked into your HOME directory. They will create a `~/cesm_?_?_?` directory with the appropriate SourceMods structure. The ensuing scripts require these SourceMods and expect them to be in your HOME directory.

Our notes on how to set up, configure, build, and run CESM for an assimilation experiment evolved into scripts. These scripts are not intended to be a 'black box'; you will have to read and understand them and modify them to your own purpose. They are heavily commented – in keeping with their origins as a set of notes. If you would like to offer suggestions on how to improve those notes - please send them to dart@ucar.edu - we'd love to hear them.

Script	Description
CESM1_1_1_setup_pmo	runs a single instance of CLM to harvest synthetic observations for an OSSE or "perfect model" experiment. It requires a single CLM state from a previous experiment and uses a specified DATM stream for forcing. This parallels an assimilation experiment in that in the multi-instance setting each CLM instance may use (should use?) a unique DATM forcing. This script has almost nothing to do with DART. There is one (trivial) section that records some configuration information in the DART setup script, but that's about it. This script should initially be run without DART to ensure a working CESM environment. As of (V7195) 3 October 2014, this script demonstrates how to create 'vector'-based CLM history files (which requires a bugfix) and has an option to use a bugfixed snow grain-size code. http://bugs.cgd.ucar.edu/show_bug.cgi?id=1730 http://bugs.cgd.ucar.edu/show_bug.cgi?id=1934
CESM1_2_1_setup_pmo	Is functionally identical to `CESM1_1_1_setup_pmo` but is appropriate for the the CESM 1_2_1 release, which supports both CLM 4 and CLM 4.5.
CESM1_1_1_setup_hybrid	runs a multi-instance CLM experiment and can be used to perform a free run or 'open loop' experiment. By default, each CLM instance uses a unique DATM forcing. This script also has almost nothing to do with DART. There is one (trivial) section that records some configuration information in the DART setup script, but that's about it. This script should initially be run without DART to ensure a working CESM. As of (V7195) 3 October 2014, this script demonstrates how to create 'vector'-based CLM history files (which requires a bugfix) and has an option to use a bugfixed snow grain-size code. http://bugs.cgd.ucar.edu/show_bug.cgi?id=1730 http://bugs.cgd.ucar.edu/show_bug.cgi?id=1934
CESM1_2_1_setup_hybrid	Is functionally identical to `CESM1_1_1_setup_hybrid` but is appropriate for the the CESM 1_2_1 release, which supports both CLM 4 and CLM 4.5.
CESM_DART_config	augments a CESM case with the bits and pieces required to run DART. When either `CESM1_?_1_setup_pmo` or `CESM1_?_1_setup_hybrid` gets executed, `CESM_DART_config` gets copied to the CESM "caseroot" directory. It is designed such that you can execute it at any time during a CESM experiment. When you do execute it, it will build the DART executables and copy them into the CESM "bld" directory, stage the run-time configurable `input.nml` in the "caseroot" directory, etc. and also *modifies* the CESM `case.run` script to call the DART scripts for assimilation or to harvest synthetic observations.

In addition to the script above, there are a couple scripts that will either perform an assimilation (assimilate.csh) or harvest observations for a perfect model experiment (perfect_model.csh). These scripts are designed to work on several compute platforms although they require configuration, mainly to indicate the location of the DART observation sequence files on your system.

6.27.2 Pertinent details of the CLM gridcell

	"The land surface is represented by 5 primary sub-grid land cover types (landunits: glacier, lake, wetland, urban, vegetated) in each grid cell. The vegetated portion of a grid cell is further divided into patches of plant functional types, each with its own leaf and stem area index and canopy height. Each subgrid land cover type and PFT patch is a separate column for energy and water calculations." – CLM documentation. The only location information available is at the gridcell level. All landunits, columns, and PFTs in that gridcell have the same location. This has ramifications for the forward observation operators. If the observation metadata has information about land use/land cover, it can be used to select only those patches that are appropriate. Otherwise, an area-weighted average of ALL patches in the gridcell is used to calculate the observation value for that location.

6.27.3 A word about forward observation operators

“Simple” observations like snowcover fraction come directly from the DART state. It is possible to configure the CLM history files to contain the CLM estimates of some quantities (mostly flux tower observations e.g, net ecosystem production, sensible heat flux, latent heat flux) that are very complicated combinations of portions of the CLM state. The forward observation operators for these flux tower observations read these quantities from the CLM `.h1.` history file. The smaller the CLM gridcell, the more likely it seems that these values will agree with point observations.

The prior and posterior values for these will naturally be identical as the history file is unchanged by the assimilation. Configuring the CLM user_nl_clm files to output the desired quantities must be done at the first execution of CLM. As soon as CONTINUE_RUN=TRUE, the namelist values for history file generation are ignored. Because the history file creation is very flexible, some additional information must be passed to DART to construct the filename of the `.h1.` history file needed for any particular time.

6.27.4 Major changes as of (v7195) 3 october 2014

The DART state vector may be constructed in a much more flexible way. Variables from two different CLM history files may also be incorporated directly into the DART state - which should GREATLY speed up the forward observation operators - and allow the observation operators to be constructed in a more flexible manner so that they can be used by any model capable of providing required inputs. It is now possible to read some variables from the restart file, some variables from a traditional history file, and some from a ‘vector-based’ history file that has the same structure (gridcell/landunit/column/pft) as the restart file. This should allow more accurate forward observation operators since the quantities are not gridcell-averaged a priori.

Another namelist item has been added `clm_vector_history_filename` to support the concept that two history files can be supported. My intent was to have the original history file (required for grid metadata) and another for support of vector-based quantities in support of forward observation operators. Upon reflection, I’m not sure I need two different history files - BUT - I’m sure there will be a situation where it comes in handy.

The new namelist specification of what goes into the DART state vector includes the ability to specify if the quantity should have a lower bound, upper bound, or both, what file the variable should be read from, and if the variable should be modified by the assimilation or not. **Only variables in the CLM restart file will be candidates for updating.** No CLM history files are modified. **It is important to know that the variables in the DART diagnostic files ``preassim.nc`` and ``analysis.nc`` will contain the unbounded versions of ALL the variables specied in ``clm_variables``.**

The example `input.nml` `model_nml` demonstrates how to construct the DART state vector. The following table explains in detail each entry for `clm_variables`:

Column 1	Column 2	Column 3	Column 4	Column 5	Column 6
Variable name	DART KIND	minimum	maximum	filename	update

Column 1	Variable name	This is the CLM variable name as it appears in the CLM netCDF file.
Column 2	DART KIND	This is the character string of the corresponding DART KIND.
Column 3	minimum	If the variable is to be updated in the CLM restart file, this specifies the minimum value. If set to 'NA', there is no minimum value.
Column 4	maximum	If the variable is to be updated in the CLM restart file, this specifies the maximum value. If set to 'NA', there is no maximum value.
Column 5	filename	This specifies which file should be used to obtain the variable. `'restart'` => clm_restart_filename `'history'` => clm_history_filename `'vector'` => clm_vector_history_filename
Column 6	update	If the variable comes from the restart file, it may be updated after the assimilation. `'UPDATE'` => the variable in the restart file is updated. `'NO_COPY_BACK'` => the variable in the restart file remains unchanged.

The following are only meant to be examples - they are not scientifically validated. Some of these that are UPDATED are probably diagnostic quantities, Some of these that should be updated may be marked NO_COPY_BACK. There are multiple choices for some DART kinds. This list is by no means complete.

```
'livecrootc',  'QTY_ROOT_CARBON',            'NA', 'NA', 'restart', 'UPDATE',
'deadcrootc',  'QTY_ROOT_CARBON',            'NA', 'NA', 'restart', 'UPDATE',
'livestemc',   'QTY_STEM_CARBON',            'NA', 'NA', 'restart', 'UPDATE',
'deadstemc',   'QTY_STEM_CARBON',            'NA', 'NA', 'restart', 'UPDATE',
'livecrootn',  'QTY_ROOT_NITROGEN',          'NA', 'NA', 'restart', 'UPDATE',
'deadcrootn',  'QTY_ROOT_NITROGEN',          'NA', 'NA', 'restart', 'UPDATE',
'livestemn',   'QTY_STEM_NITROGEN',          'NA', 'NA', 'restart', 'UPDATE',
'deadstemn',   'QTY_STEM_NITROGEN',          'NA', 'NA', 'restart', 'UPDATE',
'litr1c',      'QTY_LEAF_CARBON',            'NA', 'NA', 'restart', 'UPDATE',
'litr2c',      'QTY_LEAF_CARBON',            'NA', 'NA', 'restart', 'UPDATE',
'litr3c',      'QTY_LEAF_CARBON',            'NA', 'NA', 'restart', 'UPDATE',
'soil1c',      'QTY_SOIL_CARBON',            'NA', 'NA', 'restart', 'UPDATE',
'soil2c',      'QTY_SOIL_CARBON',            'NA', 'NA', 'restart', 'UPDATE',
'soil3c',      'QTY_SOIL_CARBON',            'NA', 'NA', 'restart', 'UPDATE',
'soil4c',      'QTY_SOIL_CARBON',            'NA', 'NA', 'restart', 'UPDATE',
'fabd',        'QTY_FPAR_DIRECT',            'NA', 'NA', 'restart', 'UPDATE',
'fabi',        'QTY_FPAR_DIFFUSE',           'NA', 'NA', 'restart', 'UPDATE',
'T_VEG',       'QTY_VEGETATION_TEMPERATURE', 'NA', 'NA', 'restart', 'UPDATE',
'fabd_sun_z',  'QTY_FPAR_SUNLIT_DIRECT',     'NA', 'NA', 'restart', 'UPDATE',
'fabd_sha_z',  'QTY_FPAR_SUNLIT_DIFFUSE',    'NA', 'NA', 'restart', 'UPDATE',
'fabi_sun_z',  'QTY_FPAR_SHADED_DIRECT',     'NA', 'NA', 'restart', 'UPDATE',
'fabi_sha_z',  'QTY_FPAR_SHADED_DIFFUSE',    'NA', 'NA', 'restart', 'UPDATE',
'elai',        'QTY_LEAF_AREA_INDEX',        'NA', 'NA', 'restart', 'UPDATE',
```

Only the first variable for a DART kind in the clm_variables list will be used for the forward observation operator. The following is perfectly legal (for CLM4, at least):

```
clm_variables = 'LAIP_VALUE', 'QTY_LEAF_AREA_INDEX', 'NA', 'NA', 'restart' , 'UPDATE',
                'tlai',       'QTY_LEAF_AREA_INDEX', 'NA', 'NA', 'restart' , 'UPDATE',
                'elai',       'QTY_LEAF_AREA_INDEX', 'NA', 'NA', 'restart' , 'UPDATE',
                'ELAI',       'QTY_LEAF_AREA_INDEX', 'NA', 'NA', 'history' , 'NO_COPY_
↪BACK',
```

(continues on next page)

(continued from previous page)

```
                'LAISHA',     'QTY_LEAF_AREA_INDEX', 'NA', 'NA', 'history' , 'NO_COPY_
↪BACK',
                'LAISUN',     'QTY_LEAF_AREA_INDEX', 'NA', 'NA', 'history' , 'NO_COPY_
↪BACK',
                'TLAI',       'QTY_LEAF_AREA_INDEX', 'NA', 'NA', 'history' , 'NO_COPY_
↪BACK',
                'TLAI',       'QTY_LEAF_AREA_INDEX', 'NA', 'NA', 'vector'  , 'NO_COPY_
↪BACK'
   /
```

however, only LAIP_VALUE will be used to calculate the LAI when an observation of LAI is encountered. All the other LAI variables in the DART state will be modified by the assimilation based on the relationship of LAIP_VALUE and the observation. Those coming from the restart file and marked 'UPDATE' **will** be updated in the CLM restart file.

6.27.5 Namelist

These namelists are read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
  clm_restart_filename          = 'clm_restart.nc',
  clm_history_filename          = 'clm_history.nc',
  clm_vector_history_filename   = 'clm_vector_history.nc',
  output_state_vector           = .false.,
  assimilation_period_days      = 2,
  assimilation_period_seconds   = 0,
  model_perturbation_amplitude  = 0.2,
  calendar                      = 'Gregorian',
  debug                         = 0
  clm_variables  = 'frac_sno',    'QTY_SNOWCOVER_FRAC',        'NA' , 'NA', 'restart
↪' , 'NO_COPY_BACK',
                   'H2OSNO',      'QTY_SNOW_WATER',            '0.0', 'NA', 'restart
↪' , 'UPDATE',
                   'H2OSOI_LIQ',  'QTY_SOIL_MOISTURE',         '0.0', 'NA', 'restart
↪' , 'UPDATE',
                   'H2OSOI_ICE',  'QTY_ICE',                   '0.0', 'NA', 'restart
↪' , 'UPDATE',
                   'T_SOISNO',    'QTY_SOIL_TEMPERATURE',      'NA' , 'NA', 'restart
↪' , 'UPDATE',
                   'SNOWDP',      'QTY_SNOW_THICKNESS',        'NA' , 'NA', 'restart
↪' , 'UPDATE',
                   'LAIP_VALUE',  'QTY_LEAF_AREA_INDEX',       'NA' , 'NA', 'restart
↪' , 'NO_COPY_BACK',
                   'cpool',       'QTY_CARBON',                '0.0', 'NA', 'restart
↪' , 'UPDATE',
                   'frootc',      'QTY_ROOT_CARBON',           '0.0', 'NA', 'restart
↪' , 'UPDATE',
                   'leafc',       'QTY_LEAF_CARBON',           '0.0', 'NA', 'restart
↪' , 'UPDATE',
                   'leafn',       'QTY_LEAF_NITROGEN',         '0.0', 'NA', 'restart
↪' , 'UPDATE',
                   'NEP',         'QTY_NET_CARBON_PRODUCTION', 'NA' , 'NA', 'history
↪' , 'NO_COPY_BACK',
```

(continues on next page)

(continued from previous page)

```
                    'TV',          'QTY_VEGETATION_TEMPERATURE', 'NA' , 'NA', 'vector'␣
↪ , 'NO_COPY_BACK',
                    'RH2M_R',      'QTY_SPECIFIC_HUMIDITY',      'NA' , 'NA', 'vector'␣
↪ , 'NO_COPY_BACK',
                    'PBOT',        'QTY_SURFACE_PRESSURE',       'NA' , 'NA', 'vector'␣
↪ , 'NO_COPY_BACK',
                    'TBOT',        'QTY_TEMPERATURE',            'NA' , 'NA', 'vector'␣
↪ , 'NO_COPY_BACK'
   /
```

Item	Type	Description
clm_restart_filename	character(len=256)	this is the filename of the CLM restart file. The DART scripts resolve linking the specific CLM restart file to this generic name. This file provides the elements used to make up the DART state vector. The variables are in their original landunit, column, and PFT-based representations.
clm_history_filename	character(len=256)	this is the filename of the CLM `.h0.` history file. The DART scripts resolve linking the specific CLM history file to this generic name. Some of the metadata needed for the DART/CLM interfaces is contained only in this history file, so it is needed for all DART routines.
clm_vector_history_filename	character(len=256)	this is the filename of a second CLM history file. The DART scripts resolve linking the specific CLM history file to this generic name. The default setup scripts actually create 3 separate CLM history files, the `.h2.` ones are linked to this filename. It is possible to create this history file at the same resolution as the restart file, which should make for better forward operators. It is only needed if some of the variables specified in `clm_variables` come from this file.
output_state_vector	logical	If .true. write state vector as a 1D array to the DART diagnostic output files. If .false. break state vector up into variables before writing to the output files.
assimilation_period_days, assimilation_period_seconds	integer	Combined, these specify the width of the assimilation window. The current model time is used as the center time of the assimilation window. All observations in the assimilation window are assimilated. BEWARE: if you put observations that occur before the beginning of the assimilation_period, DART will error out because it cannot move the model 'back in time' to process these observations.
model_perturbation_amplitude	real(r8)	Required by the DART interfaces, but not used by CLM.
calendar	character(len=32)	string specifying the calendar to use with DART. The CLM dates will be interpreted with this same calendar. For assimilations with real observations, this should be 'Gregorian'.
debug	integer	Set to 0 (zero) for minimal output. Successively higher values generate successively more output. Not all values are important, however. It seems I've only used values [3,6,7,8]. Go figure.
clm_state_variables clm_variables	character(:,6)	Strings that identify the CLM variables, their DART kind, the min & max values, what file to read from, and whether or not the file should be updated after the assimilation. *Only CLM variable names in the CLM restart file are valid.* The DART kind must be one found in the `DART/assimilation_code/mo dules/observations/obs_kind_mod.f90` AFTER it gets built by `preprocess`. Most of the land observation kinds are specified by `DART/observations/for ward_operators/obs_def_land_mod.f90` and `DART/observations/forwa rd_operators/obs_def_tower_mod.f90`, so they should be specified in the preprocess_nml:input_files variable.

```
&obs_def_tower_nml
```

(continues on next page)

(continued from previous page)

```
   casename    = '../clm_dart',
   hist_nhtfrq = -24,
   debug       = .false.
   /
```

Item	Type	Description
casename	character(len=256)	this is the name of the CESM case. It is used by the forward observation operators to help construct the filename of the CLM `.h1.` history files for the flux tower observations. When the `input.nml` gets staged in the CASEROOT directory by `CESM_DART_config`, the appropriate value should automatically be inserted.
hist_nhtfrq	integer	this is the same value as in the CLM documentation. A negative value indicates the number of hours contained in the `.h1.` file. This value is needed to constuct the right `.h1.` filename. When the `input.nml` gets staged in the CASEROOT directory by `CESM_DART_config`, the appropriate value should automatically be inserted. Due to the large number of ways of specifying the CLM history file information, the correct value here is very dependent on how the case was configured. You would be wise to check it.
debug	logical	Set to .false. for minimal output.

6.27.6 Other modules used (directly)

```
types_mod
time_manager_mod
threed_sphere/location_mod
utilities_mod
obs_kind_mod
obs_def_land_mod
obs_def_tower_mod
random_seq_mod
```

6.27.7 Public interfaces - required

use model_mod, only :	get_model_size
	adv_1step
	get_state_meta_data
	model_interpolate
	get_model_time_step
	static_init_model
	end_model
	init_time
	init_conditions
	nc_write_model_atts
	nc_write_model_vars
	pert_model_state
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	ens_mean_for_model

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

model_size = get_model_size( )

```
integer :: get_model_size
```

Returns the length of the model state vector.

`model_size`	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

Advances the model for a single time step. The time associated with the initial model state is also input although it is not used for the computation.

`x`	State vector of length model_size.
`time`	Specifies time of the initial model state.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,              intent(in)  :: index_in
type(location_type),  intent(out) :: location
integer, optional,    intent(out) ::  var_type
```

Returns metadata about a given element, indexed by index_in, in the model state vector. The location defines where the state variable is located.

`index_in`	Index of state vector element about which information is requested.
`location`	The location of state variable element.
var_type	The generic DART kind of the state variable element.

call model_interpolate(x, location, itype, obs_val, istatus)

```
real(r8), dimension(:), intent(in)  :: x
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: itype
real(r8),               intent(out) :: obs_val
integer,                intent(out) :: istatus
```

Given model state, returns the value interpolated to a given location.

`x`	A model state vector.
`location`	Location to which to interpolate.
`itype`	Not used.
`obs_val`	The interpolated value from the model.
`istatus`	If the interpolation was successful `istatus = 0`. If `istatus /= 0` the interpolation failed. Values less than zero are reserved for DART.

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

Returns the time step (forecast length) of the model;

`var`	Smallest time step of model.

call static_init_model()

Used for runtime initialization of model; reads namelist, initializes model parameters, etc. This is the first call made to the model by any DART-compliant assimilation routine.

call end_model()

A stub.

call init_time(time)

```
type(time_type), intent(out) :: time
```

Returns the time at which the model will start if no input initial conditions are to be used. This is used to spin-up the model from rest.

`time`	Initial model time.

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

Returns default initial conditions for the model; generally used for spinning up initial model states.

`x`	Initial conditions for state vector.

ierr = nc_write_model_atts(ncFileID)

```
integer              :: nc_write_model_atts
integer, intent(in)  :: ncFileID
```

Function to write model specific attributes to a netCDF file. At present, DART is using the NetCDF format to output diagnostic information. This is not a requirement, and models could choose to provide output in other formats. This function writes the metadata associated with the model to a NetCDF file opened to a file identified by ncFileID.

`ncFileID`	Integer file descriptor to previously-opened netCDF file.
`ierr`	Returns a 0 for successful completion.

ierr = nc_write_model_vars(ncFileID, statevec, copyindex, timeindex)

```
integer                                 :: nc_write_model_vars
integer,                    intent(in)  :: ncFileID
real(r8), dimension(:),     intent(in)  :: statevec
integer,                    intent(in)  :: copyindex
integer,                    intent(in)  :: timeindex
```

Writes a copy of the state variables to a netCDF file. Multiple copies of the state for a given time are supported, allowing, for instance, a single file to include multiple ensemble estimates of the state.

`ncFileID`	file descriptor to previously-opened netCDF file.
`statevec`	A model state vector.
`copyindex`	Integer index of copy to be written.
`timeindex`	The timestep counter for the given state.
`ierr`	Returns 0 for normal completion.

call pert_model_state(state, pert_state, interf_provided)

```
real(r8), dimension(:), intent(in)   :: state
real(r8), dimension(:), intent(out)  :: pert_state
logical,                intent(out)  :: interf_provided
```

Given a model state, produces a perturbed model state.

`state`	State vector to be perturbed.
`pert_state`	Perturbed state vector: NOT returned.
`interf_provided`	Returned false; interface is not implemented.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
real(r8),             intent(in)    :: maxdist
```

In distance computations any two locations closer than the given `maxdist` will be considered close by the `get_close_obs()` routine. Pass-through to the 3D Sphere locations module. See get_close_maxdist_init() for the documentation of this subroutine.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(num)
```

Pass-through to the 3D Sphere locations module. See get_close_obs_init() for the documentation of this subroutine.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, num_close, close_ind [, dist])

```
type(get_close_type), intent(in)  :: gc
type(location_type),  intent(in)  :: base_obs_loc
integer,              intent(in)  :: base_obs_kind
type(location_type),  intent(in)  :: obs(:)
integer,              intent(in)  :: obs_kind(:)
integer,              intent(out) :: num_close
integer,              intent(out) :: close_ind(:)
real(r8), optional,   intent(out) :: dist(:)
```

Pass-through to the 3D Sphere locations module. See get_close_obs() for the documentation of this subroutine.

call ens_mean_for_model(ens_mean)

```
real(r8), dimension(:), intent(in) :: ens_mean
```

A NULL INTERFACE in this model.

`ens_mean`	State vector containing the ensemble mean.

6.27.8 Public interfaces - optional

use model_mod, only :	get_gridsize
	clm_to_dart_state_vector
	sv_to_restart_file
	get_clm_restart_filename
	get_state_time
	get_grid_vertval
	compute_gridcell_value
	gridcell_components
	DART_get_var
	get_model_time

call get_gridsize(num_lon, num_lat, num_lev)

```
integer, intent(out) :: num_lon, num_lat, num_lev
```

Returns the number of longitudes, latitudes, and total number of levels in the CLM state.

`num_lon`	The number of longitude grid cells in the CLM state. This comes from the CLM history file.
`num_lat`	The number of latitude grid cells in the CLM state. This comes from the CLM history file.
`num_lev`	The number of levels grid cells in the CLM state. This comes from ‘nlevtot’ in the CLM restart file.

call clm_to_dart_state_vector(state_vector, restart_time)

```
real(r8),          intent(inout) :: state_vector(:)
type(time_type),   intent(out)   :: restart_time
```

Reads the current time and state variables from CLM netCDF file(s) and packs them into a DART state vector. This MUST happen in the same fashion as the metadata arrays are built. The variables are specified by `model_nml:clm_variables`. Each variable specifies its own file of origin. If there are multiple times in the file of origin, only the time that matches the restart file are used.

`state_vector`	The DART state vector.
`restart_time`	The valid time of the CLM state.

call sv_to_restart_file(state_vector, filename, dart_time)

```
real(r8),          intent(in) :: state_vector(:)
character(len=*), intent(in) :: filename
type(time_type),   intent(in) :: dart_time
```

This routine updates the CLM restart file with the posterior state from the assimilation. Some CLM variables that are useful to include in the DART state (frac_sno, for example) are diagnostic quantities and are not used for subsequent model advances. The known diagnostic variables are NOT updated. If the values created by the assimilation are outside physical bounds, or if the original CLM value was 'missing', the `vector_to_prog_var()` subroutine ensures that the values in the original CLM restart file are **not updated**.

`state_vector`	The DART state vector containing the state modified by the assimilation.
`filename`	The name of the CLM restart file. **The contents of some of the variables will be overwritten with new values.**
`dart_time`	The valid time of the DART state. This has to match the time in the CLM restart file.

call get_clm_restart_filename(filename)

```
character(len=*), intent(out) :: filename
```

provides access to the name of the CLM restart file to routines outside the scope of this module.

`filename`	The name of the CLM restart file.

time = get_state_time(file_handle)

```
integer,          intent(in) :: file_handle
character(len=*), intent(in) :: file_handle
type(time_type)              :: get_state_time
```

This routine has two interfaces - one for an integer input, one for a filename. They both return the valid time of the model state contained in the file. The file referenced is the CLM restart file in netCDF format.

`file_handle`	If specified as an integer, it must be the netCDF file identifier from nf90_open(). If specified as a filename, the name of the netCDF file.
`time`	A DART time-type that contains the valid time of the model state in the CLM restart file.

call get_grid_vertval(x, location, varstring, interp_val, istatus)

```
real(r8),             intent(in)  :: x(:)
type(location_type), intent(in)  :: location
character(len=*),     intent(in)  :: varstring
real(r8),             intent(out) :: interp_val
integer,              intent(out) :: istatus
```

Calculate the value of quantity at depth. The gridcell value at the levels above and below the depth of interest are calculated and then the value for the desired depth is linearly interpolated. Each gridcell value is an area-weighted value of an unknown number of column- or pft-based quantities. This is one of the workhorse routines for `model_interpolate()`.

`x`	The DART state vector.
`location`	The location of the desired quantity.
`varstring`	The CLM variable of interest - this must be part of the DART state. e.g, T_SOISNO, H2OSOI_LIQ, H2OSOI_ICE . . .
`interp_val`	The quantity at the location of interest.
`istatus`	error code. 0 (zero) indicates a successful interpolation.

call compute_gridcell_value(x, location, varstring, interp_val, istatus)

```
real(r8),             intent(in)  :: x(:)
type(location_type), intent(in)  :: location
character(len=*),     intent(in)  :: varstring
real(r8),             intent(out) :: interp_val
integer,              intent(out) :: istatus
```

Calculate the value of a CLM variable in the DART state vector given a location. Since the CLM location information is only available at the gridcell level, all the columns in a gridcell are area-weighted to derive the value for the location. This is one of the workhorse routines for `model_interpolate()`, and only select CLM variables are currently supported. Only CLM variables that have no vertical levels may use this routine.

`x`	The DART state vector.
`location`	The location of the desired quantity.
`varstring`	The CLM variable of interest - this must be part of the DART state. e.g, frac_sno, leafc, ZWT . . .
`interp_val`	The quantity at the location of interest.
`istatus`	error code. 0 (zero) indicates a successful interpolation.

call gridcell_components(varstring)

```
character(len=*), intent(in) :: varstring
```

This is a utility routine that helps identify how many land units,columns, or PFTs are in each gridcell for a particular variable. Helps answer exploratory questions about which gridcells are appropriate to test code. The CLM variable is read from the CLM restart file.

`varstring`	The CLM variable name of interest.

call DART_get_var(ncid, varname, datmat)

```
integer,                   intent(in)  :: ncid
character(len=*),          intent(in)  :: varname
real(r8), dimension(:),    intent(out) :: datmat
real(r8), dimension(:,:), intent(out) :: datmat
```

Reads a 1D or 2D variable of 'any' type from a netCDF file and processes and applies the offset/scale/FillValue attributes correctly.

`ncid`	The netCDF file identifier to an open file. ncid is the output from a nf90_open() call.
`varname`	The name of the netCDF variable of interest. The variables can be integers, floats, or doubles.
`datmat`	The shape of datmat must match the shape of the netCDF variable. Only 1D or 2D variables are currently supported.

model_time = get_model_time( )

```
integer :: get_model_time
```

Returns the valid time of the model state vector.

`model_time`	The valid time of the model state vector.

6.27.9 Files

filename	purpose
input.nml	to read the model_mod namelist
clm_restart.nc	both read and modified by the CLM model_mod
clm_history.nc	read by the CLM model_mod for metadata purposes.
.h1. history files	may be read by the obs_def_tower_mod for observation operator purposes.
dart_log.out	the run-time diagnostic output
dart_log.nml	the record of all the namelists actually USED - contains the default values

6.27.10 References

CLM User's Guide is an excellent reference for CLM.

6.27.11 Private components

N/A

6.28 CM1

6.28.1 Overview

Cloud Model 1 (CM1) version 18 (CM1r18) is compatible with the DART. CM1 is a non-hydrostatic numerical model in Cartesian 3D coordinates designed for the study of micro-to-mesoscale atmospheric phenomena in idealized to semi-idealized simulations.

The CM1 model was developed and is maintained by George Bryan at the National Center for Atmospheric Research (NCAR) Mesoscale and Microscale Meteorology Laboratory (MMM).

The model code is freely available from the CM1 website and must be downloaded and compiled outside of DART.

This model interface and scripting support were created by Luke Madaus. **Thanks Luke!**

6.28.2 namelist.input

Several modifications to the CM1 namelist `namelist.input` are required to produce model output compatible with DART. The values are described here and an example is shown below.

The `namelist.input` file is partitioned into several distinct namelists. These namelists are denoted `&param0`, `&param1`, `&param2`, ... `&param13`.

These namelists start with an ampersand `&` and terminate with a slash `/`. Thus, character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

Using CM1 output files as a prior ensemble state in DART requires each ensemble member to produce a restart file in netCDF format (which requires setting `restart_format=2` in the `&param9` namelist) and these restart files must only contain output at the analysis time (which requires setting `restart_filetype=2` in the `&param9` namelist).

Here is an example configuration of the `&param9` namelist in `namelist.input`:

```
&param9
  restart_format     = 2          restart needs to be netCDF
  restart_filetype   = 2          restart must be the analysis time - ONLY
  restart_file_theta = .true.     make sure theta is in restart file
  restart_use_theta  = .true.
/
```

Important: The only required state variable to be updated is potential temperature (`theta`). Thus two additional settings in the `&param9` namelist – `restart_file_theta = .true.` and `restart_use_theta = .true.` must be set to ensure `theta` is output the CM1 restart files.

Additional state variables that have been tested within DART include:

`ua, va, wa, ppi, u0, v0, u10, v10, t2, th2, tsk, q2, psfc, qv, qc, qr, qi qs, & qg.`

At present, observation times are evaluated relative to the date and time specified in the `&param11` namelist.

Observation locations are specified in meters relative to the domain origin as defined the `iorigin` setting of `&param2`.

6.28.3 About Testing CM1 and DART

There are two sets of scripts in the `shell_scripts` directory. Luke contributed a set written in python, and the DART team had a set written in csh. The csh scripts have not been tested in quite some time, so use with the understanding that they will need work. Those csh scripts and some unfinished python scripts reside in a `shell_scripts/unfinished` directory and should be used with the understanding that they require effort on the part of the user before the scripts will actually work.

6.28.4 Strategy and Instructions for Using the Python Scripts

A List of Prerequisites

1. CM1 is required to use netCDF restart files.
2. A collection of CM1 model states for initial conditions will be available.
3. There is a separate observation sequence file for each assimilation time.
4. The DART *input.nml* file has some required values as defined below.
5. Each time CM1 is advanced, it will start from the same filename, and the restart number in that filename will be 000001 - ALWAYS. That filename will be a link to the most current model state.

Testing a Cycling Experiment

The big picture: three scripts (`setup_filter.py`, `run_filter.py`, and `advance_ensemble.py`) are alternated to configure an experiment, perform an assimilation on a set of restart files, and make the ensemble forecast. Time management is controlled through command-line arguments.

It is required that you have generated the DART executables before you test. The term `{centraldir}` refers to a filesystem and directory that will be used to run the experiment, the working directory. `{centraldir}` should have a lot of capacity, as ensemble data assimilation will require lots of disk. The term `{dart_dir}` will refer to the location of the DART source code.

The data referenced in the directories (the initial ensemble, etc.) are provided as a compressed tar file cm1r18_3member_example_data.tar.gz.

You will have to download the tar file, uncompress it, and modify the scripts to use these directories instead of the example directories in the scripts. You will also have to compile your own cm1 executable.

1. Set some variables in both `shell_scripts/setup_filter.py` and `shell_scripts/advance_ensemble.py` as described below.

2. In the `{dart_dir}/models/cm1/shell_scripts` directory, run:

   ```
   $ ./setup_filter.py -d YYYYmmDDHHMMSS -i
   ```

   where `YYYYmmDDHHMMSS` is the date and time of the first assimilation cycle (the `-i` option indicates this is the initial setup and extra work will be performed). This will create the working directory `{centraldir}`, link in required executables, copy in the initial conditions for each member from some predetermined location, copy in the observation sequence file for this assimilation time from some predetermined location, modify namelists, and build a queue submission script in the `{centraldir}`: `run_filter.py`.

3. Change into `{centraldir}` and verify the contents of `run_filter.py`. Ensure the assimilation settings in `input.nml` are correct. Once you are satisfied, submit `run_filter.py` to the queue to perform an assimilation.

4. After the assimilation job completes, check to be sure that the assimilation completed successfully, and the archived files requested in the `setup_filter.py` `files_to_archive` variable are in `{centraldir}/archive/YYYYmmDDHHMMSS`.

5. Change into `{dart_dir}/models/cm1/shell_scripts` and advance the ensemble to the next assimilation time by running:

   ```
   $ ./advance_ensemble.py -d YYYYmmDDHHMMSS -l nnnn
   ```

   where `YYYYmmDDHHMMSS` is the date of the COMPLETED analysis (the start time for the model) and `nnnn` is the length of model integration in seconds (the forecast length). (The forecast length option is specified by 'hypen ell' - the lowercase letter L, not the number one.) `advance_ensemble.py` will submit jobs to the queue to advance the ensemble.

6. After all ensemble members have successfully completed, run:

   ```
   $ ./setup_filter.py -d YYYYmmDDHHMMSS
   ```

   where $YYYYmmDDHHMMSS$ is the **new** current analysis time. Note the $-i$ flag is NOT used here, as we do not need to (should not need to!) re-initialize the entire directory structure.

7. Change into `{centraldir}` and run:

   ```
   $ ``run_filter.py``
   ```

   to perform the assimilation.

8. Go back to step 4 and repeat steps 4-7 for each assimilation cycle until the end of the experiment.

Within the `setup_filter.py` and `advance_ensemble.py` scripts, the following variables need to be set between the "BEGIN USER-DEFINED VARIABLES" and "END USER-DEFINED VARIABLES" comment blocks:

`jobname`

A name for this experiment, will be included in the working directory path.

`ens_size`

Number of ensemble members.

`restart_filename`

The filename for each ensemble member's restart. Highly recommended to leave this as `cm1out_rst_000001.nc`

`window_mins`

The assimilation window width (in minutes) for each assimilation cycle.

`copy`

The copy command with desired flags for this system.

`link`

The link command with desired flags for this system.

`remove`

The remove command with desired flags for this system.

`files_to_archive`

A list of DART output files to archive for each assimilation cycle. Note that any inflation files generated are automatically carried over.

`centraldir`

Directory (which will be created if `setup_filter.py` is run in intialization mode) where the assimilation and model advances will take place. Should be on a system with enough space to allow for several assimilation cycles of archived output.

`dart_dir`

Path to the cm1 subdirectory of DART.

`cm1_dir`

Path to the cm1 model executable (*cm1.exe*)

`icdir`

Path to the ensemble of initial conditions. It is assumed that within this directory, each ensemble member has a subdirectory (*m1*, *m2*, *m3*, . . .) that contains:

- a restart file for cm1 at the desired start time and having the filename defined in `restart_filename` above
- a `namelist.input` file compatible with the generation of that restart file.

`obsdir`

Path to a directory containing observation sequence files to be assimilated. It is assumed that the observation sequence files are named following the convention `YYYYmmDDHHMMSS_obs_seq.prior`, where the date of the analysis time whose observations are contained in that file is the first part of the file name.

setup_filter.py and advance_ensemble.py assume that mpi queue submissions are required to run cm1.exe and filter. These variables control how that is handled.

queue_system

The name of the queueing system

mpi_run_command

The command used in a submitted script to execute an mpi task in the queue, including any required flags

queue_sub_command

The command used to submit a script to the queue

job_sub_info

A dictionary of all flags required to execute a job in the queue, with the key being the flag and the value being the variable. e.g. {'-P' : 'PROJECT CODE HERE', '-W' : '00:20'}, etc.

6.28.5 Namelist

The &model_nml namelist is read from the input.nml file. Again, namelists start with an ampersand & and terminate with a slash /. Character strings that contain a / must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   assimilation_period_days      = 0
   assimilation_period_seconds   = 21600
   model_perturbation_amplitude = 0.2
   cm1_template_file             = 'null'
   calendar                      = 'Gregorian'
   periodic_x                    = .true.
   periodic_y                    = .true.
   periodic_z                    = .false.
   debug                         = 0
   model_variables               = ' '
/
```

Description of each namelist entry

Item	Type	Description
assimilation_period_[days,seconds]	integer	This specifies the width of the assimilation window. The current model time is used as the center time of the assimilation window. All observations in the assimilation window are assimilated. BEWARE: if you put observations that occur before the beginning of the assimilation_period, DART will error out because it cannot move the model 'back in time' to process these observations.
model_perturbation_amplitude	real(r8)	unsupported
cm1_template_file	character(len=256)	filename used to read the variable sizes, location metadata, etc.
calendar	character(len=256)	Character string to specify the calendar in use. Usually 'Gregorian' (since that is what the observations use).
model_variables	character(:,5)	Strings that identify the CM1 variables, their DART quantity, the minimum & maximum possible values, and whether or not the posterior values should be written to the output file. The DART QUANTITY must be one found in the *DART/obs_kind/obs_kind_mod.f90* AFTER it gets built by *preprocess*. *model_variables(:,1)* Specifies the CM1 variable name in the netCDF file. *model_variables(:,2)* Specifies the DART quantity for that variable. *model_variables(:,3)* Specifies a minimum bound (if any) for that variable. *model_variables(:,4)* Specifies a maximum bound (if any) for that variable. *model_variables(:,5)* Specifies if the variable should be updated in the restart file. The value may be "UPDATE" or anything else.
periodic_x	logical	a value of *.true.* means the 'X' dimension is periodic.
periodic_y	logical	a value of *.true.* means the 'Y' dimension is periodic.
periodic_z	logical	unsupported
debug	integer	switch to control the amount of run-time output is produced. Higher values produce more output. 0 produces the least.

Note: The values above are the default values. A more realistic example is shown below and closely matches the values in the default `input.nml`.

```
&model_nml
   assimilation_period_days     = 0
   assimilation_period_seconds  = 60
   cm1_template_file            = 'cm1out_rst_000001.nc'
   calendar                     = 'Gregorian'
   periodic_x                   = .true.
   periodic_y                   = .true.
   periodic_z                   = .false.
   debug                        = 0
   model_variables = 'ua'   , 'QTY_U_WIND_COMPONENT'      , 'NULL', 'NULL', 'UPDATE',
                     'va'   , 'QTY_V_WIND_COMPONENT'      , 'NULL', 'NULL', 'UPDATE',
                     'wa'   , 'QTY_VERTICAL_VELOCITY'     , 'NULL', 'NULL', 'UPDATE',
                     'theta', 'QTY_POTENTIAL_TEMPERATURE' , 0.0000, 'NULL', 'UPDATE',
                     'ppi'  , 'QTY_PRESSURE'              , 'NULL', 'NULL', 'UPDATE',
                     'u10'  , 'QTY_10M_U_WIND_COMPONENT'  , 'NULL', 'NULL', 'UPDATE',
                     'v10'  , 'QTY_10M_V_WIND_COMPONENT'  , 'NULL', 'NULL', 'UPDATE',
                     't2'   , 'QTY_2M_TEMPERATURE'        , 0.0000, 'NULL', 'UPDATE',
                     'th2'  , 'QTY_POTENTIAL_TEMPERATURE' , 0.0000, 'NULL', 'UPDATE',
                     'tsk'  , 'QTY_SURFACE_TEMPERATURE'   , 0.0000, 'NULL', 'UPDATE',
                     'q2'   , 'QTY_SPECIFIC_HUMIDITY'     , 0.0000, 'NULL', 'UPDATE',
                     'psfc' , 'QTY_SURFACE_PRESSURE'      , 0.0000, 'NULL', 'UPDATE',
                     'qv'   , 'QTY_VAPOR_MIXING_RATIO'    , 0.0000, 'NULL', 'UPDATE',
                     'qc'   , 'QTY_CLOUD_LIQUID_WATER'    , 0.0000, 'NULL', 'UPDATE',
                     'qr'   , 'QTY_RAINWATER_MIXING_RATIO', 0.0000, 'NULL', 'UPDATE',
                     'qi'   , 'QTY_CLOUD_ICE'             , 0.0000, 'NULL', 'UPDATE',
                     'qs'   , 'QTY_SNOW_MIXING_RATIO'     , 0.0000, 'NULL', 'UPDATE',
                     'qg'   , 'QTY_GRAUPEL_MIXING_RATIO'  , 0.0000, 'NULL', 'UPDATE'
/
```

6.29 COAMPS Nest

6.29.1 Overview

An updated version of the COAMPS model interfaces and scripts.

This interface was contributed by Alex Reinecke of the Naval Research Lab-Monterey.

The primary differences from the original COAMPS model code are:

- the ability to assimilate nested domains
- assimilates real observations
- a simplified way to specify the state vector
- I/O COAMPS data files
- extensive script updates to accommodate additional HPC environments

6.30 COAMPS

6.30.1 Overview

DART interface module for the Coupled Ocean / Atmosphere Mesoscale Prediction (COAMPS ®) model. The 16 public interfaces listed here are standardized for all DART compliant models. These interfaces allow DART to advance the model, get the model state and metadata describing this state, find state variables that are close to a given location, and do spatial interpolation for a variety of variables required in observational operators.

The following model description is taken from the COAMPS overview web page:

> "The Coupled Ocean/Atmosphere Mesoscale Prediction System (COAMPS) has been developed by the Marine Meteorology Division (MMD) of the Naval Research Laboratory (NRL). The atmospheric components of COAMPS, described below, are used operationally by the U.S. Navy for short-term numerical weather prediction for various regions around the world. The atmospheric portion of COAMPS represents a complete three-dimensional data assimilation system comprised of data quality control, analysis, initialization, and forecast model components. Features include a globally relocatable grid, user-defined grid resolutions and dimensions, nested grids, an option for idealized or real-time simulations, and code that allows for portability between mainframes and workstations. The nonhydrostatic atmospheric model includes predictive equations for the momentum, the non-dimensional pressure perturbation, the potential temperature, the turbulent kinetic energy, and the mixing ratios of water vapor, clouds, rain, ice, graupel, and snow, and contains advanced parameterizations for boundary layer processes, precipitation, and radiation. The distributed version of the COAMPS code that can be downloaded from the web site has been designed to use the message-passing interface (MPI), OpenMP directives, and horizontal domain decomposition to achieve parallelism. The code is capable of executing efficiently across vector, parallel, or symmetric muti-processor (SMP) machines by simply changing run-time options."

6.30.2 Other modules used

```
types_mod
time_manager_mod
threed_sphere/location_mod
utilities_mod
obs_kind_mod
random_seq_mod
netcdf
typesizes
coamps_grid_mod
coamps_interp_mod
coamps_restart_mod
coamps_util_mod
```

6.30.3 Public interfaces

use model_mod, only :	get_model_size
	get_state_meta_data
	model_interpolate
	get_model_time_step
	static_init_model
	nc_write_model_atts
	nc_write_model_vars
	pert_model_state
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	ens_mean_for_model
	adv_1step
	end_model
	init_time
	init_conditions

The last 4 interfaces are only required for low-order models where advancing the model can be done by a call to a subroutine. The COAMPS model only advances by executing the coamps program. Thus the last 4 interfaces only appear as stubs in this module.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

model_size = get_model_size( )

```
integer :: get_model_size
```

Returns the length of the model state vector as an integer. This includes all nested domains.

`model_size`	The length of the model state vector.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,             intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,   intent(out) ::  var_type
```

Returns metadata about a given element, indexed by index_in, in the model state vector. The location defines where the state variable is located while the type of the variable (for instance temperature, or u wind component) is returned by var_type. The integer values used to indicate different variable types in var_type are themselves defined as public interfaces to model_mod if required.

index_in	Index of state vector element about which information is requested.
location	Returns location of indexed state variable. The location should use a location_mod that is appropriate for the model domain. For realistic atmospheric models, for instance, a three-dimensional spherical location module that can represent height in a variety of ways is provided.
var_type	Returns the type of the indexed state variable as an optional argument.

call model_interpolate(x, location, obs_kind, obs_val, istatus)

```
real(r8), dimension(:), intent(in)  :: x
type(location_type),    intent(in)  :: location
integer,                  intent(in)  ::  obs_kind
real(r8),               intent(out) :: obs_val
integer,                intent(out) :: istatus
```

Given model state, returns the value of observation type interpolated to a given location by a method of the model's choosing. All observation kinds defined in obs_kind_mod are supported. In the case where the observational operator is not defined at the given location (e.g. the observation is below the model surface or outside the domain), obs_val is returned as -888888.0 and istatus = 1. Otherwise, istatus = 0. The interpolation is performed in the domain with the highest resolution containing the observation.

x	A model state vector.
location	Location to which to interpolate.
obs_kind	Integer indexing which type of observation is to be interpolated.
obs_val	The interpolated value from the model.
istatus	Integer flag indicating the result of the interpolation.

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

Returns the model base time step as a time_type. For now this is set to 1 minute.

`var`	Smallest time step of model.

call static_init_model()

Used for runtime initialization of the model. This is the first call made to the model by any DART compliant assimilation routine. It reads the model namelist parameters, initializes the pressure levels for the state vector, and generates the location data for each member of the state.

ierr = nc_write_model_atts(ncFileId)

```
integer              ::  nc_write_model_atts
integer, intent(in)  ::  ncFileId
```

Function to write model specific attributes to a netCDF file. At present, DART is using the NetCDF format to output diagnostic information. This is not a requirement, and models could choose to provide output in other formats. This function writes the metadata associated with the model to a NetCDF file opened to a file identified by ncFileID.

`ncFileId`	Integer file descriptor opened to NetCDF file.
`ierr`	Returned error code.

ierr = nc_write_model_vars(ncFileID, statevec, copyindex, timeindex)

```
integer                                 ::  nc_write_model_vars
integer,                   intent(in)  ::  ncFileID
real(r8), dimension(:),    intent(in)  ::  statevec
integer,                   intent(in)  ::  copyindex
integer,                   intent(in)  ::  timeindex
```

Writes a copy of the state variables to a NetCDF file. Multiple copies of the state for a given time are supported, allowing, for instance, a single file to include multiple ensemble estimates of the state.

`ncFileID`	Integer file descriptor opened to NetCDF file.
`statevec`	State vector.
`copyindex`	Integer index to which copy is to be written.
`timeindex`	Integer index of which time in the file is being written.
`ierr`	Returned error code.

call pert_model_state(state, pert_state, interf_provided)

```
real(r8), dimension(:),   intent(in)    ::  state
real(r8), dimension(:),   intent(out)   ::  pert_state
logical,                  intent(out)   ::  interf_provided
```

Given a model state, produces a perturbed model state. This is used to generate initial ensemble conditions perturbed around some control trajectory state when one is preparing to spin-up ensembles. In the COAMPS interface, this can be done three different ways:

- No perturbation
- Uniform perturbation - each element of the field has the same additive perturbation
- Individual perturbation - each element of the field has a different additive perturbation The perturbation magnitude and option are supplied out of the dynamic restart vector definition - this allows us to supply a variance appropriate for each type of variable at each level.

`state`	State vector to be perturbed.
`pert_state`	Perturbed state vector is returned.
`interf_provided`	Returns .true. for this model.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
real(r8),             intent(in)    :: maxdist
```

Pass-through to the 3-D sphere locations module. See get_close_maxdist_init() for the documentation of this subroutine.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(num)
```

Pass-through to the 3-D sphere locations module. See get_close_obs_init() for the documentation of this subroutine.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, num_close, close_ind [, dist])

```
type(get_close_type), intent(in)  :: gc
type(location_type),  intent(in)  :: base_obs_loc
integer,              intent(in)  :: base_obs_kind
type(location_type),  intent(in)  :: obs(:)
```

(continues on next page)

(continued from previous page)

```
integer,               intent(in)  :: obs_kind(:)
integer,               intent(out) :: num_close
integer,               intent(out) :: close_ind(:)
real(r8), optional,    intent(out) :: dist(:)
```

Pass-through to the 3-D sphere locations module. See get_close_obs() for the documentation of this subroutine.

call ens_mean_for_model(ens_mean)

```
real(r8), dimension(:), intent(in)  :: ens_mean
```

A local copy is available here for use during other computations in the model_mod code.

ens_mean	Ensemble mean state vector

call adv_1step(x, time)

```
real(r8), dimension(:),   intent(inout) ::  x
type(time_type),          intent(in)    ::  time
```

This operation is not defined for the COAMPS model. This interface is only required if `synchronous' model state advance is supported (the model is called directly as a Fortran90 subroutine from the assimilation programs). This is generally not the preferred method for large models and a stub for this interface is provided for the COAMPS model.

x	State vector of length model_size.
time	Gives time of the initial model state. Needed for models that have real time state requirements, for instance the computation of radiational parameters. Note that DART provides a time_manager_mod module that is used to support time computations throughout the facility.

call end_model( )

Called when use of a model is completed to clean up storage, etc. A stub is provided for the COAMPS model.

call init_time(i_time)

```
type(time_type),        intent(in)  ::  i_time
```

Returns the time at which the model will start if no input initial conditions are to be used. This is frequently used to spin-up models from rest, but is not meaningfully supported for the COAMPS model.

call init_conditions(x)

```
real(r8), dimension(:), intent(out) ::  x
```

Returns default initial conditions for model; generally used for spinning up initial model states. For the COAMPS model just return 0's since initial state is always to be provided from input files.

x	Model state vector.

6.30.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
  cdtg = '2006072500',
  y_bound_skip = 3,
  x_bound_skip = 3,
  need_mean = .false.,
/
```

Item	Type	Description
cdtg	character(len=10)	Date/time group.
x_bound_skip, y_bound_skip	integer	Number of x and y boundary points to skip when perturbing the model state.
need_mean	logical	Does the forward operator computation need the ensemble mean?

6.30.5 Files

filename	purpose
input.nml	to read the model_mod namelist
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.30.6 References

The COAMPS registration web site is http://www.nrlmry.navy.mil/coamps-web/web/home and COAMPS is a registered trademark of the Naval Research Laboratory.

6.30.7 Private components

N/A

6.31 ECHAM

6.31.1 Overview

ECHAM is the atmospheric general circulation component of the Max Planck Institute Earth System Model (MPI-ESM). It was originally branched from the numerical weather prediction model developed by the European Centre for Medium-Range Weather Forecasts (ECMWF) in the late 1980's and is developed and supported by the Max Planck Institute for Meteorology in Hamburg, Germany. Thus the ECHAM acronym is comprised of EC from ECMWF, H for Hamburg and AM for atmospheric model.

There are several DART users who have working DART interface code to ECHAM. If you are interested in running DART with this model please contact the DART group at dart@ucar.edu for more information. We currently do not have a copy of the model_mod interface code nor any of the scripting required to run an assimilation, but we may be able to put you in contact with the right people to get it.

6.32 FESOM

The Finite Element Sea-ice Ocean Model (FESOM) is an unstructured mesh global ocean model using finite element methods to solve the hydro-static primitive equations with the Boussinesq approximation (Danilov et al., 2004[1]; Wang et al., 2008[2]). FESOM v1.4 is interfaced with DART by Aydoğdu et al. (2018a)[3] using a regional implementation in Turkish Straits System (Gürses et al. 2016[4], Aydoğdu et al. 2018b[5]).

[1] Danilov, S., Kivman, G., and Schröter, J.: A finite-element ocean model: principles and evaluation, Ocean Modell., 6, 125–150, 2004.

[2] Wang, Q., Danilov, S., and Schröter, J.: Finite element ocean circulation model based on triangular prismatic elements, with application in studying the effect of topography representation, J. Geophys. Res.-Oceans (1978–2012), 113, C05015, doi:10.1029/2007JC004482, 2008.

[3] Aydoğdu, A., Hoar, T. J., Vukicevic, T., Anderson, J. L., Pinardi, N., Karspeck, A., Hendricks, J., Collins, N., Macchia, F., and Özsoy, E.: OSSE for a sustainable marine observing network in the Sea of Marmara, Nonlin. Processes Geophys., 25, 537-551, doi:10.5194/npg-25-537-2018, 2018a.

[4] Gürses, Ö., Aydoğdu, A., Pinardi, N., and Özsoy, E.: A finite element modeling study of the Turkish Straits System, in: The Sea of Marmara – Marine Biodiversity, Fisheries, Conservations and Governance, edited by: Özsoy E., Çağatay, M. N., Balkis, N., and Öztürk, B., TUDAV Publication, 169–184, 2016.

[5] Aydoğdu, A., Pinardi, N., Özsoy, E., Danabasoglu, G., Gürses, Ö., and Karspeck, A.: Circulation of the Turkish Straits System under interannual atmospheric forcing, Ocean Sci., 14, 999-1019, doi:10.5194/os-14-999-2018, 2018b.

There is a recent version of the model called the Finite-volumE Sea ice–Ocean Model (FESOM2, Danilov et al. 2017[6]). A version for coastal applications FESOM-C v.2 (Androsov et al., 2019[7]) has also been published.

The FESOM V1.4 source code can be downloaded from https://fesom.de/models/fesom14

The FESOM/DART interfaces, diagnostics and support scripting were contributed by **Ali Aydoğdu**. Thanks Ali!

6.32.1 Overview

model_mod.f90

A module called *fesom_modules* is provided to pass the information from FESOM to DART. fesom_modules.f90 includes fortran routines adopted from FESOM v1.4 to read the mesh, set the variables and dimensions of the arrays. *fesom_modules* should have access to *nod2d.out*, *nod3d.out*, *elem2d.out*, *elem3d.out*, *aux3d.out*, *depth.out* and *m3d.ini* mesh files.

Forward operators use an interpolation using the closest model node in the horizontal, given that the application in Aydoğdu et al. (2018a) uses a very high-resolution mesh. In the vertical, a linear interpolation is performed between two enclosing model layers. Interpolation in model_interpolate routine can be improved, if needed.

Note that because the FESOM-native code explicitly types reals, the DART mechanism of being able to run in reduced precision by defining real(r8) to be the same as real(r4) via 'types_mod.f90' is not supported.

Workflow

1. *environment.load* Must be modified to contain the specifics of an experiment. This file is sourced by every other script below.
2. *experiment.launch* Takes the information from **environment.load** and creates runnable scripts from the template script files. This also initiates the first cycle of the experiment.

   2.1. *ensemble.sh*

   2.1.1. *initialize.template* (first cycle only)

   2.1.2. *advance_model.template* (job array to advance the ensemble)

   2.1.3. *check_ensemble.sh* (if all goes well, assimilate)

   2.1.3.1. *filter.template* (assimilate)

   2.1.3.2. *finalize.sh* if all goes well and experiment is not finished . . . continue to 2.1

Shell Scripts

Shell scripts are written in bash for LSF queuing system. They should be modified to work with others such as SLURM. FESOM executables are called externally detached from DART therefore no need for an advance model.

[6] Danilov, S., Sidorenko, D., Wang, Q., and Jung, T.: The Finite-volumE Sea ice–Ocean Model (FESOM2), Geosci. Model Dev., 10, 765-789, doi:10.5194/gmd-10-765-2017, 2017.

[7] Androsov, A., Fofonova, V., Kuznetsov, I., Danilov, S., Rakowsky, N., Harig, S., Brix, H., and Wiltshire, K. H.: FESOM-C v.2: coastal dynamics on hybrid unstructured meshes, Geosci. Model Dev., 12, 1009-1028, doi:10.5194/gmd-12-1009-2019, 2019.

Script	Queue	Definition
environment.load	serial	Includes environment variables, relevant directories, experiment specifications. This file is sourced by every other script below.
experiment.launch	serial	Main script which modifies `ensemble.sh` and calls `ensemble.${EXPINFO}.sh`. An experiment-specific summary which should be modified before launching the scripts.
ensemble.sh	serial	Calls and submits `initialize.template`, `advance_model.template` `check_ensemble.sh` one after the other.
initialize.template	serial	Called only once at the beginning of the experiment. Sets the experiment directory, copies initial ensemble, namelists.
advance_model.template	parallel	Submits a job array for all ensemble members.
check_ensemble.sh	serial	Checks if the forwarding for all members is finished. If so, first calls `filter.template` and then calls `finalize.sh` to conclude current assimilation cycle.
filter.template	parallel	Runs the filter to perform the assimilation.
finalize.sh	serial	Checks if the whole experiment is finished. If so, stops. Otherwise, resubmits `ensemble.${EXPINFO}.sh` for the next assimilation cycle.

Diagnostics

A toolbox for diagnostics is provided. Some are written for a specific regional application using Ferrybox observations of temperature and salinity. However, it shouldn't be difficult to add new tools following the present ones. A fortran toolbox post-processes the FESOM outputs and visualization is done using Generic Mapping Tools (GMT). DART post-processed netCDF outputs are visualized using FERRET. Please see the expanded description inside each source file.

Directory	code file	description
src/		
	fesom_post_main.F90	main fortran routine calling each tool selected in the namelist
	fesom_ocean_mod.F90	ocean diagnostic routines
	fesom_dart_mod.F90	DART diagnostic output routines
	fesom_forcing_mod.F90	forcing diagnostic routines
	fesom_observation_mod.F90	observation diagnostic routines
	gen_input.F90	routines for I/O (adapted from FESOM)
	gen_modules_clock.F90	routines for timing (adapted from FESOM)
	gen_modules_config.F90	routines for configuration (adapted from FESOM)
	mesh_read.F90	routines for reading the mesh (adapted from FESOM)
	Makefile	Makefile (adapted from FESOM) but reads DART environment
	oce_dens_press.F90	routines to compute density and pressure (adapted from FESOM)
	oce_mesh_setup.F90	routines for mesh setup (adapted from FESOM)
	oce_modules.F90	routines for ocean modules (adapted from FESOM)
	random_perturbation.F90	random perturbation to observation sampling
	utilities.F90	various utilities
script/		
	compute_ensemble_mean	computes ensemble mean and extracts a transect or level
	compute_increment	computes increment using DART diagnostic output
	compute_NR_diff	computes the difference between a nature run and the ensemble prior mean
	dart_obs_seq_diag	DART observation-space statistics from `obs_epoch.nc` and `obs_diag.nc`

continues on next page

Table 1 – continued from previous page

Directory	code file	description
	dart.postproc.env	DART environment variables
	fesom.postproc.env	FESOM environment variables
	observe_nature_run	creates synthetic observations from a nature run
	transect_daily_mean	extracts and plots a transect of an individual ensemble member
	zlevel_daily_mean	extracts and plots a level of an individual ensemble member
gmt/		
	plot_ensemble_mean.gmt	plots ensemble mean created by `compute_ensemble_mean`
	plot_increment.gmt	plots increment created by `compute_increment`
	plot_NR_diff.gmt	plots difference created by `compute_NR_diff`
	transect_daily_mean.gmt	plots transects created by `transect_daily_mean`
	zlevel_yearly_mean.gmt	plots levels created by `zlevel_daily_mean`
ferret/		
	frt.obs_diag_TeMPLaTe.jnl	plot DART diags created by `dart_obs_seq_diag`
	frt.obs_epoch_TeMPLaTe.jnl	plot DART diags created by `dart_obs_seq_diag`

6.32.2 References

6.33 GITM

6.33.1 Overview

The Global Ionosphere Thermosphere Model (GITM) is a 3-dimensional spherical code that models the Earth's thermosphere and ionosphere system using a stretched grid in latitude and altitude.

The **GITM** interface for **Data Assimilation Research Testbed (DART)** is under development. If you wish to use GITM, you are urged to contact us. The original scripts were configured to run on the University of Michigan machine **NYX** using the Portable Batch System (**PBS**). We have attempted to extend the scripts to work with both PBS and LSF and are only partway through the process.

DART does not come with the GITM code. You need to get that on your own. The normal procedure of building GITM creates some resource files that are subsequently needed by DART - just to compile. These include:

1. `GITM/src/ModSize.f90`
2. `GITM/src/ModTime.f90` and, for example,
3. `GITM/src/ModEarth.f90`

GITM uses binary files for their restart mechanisms, so no metadata is available to confirm the number and order of fields in the file. Care must be used to make sure the namelist-controlled set of variables to be included in the DART state vector is consistent with the restart files. Each variable must also correspond to a DART "KIND"; required for the DART interpolate routines.

For example, this configuration of `input.nml` is nowhere close to being correct:

```
&model_nml
   gitm_state_variables = 'Temperature',       'QTY_TEMPERATURE',
                          'eTemperature',      'QTY_TEMPERATURE_ELECTRON',
                          'ITemperature',      'QTY_TEMPERATURE_ION',
                          'iO_3P_NDensityS',   'QTY_DENSITY_NEUTRAL_O3P',
                          'iO2_NDensityS',     'QTY_DENSITY_NEUTRAL_O2',
                          'iN2_NDensityS',     'QTY_DENSITY_NEUTRAL_N2',
```

(continues on next page)

(continued from previous page)

```
                                    ...                    ...
/
```

These variables are then adjusted to be consistent with observations and stuffed back into the same netCDF restart files. Since DART is an ensemble algorithm, there are multiple restart files for a single restart time: one for each ensemble member. Creating the initial ensemble of states is an area of active research.

DART reads grid information for GITM from several sources. The `UAM.in` file specifies the number of latitudes/longitudes per block, and the number of blocks comes from the `GITM2/src/ModSize.f90` module. Internal to the DART code, the following variables exist:

Item	Type	Description
LON(:)	real(r8)	longitude array [0, 360)
LAT(:)	real(r8)	latitude array (-90,90)
ALT(:)	real(r8)	altitude array (0,~inf)
NgridLon	integer	the length of the longitude array
NgridLat	integer	the length of the latitude array
NgridAlt	integer	the length of the altitude array

6.33.2 Compiling

GITM has been sucessfully tested with DART using the `gfortran` compiler, version `4.2.3`. The DART components were built with the following `mkmf.template` settings.

```
FC = gfortran
LD = gfortran
NETCDF = /Users/thoar/GNU
INCS = -I${NETCDF}/include
LIBS = -L${NETCDF}/lib -lnetcdf -lcurl -lhdf5_hl -lhdf5 -lz  -lm
FFLAGS = -O0 -fbounds-check -frecord-marker=4 -ffpe-trap=invalid $(INCS)
LDFLAGS = $(FFLAGS) $(LIBS)
```

6.33.3 Converting Between DART Files and GITM Restart Files

The binary GITM files contain no metadata, so care is needed when converting between DART state variables and GITM files.

There are two programs - both require the list of GITM variables to use in the DART state vector: the `&model_nml:gitm_state_variables` variable in the `input.nml` file.

`gitm_to_dart.f90`	converts a set of GITM restart files (there is one restart file per block) *bxxxx.rst* into a DART-compatible file normally called *dart_ics* . We usually wind up linking to this static filename.
`dart_to_gitm.f90`	inserts the DART output into existing GITM restart files. There are two different types of DART output files, so there is a namelist option to specify if the DART file has two time records or just one. If there are two, the first one is the 'advance_to' time, followed by the 'valid_time' of the ensuing state. If there is just one, it is the 'valid_time' of the ensuing state. *dart_to_gitm* determines the GITM restart file name from the *input.nml model_nml:gitm_restart_dirname*. If the DART file contains an 'advance_to' time, *dart_to_gitm* creates a *DART_GITM_time_control.txt* file which can be used to control the length of the GITM integration.

6.33.4 Simple Test

The simplest way to test the converter is to compile GITM and run a single model state forward using `work/clean.sh`. To build GITM ... download GITM and unpack the code into `DART/models/gitm/GITM2` and run the following commands:

```
$ cd models/gitm/GITM2
$ ./Config.pl -install -compiler=ifortmpif90 -earth
$ make
$ cd ../work
$ ./clean.sh 1 1 0 150.0 170.0 1.0
```

6.33.5 Namelist

We adhere to the F90 standard of starting a namelist with an ampersand `&` and terminating with a slash `/` for all our namelist input. Character strings that contain a `/` **must** be enclosed in quotes to prevent them from prematurely terminating the namelist.

This namelist is read from a file called `input.nml`. This namelist provides control over the assimilation period for the model. All observations within (+/-) half of the assimilation period are assimilated. The assimilation period is the minimum amount of time the model can be advanced, and checks are performed to ensure that the assimilation window is a multiple of the model dynamical timestep.

Sample input.nml Configuration

```
# The list of variables to put into the state vector is here:
# The definitions for the DART kinds are in DART/observations/forward_operators/obs_
↪def*f90
# The order doesn't matter to DART. It may to you.

&model_nml
   gitm_restart_dirname         = 'advance_temp_e1/UA/restartOUT',
   assimilation_period_days     = 0,
   assimilation_period_seconds  = 1800,
   model_perturbation_amplitude = 0.2,
   output_state_vector          = .false.,
   calendar                     = 'Gregorian',
   debug                        = 0,
   gitm_state_variables  = 'Temperature',            'QTY_TEMPERATURE',
                           'eTemperature',           'QTY_TEMPERATURE_ELECTRON',
                           'ITemperature',           'QTY_TEMPERATURE_ION',
                           'iO_3P_NDensityS',        'QTY_DENSITY_NEUTRAL_O3P',
                           'iO2_NDensityS',          'QTY_DENSITY_NEUTRAL_O2',
                           'iN2_NDensityS',          'QTY_DENSITY_NEUTRAL_N2',
                           'iN_4S_NDensityS',        'QTY_DENSITY_NEUTRAL_N4S',
                           'iNO_NDensityS',          'QTY_DENSITY_NEUTRAL_NO',
                           'iN_2D_NDensityS',        'QTY_DENSITY_NEUTRAL_N2D',
                           'iN_2P_NDensityS',        'QTY_DENSITY_NEUTRAL_N2P',
                           'iH_NDensityS',           'QTY_DENSITY_NEUTRAL_H',
                           'iHe_NDensityS',          'QTY_DENSITY_NEUTRAL_HE',
                           'iCO2_NDensityS',         'QTY_DENSITY_NEUTRAL_CO2',
                           'iO_1D_NDensityS',        'QTY_DENSITY_NEUTRAL_O1D',
                           'iO_4SP_IDensityS',       'QTY_DENSITY_ION_O4SP',
                           'iO2P_IDensityS',         'QTY_DENSITY_ION_O2P',
```

(continues on next page)

(continued from previous page)

```
                           'iN2P_IDensityS',         'QTY_DENSITY_ION_N2P',
                           'iNP_IDensityS',          'QTY_DENSITY_ION_NP',
                           'iNOP_IDensityS',         'QTY_DENSITY_ION_NOP',
                           'iO_2DP_IDensityS',       'QTY_DENSITY_ION_O2DP',
                           'iO_2PP_IDensityS',       'QTY_DENSITY_ION_O2PP',
                           'iHP_IDensityS',          'QTY_DENSITY_ION_HP',
                           'iHeP_IDensityS',         'QTY_DENSITY_ION_HEP',
                           'ie_IDensityS',           'QTY_DENSITY_ION_E',
                           'U_Velocity_component',   'QTY_VELOCITY_U',
                           'V_Velocity_component',   'QTY_VELOCITY_V',
                           'W_Velocity_component',   'QTY_VELOCITY_W',
                           'U_IVelocity_component',  'QTY_VELOCITY_U_ION',
                           'V_IVelocity_component',  'QTY_VELOCITY_V_ION',
                           'W_IVelocity_component',  'QTY_VELOCITY_W_ION',
                           'iO_3P_VerticalVelocity', 'QTY_VELOCITY_VERTICAL_O3P',
                           'iO2_VerticalVelocity',   'QTY_VELOCITY_VERTICAL_O2',
                           'iN2_VerticalVelocity',   'QTY_VELOCITY_VERTICAL_N2',
                           'iN_4S_VerticalVelocity', 'QTY_VELOCITY_VERTICAL_N4S',
                           'iNO_VerticalVelocity',   'QTY_VELOCITY_VERTICAL_NO',
                           'f107',                   'QTY_1D_PARAMETER',
                           'Rho',                    'QTY_DENSITY',
      /
```

Description of Each Term in the Namelist

Item	Type	Description
gitm_restart_dirname	character(len=256)	The name of the directory containing the GITM restart files and runtime control information.
assimilation_period_days	integer	The number of days to advance the model for each assimilation.
assimilation_period_seconds	integer	In addition to `assimilation_period_days` the number of seconds to advance the model for each each assimilation.
model_perturbation_amplitude	real(r8)	Reserved for future use.
output_state_vector	logical	The switch to determine the form of the of the state vector in the output netCDF files. If `.true.` the state vector will be output exactly as DART uses it … one long array. If `.false.`, the state vector is parsed into prognostic variables and output that way – much easier to use with 'ncview', for example.
calendar	character(len=32)	Character string specifying the calendar being used by GITM.
debug	integer	The switch to specify the run-time verbosity. • `0` is as quiet as it gets • `> 1` provides more run-time messages • `> 5` provides ALL run-time messages
gitm_state_variables	character(len=NF90_MAX_NAME):: dimension(160)	The table that relates the GITM variables to use to build the DART state vector, and the corresponding DART kinds for those variables.

6.33.6 Files

filename	purpose
input.nml	to read the model_mod namelist
Several GITM source modules: ModConstants, ModSizeGitm, ModEarth . . .	provides grid dimensions, model state, and 'valid_time' of the model state
header.rst, bNNNN.rst	provides the 'valid_time' of the model state and the model state, respectively
true_state.nc	the time-history of the "true" model state from an OSSE
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.33.7 References

NASA's official *GITM* description can be found at their Community Coordinated Modeling Center website.

6.34 Ikeda

6.34.1 Overview

DART interface module for the Ikeda model. The 16 public interfaces are standardized for all DART compliant models. These interfaces allow DART to advance the model, get the model state and metadata describing this state, find state variables that are close to a given location, and do spatial interpolation for model state variables.

The Ikeda model is a 2D chaotic map useful for visualization data assimilation updating directly in state space. There are three parameters: a, b, and mu. The state is 2D, x = [X Y]. The equations are:

```
X(i+1) = 1 + mu * ( X(i) * cos( t ) - Y(i) * sin( t ) )
Y(i+1) =     mu * ( X(i) * sin( t ) + Y(i) * cos( t ) ),
```

where

```
t = a - b / ( X(i)**2 + Y(i)**2 + 1 )
```

Note the system is time-discrete already, meaning there is no delta_t. The system stems from nonlinear optics (Ikeda 1979, Optics Communications). Interface written by **Greg Lawson, CalTech**. Thanks Greg!

> "The initial conditions were generated by observing state variable 1 with an enormous (~1,000,000.0) observation error variance. The observation was defined to be taken at day=0, seconds = 0. `create_fixed_network_sequence` was run to create a sequence with 3000 hourly observations starting at day=0, seconds =0. The initial conditions for filter can accomodate 100 ensemble members."

6.34.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   a  = 0.40,
   b  = 6.00,
   mu = 0.83,
   time_step_days      = 0,
   time_step_seconds   = 3600,
   output_state_vector = .true.
/
```

Item	Type	Description
a	real(r8)	Model parameter.
b	real(r8)	Model parameter.
mu	real(r8)	Model parameter.
time_step_days	integer	Model advance time in days.
time_step_seconds	integer	Model advance time in seconds.
output_state_vector	logical	If true, output the state vector data to the diagnostic files as a single 1D array. If false, break up output data into logical model variables.

6.34.3 Other modules used

```
types_mod
time_manager_mod
oned/location_mod
utilities_mod
```

6.34.4 Public interfaces

use model_mod, only :	get_model_size
	adv_1step
	get_state_meta_data
	model_interpolate
	get_model_time_step
	static_init_model
	end_model
	init_time
	init_conditions
	nc_write_model_atts
	nc_write_model_vars
	pert_model_state
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	ens_mean_for_model

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

model_size = get_model_size( )

```
integer :: get_model_size
```

Returns the length of the model state vector as an integer. This is fixed at 2 for this model.

`model_size`	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

Advances the model for a single time step. The time associated with the initial model state is also input although it is not used for the computation.

`x`	State vector of length model_size.
`time`	Unused in this model.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,             intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,   intent(out) ::  var_type
```

Returns the location of the given index, and a dummy integer as the var_type.

`index_in`	Index of state vector element about which information is requested.
`location`	Returns location of indexed state variable. The location should use a location_mod that is appropriate for the model domain. For realistic atmospheric models, for instance, a three-dimensional spherical location module that can represent height in a variety of ways is provided.
var_type	Returns the type of the indexed state variable as an optional argument.

call model_interpolate(x, location, itype, obs_val, istatus)

```
real(r8), dimension(:), intent(in)  :: x
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: itype
real(r8),               intent(out) :: obs_val
integer,                intent(out) :: istatus
```

A NULL INTERFACE in this model. Always returns istatus = 0.

`x`	A model state vector.
`location`	Location to which to interpolate.
`itype`	Integer indexing which type of state variable is to be interpolated. Can be ignored for low order models with a single type of variable.
`obs_val`	The interpolated value from the model.
`istatus`	Quality control information about the observation of the model state.

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

Returns the models base time step, or forecast length, as a time_type. This is settable in the namelist.

`var`	Smallest time step of model.

call static_init_model()

Reads the namelist, defines the 2 initial locations of the state variables, and sets the timestep.

call end_model()

A NULL INTERFACE in this model.

call init_time(time)

```
type(time_type), intent(out) :: time
```

Returns a time of 0.

`time`	Initial model time.

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

Sets 2 initial locations close to the attractor.

`x`	Initial conditions for state vector.

ierr = nc_write_model_atts(ncFileID)

```
integer             :: nc_write_model_atts
integer, intent(in) :: ncFileID
```

Uses the default template code.

`ncFileID`	Integer file descriptor to previously-opened netCDF file.
`ierr`	Returns a 0 for successful completion.

ierr = nc_write_model_vars(ncFileID, statevec, copyindex, timeindex)

```
integer                                :: nc_write_model_vars
integer,                intent(in) :: ncFileID
real(r8), dimension(:), intent(in) :: statevec
integer,                intent(in) :: copyindex
integer,                intent(in) :: timeindex
```

Uses the default template code.

`ncFileID`	file descriptor to previously-opened netCDF file.
`statevec`	A model state vector.
`copyindex`	Integer index of copy to be written.
`timeindex`	The timestep counter for the given state.
`ierr`	Returns 0 for normal completion.

call pert_model_state(state, pert_state, interf_provided)

```
real(r8), dimension(:), intent(in)  :: state
real(r8), dimension(:), intent(out) :: pert_state
logical,                intent(out) :: interf_provided
```

Given a model state, produces a perturbed model state. This particular model does not implement an interface for this and so returns .false. for interf_provided.

`state`	State vector to be perturbed.
`pert_state`	Perturbed state vector: NOT returned.
`interf_provided`	Returned false; interface is not implemented.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
real(r8),             intent(in)    :: maxdist
```

Pass-through to the 1-D locations module. See get_close_maxdist_init() for the documentation of this subroutine.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(num)
```

Pass-through to the 1-D locations module. See get_close_obs_init() for the documentation of this subroutine.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, num_close, close_ind [, dist])

```
type(get_close_type), intent(in)  :: gc
type(location_type),  intent(in)  :: base_obs_loc
integer,              intent(in)  :: base_obs_kind
type(location_type),  intent(in)  :: obs(:)
integer,              intent(in)  :: obs_kind(:)
integer,              intent(out) :: num_close
integer,              intent(out) :: close_ind(:)
real(r8), optional,   intent(out) :: dist(:)
```

Pass-through to the 1-D locations module. See get_close_obs() for the documentation of this subroutine.

call ens_mean_for_model(ens_mean)

```
real(r8), dimension(:), intent(in) :: ens_mean
```

A NULL INTERFACE in this model.

`ens_mean`	State vector containing the ensemble mean.

6.34.5 Files

filename	purpose
input.nml	to read the model_mod namelist
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.34.6 References

Ikeda 1979, Optics Communications

6.34.7 Private components

N/A

6.35 LMDZ

6.35.1 Overview

The Laboratoire de Météorologie Dynamique Zoom (LMDZ) model is a global atmospheric model developed by the Institut Pierre-Simon Laplace (IPSL) in France. It serves as the atmospheric component of the IPSL Integrated Climate Model.

The DART interface to LMDZ was primarily developed by **Tarkeshwar Singh** while he was at the Indian Institute of Technology at Delhi. He later moved to the Nansen Environmental and Remote Sensing Center in Bergen, Norway. A detailed description of the LMDZ DART implementation is published in Singh et al. (2015).[1]

Please email Tarkeshwar for documentation beyond what is contained within the repository.

Assimilation with LMDZ is supported in the Lanai release of DART. If you are interested in using LMDZ in the Manhattan version of DART, we encourage you to contact us. We would like to participate!

[1] **Singh, Tarkeshwar, Rashmi Mitta, and H.C. Upadhyaya**, 2015: Ensemble Adjustment Kalman Filter Data Assimilation for a Global Atmospheric Model. *International Conference on Dynamic Data-Driven Environmental Systems Science*, 284-298, doi:10.1007/978-3-319-25138-7_26.

6.35.2 References

6.36 Lorenz 05

6.36.1 Naming History

In earlier versions of DART, this collection of models was referred to as Lorenz 04. Edward Lorenz provided James A. Hansen these model formulations before they had been published, since both Lorenz and Hansen were faculty members at MIT at the time. Hansen developed the DART model interface and incorporated it into the DART codebase in 2004. Thus, within DART, it was named Lorenz 04.

The collection of models was published a year later in Lorenz (2005),[1] thus, within the wider community, the models are typically referred to as Lorenz 05. To reflect this fact, the collection of models was renamed within DART from Lorenz 04 to Lorenz 05 during the Manhattan release.

6.36.2 Overview

Lorenz (2005) provides a fascinating account of the difficulties involved in designing simple models that exhibit chaotic behavior and realistically simulate aspects of atmospheric flow. It presents three models of increasing complexity:

- Model I is a single-scale model, similar to Lorenz (1996),[2] intended to represent the atmosphere at a specific height and latitude.
- Model II is also a single-scale model, similar to Model I, but with spatial continuity in the waves.
- Model III is a two-scale model. It is fundamentally different from the Lorenz 96 two-scale model because of the spatial continuity and the fact that both scales are projected onto a single variable of integration. The scale separation is achieved by a spatial filter and is therefore not perfect (i.e. there is leakage).

Model II and Model III are implemented in this DART model interface, and the user is free to choose Model II or III by editing the *namelist*. For users interested in Model I, please use Lorenz 96. The slow scale in Model III is Model II, and thus Model II is a deficient form of Model III.

The Lorenz 05 model has a `work/workshop_setup.csh` script that compiles and runs an example. This example may be used anywhere in the DART_tutorial to explore multiscale dynamics and to provide insight into model/assimilation behavior. The example **may or may not** result in good (*or even decent!*) results!

Model Formulation

For Lorenz 05, DART to advances the model, gets the model state and metadata describing this state, finds state variables that are close to a given location, and does spatial interpolation for model state variables.

[1] Lorenz, Edward N., 2005: Designing Chaotic Models. *Journal of the Atmospheric Sciences*, **62**, 1574-1587.

[2] Lorenz, Edward N., 1996: Predictability: A Problem Partly Solved. *Seminar on Predictability.* **1**, ECMWF, Reading, Berkshire, UK, 1-18.

6.36.3 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   model_size        = 960,
   forcing           = 15.00,
   delta_t           = 0.001,
   space_time_scale  = 10.00,
   coupling          = 3.00,
   K                 = 32,
   smooth_steps      = 12,
   time_step_days    = 0,
   time_step_seconds = 3600,
   model_number      = 3
/
```

Description of each namelist entry

Contents	Type	Description
model_size	integer	Number of variables in model
forcing	real(r8)	Forcing, F, for model
delta_t	real(r8)	Non-dimensional timestep
space_time_scale	real(r8)	Determines temporal and spatial relationship between fast and slow variables (model III)
coupling	real(r8)	Linear coupling between fast and slow variables (model III)
K	integer	Determines the wavenumber of the slow variables (K=1, smooth_steps=0 reduces model II to Lorenz 96)
smooth_steps	integer	Determines filter length to separate fast and slow scales
time_step_days	integer	Arbitrary real time step days
time_step_seconds	integer	Arbitrary real time step seconds (could choose this for proper scaling)
model_number	integer	2 = single-scale, 3 = 2-scale. (This follows the notation in the paper.)

6.36.4 References

6.37 Lorenz 63

6.37.1 Overview

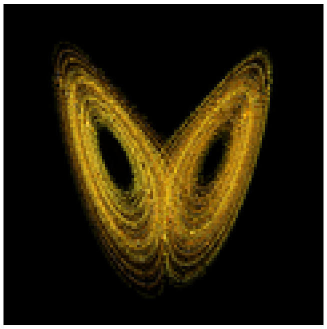
This 3-variable model was described in Lorenz (1963).[1] In Lorenz 63, DART advances the model, gets the model state and metadata describing this state, finds state variables that are close to a given location, and does spatial interpolation

[1] Lorenz, Edward N., 1963: Deterministic Nonperiodic Flow. *Journal of the Atmospheric Sciences*, **20**, 130-141, doi:0.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2

for model state variables. The distinctive part of the model interface is the *namelist*.

Lorenz 63 was developed as a simplified model to study convection rolls in the atmosphere. It is a deceptively simple model – its formulation is simpler than Lorenz's earlier atmospheric models – yet it demonstrates chaotic behavior. It has thus become a widely studied model.

Plotting the location of the x, y, z values as they progress through time traces out the classic 'butterfly' attractor plot which has become an iconic image of chaotic systems:



The system of equations for Lorenz 63 is:

$$\frac{dx}{dt} = \sigma(y - x)\frac{dy}{dt} = x(r - z) - y\frac{dz}{dt} = xy - bz$$

and, within DART, the constants have default values of:

$$\sigma = 10, r = 28, b = 8/3$$

that can be altered by editing the `&model_nml` *namelist* in the `input.nml` file.

This model is an interesting data assimilation test in that different ensemble members may bifurcate over to the other lobe of the attractor on different cycles. Also, as they diverge from each other they do not spread out uniformly in 3D space, but spread along the linear attractor lines.

The Lorenz 63 model has a `work/workshop_setup.csh` script that compiles and runs an example. This example is referenced at various points in the DART_tutorial and is intended to provide insight into model/assimilation behavior. The example **may or may not** result in good (*or even decent!*) results!

`run_lorenz_63.m` is an excellent Matlab tool to explore the behavior of the Lorenz 63 model. It is part of the DART_LAB Tutorial.

6.37.2 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   sigma  = 10.0,
   r      = 28.0,
   b      = 2.6666666666667,
   deltat = 0.01,
   time_step_days = 0,
   time_step_seconds = 3600
   solver = 'RK2'
/
```

Description of each namelist entry

Item	Type	Description
sigma	real(r8)	Model parameter.
r	real(r8)	Model parameter.
b	real(r8)	Model parameter.
deltat	real(r8)	Non-dimensional timestep. This is mapped to the dimensional timestep specified by time_step_days and time_step_seconds.
time_step_days	integer	Number of days for dimensional timestep, mapped to deltat.
time_step_seconds	integer	Number of seconds for dimensional timestep, mapped to deltat.
solver	character(8)	The name of the solver to use. 'RK2', the default, is a two-step Runge-Kutta used in the original Lorenz 63 paper. 'RK4' is the only other option which uses the four-step classic Runge-Kutta method.

6.37.3 References

6.38 Lorenz 84

6.38.1 Overview

This model was described in Lorenz (1984).[1] In Lorenz 84, DART advances the model, gets the model state and metadata describing this state, find states variables that are close to a given location, and does spatial interpolation for model state variables. The distinctive part of the model interfaces is the *namelist*.

The system of equations is:

$$\frac{dx}{dt} = -y^2 - z^2 - ax + aF\frac{dy}{dt} = xy - bxz - y + G\frac{dz}{dt} = bxy + xz - z$$

and, within DART, the model parameters have default values of:

$$a = \frac{1}{4}, b = 4, F = 8, G = \frac{5}{4}$$

that can be altered by editing the `&model_nml` *namelist* in the `input.nml` file.

The Lorenz 84 model has a `work/workshop_setup.csh` script that compiles and runs an example. This example is referenced specifically in Section 7 of the DART_tutorial and is intended to provide insight into model/assimilation behavior. The example **may or may not** result in good (*or even decent!*) results!

The Lorenz 84 model may be used instead of the Lorenz 63 model in many sections of the Tutorial. It has a more complex attractor, is not as periodic as Lorenz 63 and may be more challenging for certain filter variants.

[1] Lorenz, Edward N., 1984: Irregularity: A Fundamental Property of the Atmosphere. *Tellus*, **36A**, 98-110, doi:10.1111/j.1600-0870.1984.tb00230.x

6.38.2 Namelist

The &model_nml namelist is read from the input.nml file. Namelists start with an ampersand & and terminate with a slash /. Character strings that contain a / must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
  a      = 0.25,
  b      = 4.00,
  f      = 8.00,
  g      = 1.25,
  deltat = 0.01,
  time_step_days    = 0,
  time_step_seconds = 3600
/
```

Description of each namelist entry

Item	Type	Description
a	real(r8)	Model parameter.
b	real(r8)	Model parameter.
f	real(r8)	Model parameter.
g	real(r8)	Model parameter.
deltat	real(r8)	Non-dimensional timestep. This is mapped to the dimensional timestep specified by time_step_days and time_step_seconds.
time_step_days	integer	Number of days for dimensional timestep, mapped to deltat.
time_step_seconds	integer	Number of seconds for dimensional timestep, mapped to deltat.

References

6.39 Lorenz 96

6.39.1 Overview

The Lorenz 96 model was first described by Edward Lorenz during a seminar at the European Centre for Medium-Range Weather Forecasts in the Autumn of 1995, the proceedings of which were published as Lorenz (1996)[1] the following year, hence the model is commonly referred to as Lorenz 96.

Lorenz and Emmanuel (1998)[2] describe the model as:

> … consisting of 40 ordinary differential equations, with the dependent variables representing values of some atmospheric quantity at 40 sites spaced equally about a latitude circle. The equations contain quadratic, linear, and constant terms representing advection, dissipation, and external forcing. Numerical integration indicates that small errors (differences between solutions) tend to double in about 2 days. Localized errors tend to spread eastward as they grow, encircling the globe after about 14 days.

[1] Lorenz, Edward N., 1996: Predictability: A Problem Partly Solved. *Seminar on Predictability.* **1**, ECMWF, Reading, Berkshire, UK, 1-18.

[2] Lorenz, Edward N., and Kerry A. Emanuel, 1998: Optimal Sites for Supplementary Weather Observations: Simulations with a Small Model. *Journal of the Atmospheric Sciences*, **55**, 399-414, doi:10.1175/1520-0469(1998)055<0399:OSFSWO>2.0.CO;2

> We have chosen a model with J variables, denoted by:
>
> $$X_1, ..., X_j;$$
>
> in most of our experiments we have let $J = 40$. The governing equations are:
>
> $$dX_j/dt = (X_{j+1} - X_{j-2})X_{j-1} - X_j + F(1)$$
>
> for:
>
> $$j = 1, ..., J.$$
>
> To make Eq. (1) meaningful for all values of *j* we define:
>
> $$X_{-1} = X_{J-1}, X_0 = X_J, \& X_{J+1} = X_1,$$
>
> so that the variables form a cyclic chain, and may be looked at as values of some unspecified scalar meteorological quantity, perhaps vorticity or temperature, at *J* equally spaced sites extending around a latitude circle. Nothing will simulate the atmosphere's latitudinal or vertical extent.

For Lorenz 96, DART advances the model, gets the model state and metadata describing this state, finds state variables that are close to a given location, and does spatial interpolation for model state variables.

The Lorenz 96 model has a `work/workshop_setup.csh` script that compiles and runs an example. This example is referenced at various points in the DART_tutorial and is intended to provide insight into model/assimilation behavior. The example **may or may not** result in good (*or even decent!*) results! Be aware that the `input.nml` file is modified by the `workshop_setup.csh` script.

There are also some excellent Matlab tools to explore the behavior of the Lorenz 96 model, namely `run_lorenz_96.m` and `run_lorenz_96_inf.m`, both of which are part of the DART_LAB Tutorial.

6.39.2 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   model_size        = 40,
   forcing           = 8.00,
   delta_t           = 0.05,
   time_step_days    = 0,
   time_step_seconds = 3600
/
```

Description of each namelist entry

Item	Type	Description
model_size	integer	Number of variables in model.
forcing	real(r8)	Forcing, F, for model.
delta_t	real(r8)	Non-dimensional timestep. This is mapped to the dimensional timestep specified by time_step_days and time_step_seconds.
time_step_days	integer	Number of days for dimensional timestep, mapped to delta_t.
time_step_seconds	integer	Number of seconds for dimensional timestep, mapped to delta_t.

6.39.3 References

6.40 Lorenz 96 2-scale

6.40.1 Overview

The Lorenz 96 2-scale model was first described by Edward Lorenz during a seminar at the European Centre for Medium-Range Weather Forecasts in the Autumn of 1995, the proceedings of which were published as Lorenz (1996)[1] the following year, hence the model is commonly referred to as Lorenz 96.

The model state varies on two separate time scales, one for the X dimension and another in the Y dimension. It is constructed by coupling together two implementations of the Lorenz 96 single-scale model. The constant F term in Lorenz 96 single-scale model is replaced by a term that couples the two scales together.

Lorenz 96 2-scale is a widely studied model because the differing timescales can be viewed as an analog of processes that occur on different time and spatial scales in the atmosphere such as large-scale flow and localized convection. The *references* contain some of the earlier studies including Palmer (2001),[2] Smith (2001),[3] Orrell (2002),[4] Orrel (2003),[5] Vannitsem and Toth (2002),[6] Roulston and Smith (2003),[7] and Wilks (2005).[8]

The Lorenz 96 2-scale model has a `work/workshop_setup.csh` script that compiles and runs an example. This example may be explored in the DART_tutorial and is intended to provide insight into model/assimilation behavior. The example **may or may not** result in good (*or even decent!*) results!

Development History

This DART model interface was developed by Josh Hacker as an adaptation of Tim Hoar's Lorenz 96 implementation. The 2-scale model is the second model described in Lorenz (1996).

6.40.2 Quick Start

To run Lorenz 96 2-scale with its default settings:

1. Ensure you have the correct settings in mkmf.template in `<DARTROOT>/build_templates/mkmf.template`
2. Build the DART executables using the `quickbuild.csh` script in the `./work` directory.
3. Once the executables have been built, the two Perl scripts provided in the `./shell_scripts` directory, `spinup_model.pl` and `run_expt.pl`, can be used to spin up the model and run an experiment.

[1] Lorenz, Edward N., 1996: Predictability: A Problem Partly Solved. *Seminar on Predictability*. **1**, ECMWF, Reading, Berkshire, UK, 1-18.

[2] Palmer, Timothy N., 2001: A nonlinear dynamical perspective on model error: A proposal for non-local stochastic-dynamic parametrization in weather and climate prediction models. *Quarterly Journal of the Royal Meteorological Society*, **127**, 279–304. https://doi.org/10.1002/qj.49712757202

[3] Smith, Leonard A., 2001: Disentangling uncertainty and error: On the predictability of nonlinear systems. *Nonlinear dynamics and statistics,* Alistair I. Mees, Editor, Birkhauser, Boston, USA, 31–64.

[4] Orrell, David, 2002: Role of the metric in forecast error growth: How chaotic is the weather? *Tellus*, **54A**, 350–362.

[5] Orrell, David, 2003: Model error and predictability over different timescales in the Lorenz '96 Systems. *Journal of the Atmospheric Sciences*, **60**, 2219–2228.

[6] Vannitsem, Stéphane and Zoltan Toth, 2002: Short-term dynamics of model errors. *Journal of the Atmospheric Sciences*, **59**, 2594–2604.

[7] Roulston, Mark S. and Leonard A. Smith, 2003: Combining dynamical and statistical ensembles. *Tellus*, **55A**, 16–30.

[8] Wilks, Daniel S., 2005: Effects of stochastic parametrizations in the Lorenz '96 system. *Quarterly Journal of the Royal Meteorological Society*. **131**. 389-407. https://doi.org/10.1256/qj.04.03

6.40.3 Namelist

The model also implements the variant of Smith (2001), which can be invoked by setting `local_y = .true.` in the `&model_nml` namelist in the `input.nml` file.

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   model_size_x        = 36,
   y_per_x             = 10,
   forcing             = 15.00,
   delta_t             = 0.005,
   coupling_b          = 10.0,
   coupling_c          = 10.0,
   coupling_h          = 1.0,
   local_y             = .false.,
   time_step_days      = 0,
   time_step_seconds   = 3600
   template_file       = 'filter_input.nc'
/
```

Description of each namelist entry

Item	Type	Description
model_size_x	integer	Number of variables in x-dimension.
y_per_x	integer	Scaling factor for number of variables in y-dimension compared to x-dimension.
forcing	real(r8)	Forcing, F, for model.
delta_t	real(r8)	Non-dimensional timestep. This is mapped to the dimensional timestep specified by time_step_days and time_step_seconds.
coupling_b	real(r8)	
coupling_c	real(r8)	
coupling_h	real(r8)	
local_y	boolean	
time_step_days	integer	Number of days for dimensional timestep, mapped to delta_t.
time_step_seconds	integer	Number of seconds for dimensional timestep, mapped to delta_t.
template_file	character(len=256)	this in script

References

6.41 Forced Lorenz 96

6.41.1 Overview

The *forced_lorenz_96* model implements the standard Lorenz (1996)[1] equations except that the forcing term, `F`, is added to the state vector and is assigned an independent value at each gridpoint. The result is a model that is twice as big as the standard L96 model. The forcing can be allowed to vary in time or can be held fixed so that the model looks like the standard L96 but with a state vector that includes the constant forcing term. An option is also included

[1] Lorenz, Edward N., 1996: Predictability: A Problem Partly Solved. *Seminar on Predictability*. **1**, ECMWF, Reading, Berkshire, UK, 1-18.

to add random noise to the forcing terms as part of the time tendency computation which can help in assimilation performance. If the random noise option is turned off (see namelist) the time tendency of the forcing terms is 0.

DART state vector composition:

state variables	forcing terms
traditional Lorenz_96 state	*"extended" state*
`indices 1 - 40`	`indices 41 - 80`

The *forced_lorenz_96* model has a `work/workshop_setup.csh` script that compiles and runs an example. This example is referenced in Section 20 of the DART_tutorial and is intended to provide insight into parameter estimation and model/assimilation behavior. Be aware that the `input.nml` file is modified by the `workshop_setup.csh` script.

6.41.2 Quick Start

To become familiar with the model, try this quick experiment.

1. compile everything in the `model/forced_lorenz_96/work` directory.

```
cd $DARTROOT/models/forced_lorenz_96/work
./quickbuild.csh
```

2. make sure the `input.nml` looks like the following (there is a lot that has been left out for clarity, these are the settings of interest for this example):

```
&perfect_model_obs_nml
   start_from_restart    = .true.,
   output_restart        = .true.,
   async                 = 0,
   restart_in_file_name  = "perfect_ics",
   obs_seq_in_file_name  = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   ...
/

&filter_nml
   async                    = 0,
   ens_size                 = 80,
   start_from_restart       = .true.,
   output_restart           = .true.,
   obs_sequence_in_name     = "obs_seq.out",
   obs_sequence_out_name    = "obs_seq.final",
   restart_in_file_name     = "filter_ics",
   restart_out_file_name    = "filter_restart",
   num_output_state_members = 80,
   num_output_obs_members   = 80,
   ...
/

&model_nml
   num_state_vars    = 40,
   forcing           = 8.00,
   delta_t           = 0.05,
   time_step_days    = 0,
   time_step_seconds = 3600,
```

(continues on next page)

(continued from previous page)

```
   reset_forcing      = .false.,
   random_forcing_amplitude = 0.10
/
```

3. Run `perfect_model_obs` to generate `true_state.nc` and `obs_seq.out`. The default `obs_seq.in` will cause the model to advance for 1000 time steps.

```
./perfect_model_obs
```

4. If you have *ncview*, explore the `true_state.nc`. Notice that the State Variable indices from 1-40 are the dynamical part of the model and 41-80 are the Forcing variables.

```
ncview true_state.nc
```

5. Run `filter` to generate `preassim.nc`, `analysis.nc` and `obs_seq.final`.

```
./filter
```

6. Launch Matlab and run `plot_ens_time_series`.

```
>> plot_ens_time_series
Input name of prior or posterior diagnostics file for preassim.nc:
preassim.nc
OPTIONAL: if you have the true state and want it superimposed,
provide the name of the input file. If not, enter a dummy filename.
Input name of True State file for true_state.nc:
true_state.nc
Using state state variable IDs 1 13 27
If these are OK, ;
If not, please enter array of state variable ID's
To choose from entire state enter A 25 50 75 (between 1 and 80)
To choose traditional model state enter S 1 23 40 (between 1 and 40)
To choose forcing estimates enter F 2 12 22 (between 1 and 40)
(no intervening syntax required)
A 20 30 40 60 70 80
```

Indices 20, 30, and 40 will be from the dynamical part of the lorenz_96 attractor, indices 60, 70, and 80 will be the corresponding Forcing values. Here are some images for just indices 20 and 60. Click on each image for a high-res version.

Repeat the experiment with *reset_forcing* = *.true.* when creating the true state and *reset_forcing* = *.false.* when assimilating. What happens?

6.41.3 Namelist

The model also implements the variant of Smith (2001), which can be invoked by setting `local_y = .true.` in the `&model_nml` namelist in the `input.nml` file.

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   num_state_vars    = 40,
   forcing           = 8.00,
```

(continues on next page)

(continued from previous page)

```
   delta_t            = 0.05,
   time_step_days     = 0,
   time_step_seconds = 3600,
   reset_forcing      = .false.,
   random_forcing_amplitude = 0.10
/
```

Description of each namelist entry

Item	Type	Description
num_state_vars	integer	Number of variables in model.
forcing	real(r8)	Forcing, F, for model.
delta_t	real(r8)	Non-dimensional timestep.
time_step_days	real(r8)	Base model time step maps to this much real time.
time_step_seconds	real(r8)	Base model time step maps to this.
reset_forcing	logical	If true, all forcing values are held fixed at the value specified for the forcing namelist.
random_forcing_amplitude	real(r8)	Standard deviation of the gaussian noise with zero mean that is added to each forcing value's time step.

6.41.4 References

6.42 MITgcm_ocean

6.42.1 Overview

The MIT ocean GCM version 'checkpoint59a' is the foundation of this directory. It was modified by Ibrahim Hoteit of Scripps for his use, and so it differs from the original distribution.

Since the model is highly parallelized, it can be compiled with a target number of processors in mind. From DART's perspective, the most logical strategy is to run `filter` or `perfect_model_obs` with **async=4**: advance the model in parallel ... one ensemble member after another. In this mode, the same set of processors are used for the data assimilation. The performance of the parallel assimilation algorithm has been tested up through 64 processors, and should scale well beyond that - but it remains to be quantified. The scaling for the ocean model is unknown to me, but Ibrahim routinely runs with many more than 64 processors.

As for all DART experiments, the overall design for an experiment is this: the DART program `filter` will read the initial conditions file, the observation sequence file, and the DART namelist to decide whether or not to advance the ocean model. All of the control of the execution of the ocean model is done by DART directly. If the model needs to be advanced, `filter` makes a call to the shell to execute the script `advance_model.csh`. `advance_model.csh` is ENTIRELY responsible for getting all the input files, data files, namelists, etc. into a temporary directory, running the model, and copying the results back to the parent directory (which we call CENTRALDIR). The whole process hinges on setting the ocean model namelist values such that it is doing a cold start for every model advance.

Observations

The observations for the ocean model were the first observations of oceanic quantities, so there is an `observations/forward_operators/obs_def_MITgcm_ocean_mod.f90` file containing the novel observation definitions like *salinity, sea surface height, current components ….* In keeping with the DART philosophy, there is a concept of inheritance between platform-specific observations like *DRIFTER_U_CURRENT_COMPONENT* and the general *U_CURRENT_COMPONENT*. Using the specific types when possible will allow flexibility specifying what kinds of observations to assimilate. *PROGRAM create_ocean_obs* is the program to create a DART observation sequence from a very particular ASCII file.

Converting between DART and the model

There are a set of support programs:

PROGRAM trans_pv_sv	converts the ocean model snapshot files into a DART-compatible format
PROGRAM trans_sv_pv	converts the DART output into snapshot files to be used as ocean model input datasets (specified in `data&PARM05`); creates a new `data` namelist file (`data.DART`) containing the correct `&PARM03; startTime,endTime` values to advance the ocean model the expected amount; and creates a new `data.cal` namelist file (`data.cal.DART`) containing the calendar information.
PROGRAM create_ocean_obs	create observation sequence files

The data assimilation period is controlled in the `input.nml&model_nml` namelist. In combination with the ocean model dynamics timestep `data&PARM03:deltaTClock` this determines the amount of time the model will advance for each assimilation cycle.

Generating the initial ensemble

The MITgcm_ocean model cannot (as of Oct 2008) take one single model state and generate its own ensemble (typically done with pert_model_state). This means I don't really know how to perform a 'perfect model' experiment until I find a way to correctly perturb a single state to create an ensemble.

The ensemble has to come from 'somewhere else'. I ran the model forward (outside the DART framework) for 14 days and output snapshot files ever 12 hours. One state vector can be generated from a set of snapshot files using `trans_pv_sv`. I called this my 'initial ensemble' - it's better than nothing, but it is ENTIRELY unknown if this creates an intial ensemble with sufficient spread. Just for comparison, the initial ensemble for the atmospheric models is derived from 'climatological' values. If they need an 80-member ensemble for July 14, 2008; they use the July 1 estimates of the atmosphere from 1900 to 1979. By the time they assimilate (every 6 hours) for several days, things are on-track.

There is a `shell_scripts/MakeInitialEnsemble.csh` script that was intended to automate this process - with modest success. It does illustrate the steps needed to convert each snapshot file to a DART initial conditions file and then run the restart_file_utility to overwrite the timestep in the header of the initial conditions file. After you have created all the initial conditions files, you can simply 'cat' them all together. Even if the script doesn't work *out-of-the-box*, it should be readable enough to be some help.

Fortran direct-access big-endian data files

The MITgcm_ocean model uses Fortran direct-access big-endian data files. It is up to you to determine the proper compiler flags to compile DART such that DART can read and write these files. Every compiler/architecture is different, but we have put notes in each `mkmf.template` if we know how to achieve this.

Controlling the model advances

The assimilation period is specified by two namelist parameters in the `input.nml&model_nml` namelist: `assimilation_period_days` and `assimilation_period_seconds`. Normally, all observations within (+/-) HALF of the total assimilation period are used in the assimilation.

The time of the initial conditions is specified by two namelist parameters in the `input.nml&model_nml` namelist: `init_time_days` and `init_time_seconds`; depending on the settings of these parameters, the times may or may not come directly from the DART initial conditions files.

The ocean model **MUST always** start from the input datasets defined in the `data&PARM05` namelist. Apparently, this requires `data&PARM03:startTime` to be **0.0**. One of the DART support routines (*PROGRAM trans_sv_pv*) converts the DART state vector to the files used in `data&PARM05` and creates new `data.cal&CAL_NML` and `data&PARM03` namelists with values appropriate to advance the model to the desired time.

The ocean model then advances till `data&PARM03:endTime` and writes out snapshot files. *PROGRAM trans_pv_sv* converts the snapshot files to a DART-compatible file which is ingested by `filter`. `filter` also reads the observation sequence file to determine which observations are within the assimilation window, assimilates them, and writes out a set of restart files, one for each ensemble member. `filter` then waits for each instance of the ocean model (one instance for each ensemble member) to advance to `data&PARM03:endTime`. The whole process repeats until 1) there are no more observations to assimilate (i.e. the observation sequence file is exhausted) or 2) the time specified by `input.nml&filter_nml:last_obs_days,last_obs_seconds` has been reached.

Getting started

I always like running something akin to a 'perfect model' experiment to start. Since I have not come up with a good way to perturb a single model state to generate an ensemble, here's the next best thing. Please keep in mind that the details for running each program are covered in their own documentation.

1. create a set of initial conditions for DART as described in Generating the intial ensemble and keep a copy of the 'middle' snapshot - then use it as the initial condition for `perfect_model_obs`.
2. create a TINY set of 'perfect' observations in the normal fashion: *program create_obs_sequence* and then *program create_fixed_network_seq* to create an empty observation sequence file (usually called `obs_seq.in`)
3. modify `data`, `data.cal`, and `input.nml` to control the experiment and populate the observation sequence file by running *program perfect_model_obs*

4. Now use the full ensemble of initial conditions from Step 1 and run *PROGRAM filter*

A perfectly sensible approach to get to know the system would be to try to

1. assimilate data for the first assimilation period and stop. Do not advance the model at all. The filter namelist can control all of this and you do not need to have a working `advance_model.csh` script, or even a working ocean model (as long as you have input data files).
2. advance the model first and then assimilate data for the first assimilation period and stop.
3. advance, assimilate and advance again. This tests the whole DART facility.

Exploring the output

Is pretty much like any other model. The netCDF files have the model prognostic variables before and after the assimilation. There are Matlab® scripts for perusing the netCDF files in the `DART/matlab` directory. There are Matlab® scripts for exploring the performance of the assimilation in observation-space (after running *PROGRAM obs_diag (for observations that use the threed_sphere location module)* to explore the `obs_seq.final` file) - use the scripts starting with `'plot_'`, e.g. `DART/diagnostics/matlab/plot_*.m`. As always, there are some model-specific item you should know about in `DART/models/MITgcm_ocean/matlab`, and `DART/models/MITgcm_ocean/shell_scripts`.

6.42.2 Other modules used

```
types_mod
time_manager_mod
threed_sphere/location_mod
utilities_mod
obs_kind_mod
mpi_utilities_mod
random_seq_mod
```

6.42.3 Public interfaces

Only a select number of interfaces used are discussed here.

use location_mod, only :	location_type
	get_location
	set_location

The ocean model namelists `data`, and `data.cal` *MUST* be present. These namelists are needed to reconstruct the valid time of the snapshot files created by the ocean model. Be aware that as DART advances the model, the `data` namelist gets modified to reflect the current time of the model output.

Required Interface Routines

use model_mod, only :

get_model_size

adv_1step

get_state_meta_data

model_interpolate

get_model_time_step

static_init_model

end_model

init_time

init_conditions

nc_write_model_atts

nc_write_model_vars

pert_model_state

get_close_maxdist_init

get_close_obs_init

get_close_obs

ens_mean_for_model

Unique Interface Routines

use model_mod, only :

MIT_meta_type

read_meta

write_meta

prog_var_to_vector

vector_to_prog_var

read_snapshot

write_snapshot

get_gridsize

snapshot_files_to_sv

sv_to_snapshot_files

timestep_to_DARTtime

DARTtime_to_MITtime

DARTtime_to_timestepindex

write_data_namelistfile

Ocean model namelist interfaces `&PARM03`, `&PARM04`, and `&PARM04` are read from file `data`. Ocean model namelist interface `&CAL_NML`, is read from file `data.cal`.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

model_size = get_model_size( )

```
integer :: get_model_size
```

Returns the length of the model state vector. Required.

`model_size`	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

`adv_1step` is not used for the MITgcm_ocean model. Advancing the model is done through the `advance_model` script. This is a NULL_INTERFACE, provided only for compatibility with the DART requirements.

`x`	State vector of length model_size.
`time`	Specifies time of the initial model state.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,             intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,   intent(out) ::  var_type
```

`get_state_meta_data` returns metadata about a given element of the DART representation of the model state vector. Since the DART model state vector is a 1D array and the native model grid is multidimensional, `get_state_meta_data` returns information about the native model state vector representation. Things like the `location`, or the type of the variable (for instance: salinity, temperature, u current component, . . .). The integer values used to indicate different variable types in `var_type` are themselves defined as public interfaces to model_mod if required.

`index_in`	Index of state vector element about which information is requested.
`location`	Returns the 3D location of the indexed state variable. The `location_ type` comes from `DART/location/threed_sphere/location_mod.f90`. Note that the lat/lon are specified in degrees by the user but are converted to radians internally.
var_type	Returns the type of the indexed state variable as an optional argument. The type is one of the list of supported observation types, found in the block of code starting `! Integer definitions for DART TYPES` in `DART/assimilation_code/modules/observations/obs_kind_mod.f90`

The list of supported variables in `DART/assimilation_code/modules/observations/obs_kind_mod.f90` is created by `preprocess` using the entries in `input.nml`[`&preprocess_nml, &obs_kind_nml`], `DEFAULT_obs_kin_mod.F90` and `obs_def_MITgcm_ocean_mod.f90`.

call model_interpolate(x, location, itype, obs_val, istatus)

```
real(r8), dimension(:), intent(in)  :: x
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: itype
real(r8),               intent(out) :: obs_val
integer,                intent(out) :: istatus
```

Given a model state, `model_interpolate` returns the value of the desired observation type (which could be a state variable) that would be observed at the desired location. The interpolation method is either completely specified by the model, or uses some standard 2D or 3D scalar interpolation routines. Put another way, `model_interpolate` will apply the forward operator **H** to the model state to create an observation at the desired location.

If the interpolation is valid, `istatus = 0`. In the case where the observation operator is not defined at the given location (e.g. the observation is below the lowest model level, above the top level, or 'dry'), interp_val is returned as 0.0 and istatus = 1.

`x`	A model state vector.
`location`	Location to which to interpolate.
`itype`	Not used.
`obs_val`	The interpolated value from the model.
`istatus`	Integer flag indicating the success of the interpolation. success == 0, failure == anything else

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

`get_model_time_step` returns the forecast length to be used as the "model base time step" in the filter. This is the minimum amount of time the model can be advanced by `filter`. *This is also the assimilation window*. All observations within (+/-) one half of the forecast length are used for the assimilation. In the `MITgcm_ocean` case, this is set from the namelist values for `input.nml&model_nml:assimilation_period_days`, `assimilation_period_seconds`, after ensuring the forecast length is a multiple of the ocean model dynamical timestep declared by `data&PARM03:deltaTClock`.

`var`	Smallest time step of model.

Please read the note concerning Controlling the model advances

call static_init_model()

`static_init_model` is called for runtime initialization of the model. The namelists are read to determine runtime configuration of the model, the calendar information, the grid coordinates, etc. There are no input arguments

and no return values. The routine sets module-local private attributes that can then be queried by the public interface routines.
The namelists (all mandatory) are:
`input.nml&model_mod_nml`,
`data.cal&CAL_NML`,
`data&PARM03`,
`data&PARM04`, and
`data&PARM05`.

call end_model()

`end_model` is used to clean up storage for the model, etc. when the model is no longer needed. There are no arguments and no return values. This is required by DART but nothing needs to be done for the MITgcm_ocean model.

call init_time(time)

```
type(time_type), intent(out) :: time
```

`init_time` returns the time at which the model will start if no input initial conditions are to be used. This is frequently used to spin-up models from rest, but is not meaningfully supported for the MITgcm_ocean model. The only time this routine would get called is if the `input.nml&perfect_model_obs_nml:start_from_restart` is .false., which is not supported in the MITgcm_ocean model.

`time`	the starting time for the model if no initial conditions are to be supplied. As of Oct 2008, this is hardwired to 0.0

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

`init_conditions` returns default initial conditions for model; generally used for spinning up initial model states. For the MITgcm_ocean model it is just a stub because the initial state is always provided by the input files.

`x`	Model state vector. [default is 0.0 for every element of the state vector]

ierr = nc_write_model_atts(ncFileID)

```
integer               :: nc_write_model_atts
integer, intent(in)  :: ncFileID
```

nc_write_model_atts writes model-specific attributes to an opened netCDF file: In the MITgcm_ocean case, this includes information like the coordinate variables (the grid arrays: XG, XC, YG, YC, ZG, ZC, . . .), information from some of the namelists, and either the 1D state vector or the prognostic variables (S,T,U,V,Eta). All the required information (except for the netCDF file identifier) is obtained from the scope of the model_mod module.

ncFileID	Integer file descriptor to previously-opened netCDF file.
ierr	Returns a 0 for successful completion.

nc_write_model_atts is responsible for the model-specific attributes in the following DART-output netCDF files: true_state.nc, preassim.nc, and analysis.nc.

ierr = nc_write_model_vars(ncFileID, statevec, copyindex, timeindex)

```
integer                                :: nc_write_model_vars
integer,                   intent(in)  :: ncFileID
real(r8), dimension(:),  intent(in)  :: statevec
integer,                   intent(in)  :: copyindex
integer,                   intent(in)  :: timeindex
```

nc_write_model_vars writes a copy of the state variables to a NetCDF file. Multiple copies of the state for a given time are supported, allowing, for instance, a single file to include multiple ensemble estimates of the state. Whether the state vector is parsed into prognostic variables (S,T,U,V,Eta) or simply written as a 1D array is controlled by input.nml&model_mod_nml:output_state_vector. If output_state_vector = .true. the state vector is written as a 1D array (the simplest case, but hard to explore with the diagnostics). If output_state_vector = .false. the state vector is parsed into prognostic variables before being written.

ncFileID	file descriptor to previously-opened netCDF file.
statevec	A model state vector.
copyindex	Integer index of copy to be written.
timeindex	The timestep counter for the given state.
ierr	Returns 0 for normal completion.

call pert_model_state(state, pert_state, interf_provided)

```
real(r8), dimension(:),  intent(in)   :: state
real(r8), dimension(:),  intent(out) :: pert_state
logical,                   intent(out) :: interf_provided
```

Given a model state, pert_model_state produces a perturbed model state. This is used to generate ensemble initial conditions perturbed around some control trajectory state when one is preparing to spin-up ensembles. Since the DART state vector for the MITgcm_ocean model contains both ‘wet’ and ‘dry’ cells, (the ‘dry’ cells having a value of a perfect 0.0 - not my choice) it is imperative to provide an interface to perturb **just** the wet cells (`interf_provided == .true.`).

At present (Oct 2008) the magnitude of the perturbation is wholly determined by `input.nml&model_mod_nml:model_perturbation_amplitude` and **utterly, completely fails**. The resulting model states cause a fatal error when being read in by the ocean model - something like

```
*** ERROR *** S/R INI_THETA: theta = 0 identically.
If this is intentional you will need to edit ini_theta.F to avoid this safety check
```

A more robust perturbation mechanism is needed (see, for example this routine in the CAM model_mod.f90). Until then, you can avoid using this routine by using your own ensemble of initial conditions. This is determined by setting `input.nml&filter_nml:start_from_restart = .false.` See also Generating the initial ensemble at the start of this document.

`state`	State vector to be perturbed.
`pert_state`	The perturbed state vector.
`interf_provided`	Because of the 'wet/dry' issue discussed above, this is always `.true.`, indicating a model-specific perturbation is available.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
real(r8),             intent(in)    :: maxdist
```

Pass-through to the 3-D sphere locations module. See get_close_maxdist_init() for the documentation of this subroutine.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(num)
```

Pass-through to the 3-D sphere locations module. See get_close_obs_init() for the documentation of this subroutine.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, num_close, close_ind [, dist])

```
type(get_close_type), intent(in)  :: gc
type(location_type),  intent(in)  :: base_obs_loc
integer,              intent(in)  :: base_obs_kind
type(location_type),  intent(in)  :: obs(:)
```

(continues on next page)

(continued from previous page)

```
integer,              intent(in)  :: obs_kind(:)
integer,              intent(out) :: num_close
integer,              intent(out) :: close_ind(:)
real(r8), optional,   intent(out) :: dist(:)
```

Pass-through to the 3-D sphere locations module. See get_close_obs() for the documentation of this subroutine.

call ens_mean_for_model(ens_mean)

```
real(r8), dimension(:), intent(in) :: ens_mean
```

`ens_mean_for_model` saves a copy of the ensemble mean to module-local storage. Sometimes the ensemble mean is needed rather than individual copy estimates. This is a NULL_INTERFACE for the MITgcm_ocean model. At present there is no application which requires module-local storage of the ensemble mean. No storage is allocated.

`ens_mean`	Ensemble mean state vector

6.42.4 Unique interface routines

```
type MIT_meta_type
   private
   integer           :: nDims
   integer           :: dimList(3)
   character(len=32) :: dataprec
   integer           :: reclen
   integer           :: nrecords
   integer           :: timeStepNumber
end type MIT_meta_type
```

`MIT_meta_type` is a derived type used to codify the metadata associated with a snapshot file.

Component	Description
nDims	the number of dimensions for the associated object. S,T,U,V all have nDims==3, Eta has nDims==2
dimList	the extent of each of the dimensions
dataprec	a character string depicting the precision of the data storage. Commonly 'float32'
reclen	the record length needed to correctly read using Fortran direct-access. This is tricky business. Each vendor has their own units for record length. Sometimes it is bytes, sometimes words, sometimes ???. See comments in code for `item_size_direct_access`
nrecords	the number of records (either 2D or 3D hyperslabs) in the snapshot file
timeStepNumber	the timestep number … the snapshot filenames are constructed using the timestepcount as the unique part of the filename. To determine the valid time of the snapshot, you must multiply the timeStepNumber by the amount of time in each timestep and add the start time.

metadata = read_meta(fbase [, vartype])

```
character(len=*),           intent(in)  :: fbase
character(len=*), OPTIONAL, intent(in)  :: vartype
type(MIT_meta_type),        intent(out) :: metadata
```

`read_meta` reads the metadata file for a particular snapshot file. This routine is primarily bulletproofing, since the snapshot files tend to move around a lot. I don't want to use a snapshot file from a 70-level case in a 40-level experiment; and without checking the metadata, you'd never know. The metadata for the file originally comes from the namelist values specifying the grid resolution, etc. If the metadata file exists, the metadata in the file is compared to the original specifications. If the metadata file does not exist, no comparison is done.

The filename is fundamentally comprised of three parts. Take 'U.0000000024.meta' for example. The first part of the name is the variable, the second part of the name is the timestepnumber, the last part is the file extension. For various reasons, sometimes it is convenient to call this function without the building the entire filename outside the function and then passing it in as an argument. Since the '.meta' extension seems to be fixed, we will only concern ourselves with building the 'base' part of the filename, i.e., the first two parts.

`fbase`	If *vartype* is supplied, this is simply the timestepnumber converted to a character string of length 10. For example, '0000000024'. If *vartype* is **not** supplied, it is the entire filename without the extension; 'U.0000000024', for example.
vartype	is an optional argument specifying the first part of the snapshot filename. Generally, 'S','T','U','V', or 'Eta'.
`metadata`	The return value of the function is the metadata for the file, packed into a user-derived variable type specifically designed for the purpose.

Metadata example

```
metadata = read_meta('U.0000000024')
 ... or ...
metadata = read_meta('0000000024','U')
```

call write_meta(metadata, filebase)

```
type(MIT_meta_type),         intent(in) ::  metadata
character(len=*),            intent(in) ::  filebase
```

`write_meta` writes a metadata file. This routine is called by routines `write_2d_snapshot`, and `write_3d_snapshot` to support converting the DART state vector to something the ocean model can ingest.

`metadata`	The user-derived varible, filled with the metadata for the file.
`filebase`	the filename without the extension; ‘U.0000000024’, for example. (see the Description in `read_meta`)

call prog_var_to_vector(s,t,u,v,eta,x)

```
real(r4), dimension(:,:,:), intent(in)  :: s,t,u,v
real(r4), dimension(:,:),   intent(in)  :: eta
real(r8), dimension(:),     intent(out) :: x
```

`prog_var_to_vector` packs the prognostic variables [S,T,U,V,Eta] read from the snapshot files into a DART vector. The DART vector is simply a 1D vector that includes all the ‘dry’ cells as well as the ‘wet’ ones. This routine is not presently used (since we never have [S,T,U,V,Eta] as such in memory). See snapshot_files_to_sv.

`s,`
`t,`
`u,v`
	The 3D arrays read from the individual snapshot files.

`eta`	The 2D array read from its snapshot file.
`x`	the 1D array containing the concatenated s,t,u,v,eta variables. To save storage, it is possible to modify the definition of `r8` in `DART/common/types_mod.f90` to be the same as that of `r4`.

call vector_to_prog_var(x,varindex,hyperslab)

```
real(r8), dimension(:),     intent(in)  :: x
integer,                    intent(in)  :: varindex
real(r4), dimension(:,:,:), intent(out) :: hyperslab -or-
real(r4), dimension(:,:),   intent(out) :: hyperslab
```

`vector_to_prog_var` unpacks a prognostic variable [S,T,U,V,Eta] from the DART vector `x`.

`x`	the 1D array containing the 1D DART state vector.

`varindex`	an integer code specifying which variable to unpack. The following parameters are in module storage:
	`integer, parameter :: S_index   = 1`
	`integer, parameter :: T_index   = 2`
	`integer, parameter :: U_index   = 3`
	`integer, parameter :: V_index   = 4`
	`integer, parameter :: Eta_index = 5`

`hyperslab`	The N-D array containing the prognostic variable. The function is overloaded to be able to return both 2D and 3D arrays.

Vector_to_prog_var

```
call vector_to_prog_var(statevec,V_index,data_3d)
 - or -
call vector_to_prog_var(statevec,Eta_index,data_2d)
```

call read_snapshot(fbase, x, timestep, vartype)

```
character(len=*),            intent(in)   :: fbase
real(r4), dimension(:,:,:), intent(out)  :: x - or -
real(r4), dimension(:,:),   intent(out)  :: x
integer,                     intent(out)  :: timestep
character(len=*), optional, intent(in)   :: vartype
```

`read_snapshot` reads a snapshot file and returns a hyperslab that includes all the 'dry' cells as well as the 'wet' ones. By design, the MITgcm_ocean model writes out Fortran direct-access big-endian binary files, independent of the platform. Since it is not guaranteed that the binary file we need to read is on the same architecture that created the file, getting the compiler settings in `mkmf.template` correct to read Fortran direct-access big-endian binary files is **imperative** to the process. Since each compiler issues its own error, there's no good way to even summarize the error messages you are likely to encounter by improperly reading the binary files. Read each template file for hints about the proper settings. See also the section Fortran direct-access big-endian datafiles in the "Discussion" of this document.

`fbase`	The 'base' portion of the filename, i.e., without the [.meta, .data] extension. If *vartype* is supplied, *vartype* is prepended to `fbase` to create the 'base' portion of the filename.
`x`	The hyperslab containing what is read. The function is overloaded to be able to return a 2D or 3D array. `x` must be allocated before the call to `read_snapshot`.
`timestep`	the timestepcount in the `'fbase'`.meta file, if the .meta file exists. Provided for bulletproofing.
vartype	The character string representing the 'prognostic variable' portion of the snapshot filename. Commonly 'S','T','U','V', or 'Eta'. If supplied, this is prepended to `fbase` to create the 'base' portion of the filename.

Code snippet

```
real(r4), allocatable :: data_2d_array(:,:), data_3d_array(:,:,:)
...
allocate(data_2d_array(Nx,Ny), data_3d_array(Nx,Ny,Nz))
...
call read_snapshot('S.0000000024', data_3d_array, timestepcount_out)
call read_snapshot(  '0000000024', data_2d_array, timestepcount_out, 'Eta')
call read_snapshot(  '0000000024', data_3d_array, timestepcount_out, 'T')
...
```

call write_snapshot(x, fbase, timestepcount)

```
real(r4), dimension(:,:),   intent(in) :: x - or -
real(r4), dimension(:,:,:), intent(in) :: x
character(len=*),           intent(in) :: fbase
integer, optional,          intent(in) :: timestepcount
```

`write_snapshot` writes a hyperslab of data to a snapshot file and corresponding metadata file. This routine is an integral part of sv_to_snapshot_files, the routine that is responsible for unpacking the DART state vector and writing out a set of snapshot files used as input to the ocean model.

`x`	The hyperslab containing the prognostic variable data to be written. The function is overloaded to be able to ingest a 2D or 3D array.
`fbase`	The 'base' portion of the filename, i.e., without the [.meta, .data] extension.
`timestepcount`	the timestepcount to be written into the `'fbase'`.meta file. If none is supplied, `timestepcount` is 0. I'm not sure this is ever used, since the timestepcount can be gotten from `fbase`.

call get_gridsize(num_x, num_y, num_z)

```
integer, intent(out) :: num_x, num_y, num_z
```

`get_gridsize` returns the dimensions of the compute domain. The gridsize is determined from `data&PARM04:delY,delX`, and `delZ` when the namelist is read by `static_init_model`. The MITgcm_ocean model is interesting in that it has a staggered grid but all grid variables are declared the same length.

`num_x`	The number of longitudinal gridpoints.
`num_y`	The number of latitudinal gridpoints.
`num_z`	The number of vertical gridpoints.

call snapshot_files_to_sv(timestepcount, state_vector)

```
integer,  intent(in)     :: timestepcount
real(r8), intent(inout) :: state_vector
```

`snapshot_files_to_sv` reads the snapshot files for a given timestepcount and concatenates them into a DART-compliant 1D array. All the snapshot filenames are constructed given the `timestepcount` - read the 'Description' section of read_meta, particularly the second paragraph.

`timestepcount`	The integer that corresponds to the middle portion of the snapshot filename.
`state_vector`	The 1D array of the DART state vector.

The files are read in this order [S,T,U,V,Eta] (almost alphabetical!) and the multidimensional arrays are unwrapped with the leftmost index being the fastest-varying. You shouldn't need to know this, but it is critical to the way `prog_var_to_vector` and `vector_to_prog_var` navigate the array.

```
do k = 1, Nz   ! depth
do j = 1, Ny   ! latitudes
do i = 1, Nx   ! longitudes
   state_vector(indx) = data_3d_array(i, j, k)
   indx = indx + 1
enddo
enddo
enddo
```

call sv_to_snapshot_files(state_vector, date1, date2)

```
real(r8), intent(in)     :: state_vector
integer,  intent(in)     :: date1, date2
```

`sv_to_snapshot_files` takes the DART state vector and creates a set of snapshot files. The filenames of these snapshot files is different than that of snapshot files created by the ocean model. See the 'Notes' section for an explanation.

`state_vector`	The DART 1D state vector.
`date1`	The year/month/day of the valid time for the state vector, in YYYYMMDD format - an 8-digit integer. This is the same format as `data.cal&CAL_NML:startDate_1`
`date2`	The hour/min/sec of the valid time for the state vector, in HHMMSS format. This is the same format as `data.cal&CAL_NML:startDate_2`

Since the snapshot files have the potential to move around a lot, I thought it best to have a more descriptive name than simply the snapshot number. DART creates snapshot files with names like `S.19960718.060000.data` to let you know it is a snapshot file for 06Z 18 July 1996. This is intended to make it easier to create initial conditions files and, should the assimilation fail, inform as to _when_ the assimilation failed. Since DART needs the ocean model to coldstart (`data&PARM02:startTime = 0.0`) for every model advance, every snapshot file has the same timestamp. The `advance_model.csh` script actually has to rename the DART-written snapshot files to that declared by the `data&PARM05` namelist, so the name is not really critical from that perspective. **However**, the components of the DART-derived snapshot files **are** used to create an appropriate `data.cal&CAL_NML` for each successive model advance.

mytime = timestep_to_DARTtime(TimeStepIndex)

```
integer,         intent(in)  :: TimeStepIndex
type(time_type), intent(out) :: mytime
```

timestep_to_DARTtime combines the TimeStepIndex with the time per timestep (from data&PARM03) and the start date supplied by data.cal&CAL_NML to form a Gregorian calendar date which is then converted to a DART time object. As of Oct 2008, this model_mod is forced to use the Gregorian calendar.

TimeStepIndex	an integer referring to the ocean model timestep ... the middle part of the ocean-model-flavor snapshot filename.
mytime	The DART representation of the time indicated by the TimeStepIndex

The time per timestep is something I don't understand that well. The data&PARM03 namelist has three variables: deltaTmom, deltaTtracer, and deltaTClock. Since I don't know which one is relavent, and every case I looked at had them set to be the same, I decided to require that they all be identical and then it wouldn't matter which one I used. The values are checked when the namelist is read.

```
! Time stepping parameters are in PARM03
call find_namelist_in_file("data", "PARM03", iunit)
read(iunit, nml = PARM03, iostat = io)
call check_namelist_read(iunit, io, "PARM03")

if ((deltaTmom   == deltaTtracer) .and. &
    (deltaTmom   == deltaTClock ) .and. &
    (deltaTClock == deltaTtracer)) then
   timestep       = deltaTmom                    ! need a time_type version
else
   write(msgstring,*)"namelist PARM03 has deltaTmom /= deltaTtracer /= deltaTClock"
   call error_handler(E_MSG,"static_init_model", msgstring, source, revision, revdate)
   write(msgstring,*)"values were ",deltaTmom, deltaTtracer, deltaTClock
   call error_handler(E_MSG,"static_init_model", msgstring, source, revision, revdate)
   write(msgstring,*)"At present, DART only supports equal values."
   call error_handler(E_ERR,"static_init_model", msgstring, source, revision, revdate)
endif
```

call DARTtime_to_MITtime(darttime, date1, date2)

```
type(time_type), intent(in)  :: darttime
integer,         intent(out) :: date1, date2
```

DARTtime_to_MITtime converts the DART time to a pair of integers that are compatible with the format used in data.cal&CAL_NML

darttime	The DART time to be converted.
date1	The year/month/day component of the time in YYYYMMDD format - an 8-digit integer. This is the same format as data.cal&CAL_NML:startDate_1
date2	The hour/min/sec component of the time in HHMMSS format. This is the same format as data.cal&CAL_NML:startDate_2

timeindex = DARTtime_to_timestepindex(darttime)

```
type(time_type), intent(in)  :: darttime
integer,         intent(out) :: timeindex
```

`DARTtime_to_timestepindex` converts the DART time to an integer representing the number of timesteps since the date in `data.cal&CAL_NML`, i.e., the start of the model run. The size of each timestep is determined as discussed in the timestep_to_DARTtime section.

`darttime`	The DART time to be converted.
`timeindex`	The number of timesteps corresponding to the DARTtime . . .

call write_data_namelistfile()

There are no input arguments to `write_data_namelistfile`. `write_data_namelistfile` reads the `data` namelist file and creates an almost-identical copy named `data.DART` that differs only in the namelist parameters that control the model advance.

(NOTE) `advance_model.csh` is designed to first run `trans_sv_pv` to create appropriate `data.DART` and `data.cal.DART` files. The script then renames them to that expected by the ocean model.

6.42.5 Namelists

We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input. Consider yourself forewarned that character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
namelist /model_nml/  assimilation_period_days, &
     assimilation_period_seconds, output_state_vector, model_perturbation_amplitude
```

This namelist is read in a file called `input.nml`. This namelist provides control over the assimilation period for the model. All observations within (+/-) half of the assimilation period are assimilated. The assimilation period is the minimum amount of time the model can be advanced, and checks are performed to ensure that the assimilation window is a multiple of the ocean model dynamical timestep indicated by `PARM03:deltaTClock`.

Contents	Type	Description
assimilation_period_days	integer *[default: 7]*	The number of days to advance the model for each assimilation.
assimilation_period_seconds	integer *[default: 0]*	In addition to `assimilation_period_days`, the number of seconds to advance the model for each assimilation.
output_state_vector	logical *[default: .true.]*	The switch to determine the form of the state vector in the output netcdf files. If `.true.` the state vector will be output exactly as DART uses it … one long array. If `.false.`, the state vector is parsed into prognostic variables and output that way – much easier to use with 'ncview', for example.
model_perturbation_amplitude	real(r8) *[default: 0.2]*	The amount of noise to add when trying to perturb a single state vector to create an ensemble. Only needed when `inpu t.nml&filter_nml:start_from_restart = .false.` See also Generating the initial ensemble at the start of this document. units: standard deviation of a gaussian distribution with the mean at the value of the state vector element.

Model namelist

```
&model_nml
   assimilation_period_days     = 1,
   assimilation_period_seconds  = 0,
   model_perturbation_amplitude = 0.2,
   output_state_vector          = .false.  /
```

```
namelist /CAL_NML/  TheCalendar, startDate_1, startDate_2, calendarDumps
```

This namelist is read in a file called `data.cal` This namelist is the same one that is used by the ocean model. The values **must** correspond to the date at the start of an experiment. This is more important for `create_ocean_obs`, `trans_pv_sv` than for `filter` and *PROGRAM trans_sv_pv* since `trans_sv_pv` takes the start time of the experiment from the DART initial conditions file and actually writes a new `data.cal.DART` and a new `data.DART` file. `advance_model.csh` renames `data.DART` and `data.cal.DART` to be used for the model advance.

Still, the files must exist before DART runs to avoid unnecessarily complex logic. If you are running the support programs in a standalone fashion (as you might if you are converting snapshot files into an intial ensemble), it is critical that the values in this namelist are correct to have accurate times in the headers of the restart files. You can always patch the times in the headers with `restart_file_utility`.

```
namelist /PARM03/  startTime, endTime, deltaTmom, &
                     deltaTtracer, deltaTClock, dumpFreq, taveFreq, ...
```

This namelist is read in a file called `data`. This namelist is the same one that is used by the ocean model. Only the variables listed here are used by the DART programs, there are more variables that are used only by the ocean model. There are two scenarios of interest for this namelist.

1. During an experiment, the `advance_model.csh` script is invoked by `filter` and the namelist is read by `trans_sv_pv` and REWRITTEN for use by the ocean model. Since this all happens in a local directory for the model advance, only a copy of the input `data` file is overwritten. The intent is that the `data` file is preserved 'perfectly' except for the values in `&PARM03` that pertain to controlling the model advance: `endTime`, `dumpFreq`, and `taveFreq`.
2. Outside the confines of `trans_sv_pv`, this namelist is always simply read and is unchanged.

Contents	Type	Description
startTime	real(r8)	This **must** be 0.0 to tell the ocean model to read from the input files named in `data&PARM05`.
endTime	real(r8)	The number of seconds for one model advance. (normally set by `trans_sv_pv`)
deltaTmom, deltaTtracer, deltaTClock	real(r8)	These are used when trying to interpret the timestepcount in the snapshot files. They must all be identical unless someone can tell me which one is used when the ocean model creates snapshot filenames.
dumpFreq, taveFreq	real(r8)	Set to the same value value as `endTime`. I have never run with different settings, my one concern would be how this affects a crappy piece of logic in `advance_model.csh` that requires there to be exactly ONE set of snapshot files - and that they correspond to the completed model advance.

This namelist is the same one that is used by the ocean model. Only some of the namelist variables are needed by DART; the rest are ignored by DART but could be needed by the ocean model. Here is a fragment for a daily assimilation timestep with the model dynamics having a much shorter timestep.

Parm03 namelist

```
&PARM03
   startTime    =     0.,
     endTime    = 86400.,
   deltaTmom    =   900.,
   deltaTtracer =   900.,
   deltaTClock  =   900.,
   dumpFreq     = 86400.,
   taveFreq     = 86400.,
     ...
```

This would result in snapshot files with names like `[S,T,U,V,Eta].0000000096.data` since 86400/900 = 96. These values remain fixed for the entire assimilation experiment, the only thing that changes from the ocean model's perspective is a new `data.cal` gets created for every new assimilation cycle. `filter` is responsible for starting and stopping the ocean model. The DART model state has a valid time associated with it, this information is used to create the new `data.cal`.

```
namelist /PARM04/  phiMin, thetaMin, delY, delX, delZ, ...
```

This namelist is read in a file called `data`. This namelist is the same one that is used by the ocean model. Only the variables listed here are used by the DART programs, there are more variables that are used only by the ocean model.

Contents	Type	Description
phiMin	real(r8)	The latitude of the southmost grid edge. In degrees.
thetaMin	real(r8)	The longitude of the leftmost grid edge. In degrees.
delY	real(r8), dimension(1024)	The latitudinal distance between grid cell edges. In degrees. The array has a default value of 0.0. The number of non-zero entries determines the number of latitudes. static_init_model() converts the namelist values to grid centroids and edges.
delX	real(r8), dimension(1024)	The longitudinal distance between grid cell edges. In degrees. The array has a default value of 0.0. The number of non-zero entries determines the number of longitudes. static_init_model() converts the namelist values to grid centroids and edges.
delZ	real(r8), dimension(512)	The vertical distance between grid cell edges i.e., the thickness of the layer. In meters. The array has a default value of 0.0. The number of non-zero entries determines the number of depths. static_init_model() converts the namelist values to grid centroids and edges.

This namelist is the same one that is used by the ocean model. Only some of the namelist variables are needed by DART; the rest are ignored by DART but could be needed by the ocean model. Here is a fragment for a (NY=225, NX=256, NZ=. . .) grid

Parm04 namelist

```
&PARM04
   phiMin   =     8.4,
   thetaMin =   262.0,
   delY     = 225*0.1,
   delX     = 256*0.1,
   delZ     =  5.0037,
               5.5860,
               6.2725,
               7.0817,
               8.0350,
               9.1575,
              10.4786,
              12.0322,
              13.8579,
              16.0012,
                ...
```

Note that the `225*0.1` construct exploits the Fortran repeat mechanism to achieve 225 evenly-spaced gridpoints without having to manually enter 225 identical values. No such construct exists for the unevenly-spaced vertical layer thicknesses, so each layer thickness is explicitly entered.

```
namelist /PARM05/  bathyFile, hydrogSaltFile, hydrogThetaFile, &
                  uVelInitFile, vVelInitFile, pSurfInitFile
```

This namelist is read in a file called `data`. The only DART component to use this namelist is the shell script responsible for advancing the model - `advance_model.csh`.

Contents	Type	Description
bathyFile	character(len=*)	The Fortran direct-access big-endian binary file containing the bathymetry.
hydrogSaltFile	character(len=*)	The Fortran direct-access big-endian binary (snapshot) file containing the salinity. `S.0000000096.data`, for example. Units: psu
hydrogThetaFile	character(len=*)	The Fortran direct-access big-endian binary (snapshot) file containing the temperatures. `T.0000000096.data`, for example. Units: degrees C
uVelInitFile	character(len=*)	The Fortran direct-access big-endian binary (snapshot) file containing the U current velocities. `U.0000000096.data`, for example. Units: m/s
vVelInitFile	character(len=*)	The Fortran direct-access big-endian binary (snapshot) file containing the V current velocities. `V.0000000096.data`, for example. Units: m/s
pSurfInitFile	character(len=*)	The Fortran direct-access big-endian binary (snapshot) file containing the sea surface heights. `Eta.0000000096.data`, for example. Units: m

This namelist specifies the input files to the ocean model. DART must create these input files. `advance_model.csh` has an ugly block of code that actually 'reads' this namelist and extracts the names of the input files expected by the ocean model. `advance_model.csh` then **renames** the snapshot files to be that expected by the ocean model. For this reason (and several others) a DART experiment occurrs in a separate directory we call CENTRALDIR, and each model advance happens in a run-time subdirectory. The data files copied to the run-time directory are deemed to be volatile, i.e., we can overwrite them and change them during the course of an experiment.

6.42.6 Files

- input namelist files: `data, data.cal, input.nml`
- input data file: `filter_ics, perfect_ics`
- output data files: `[S,T,U,V,Eta].YYYYMMDD.HHMMSS.[data,meta]`

Please note that there are **many** more files needed to advance the ocean model, none of which are discussed here.

6.42.7 References

- none

6.42.8 Private components

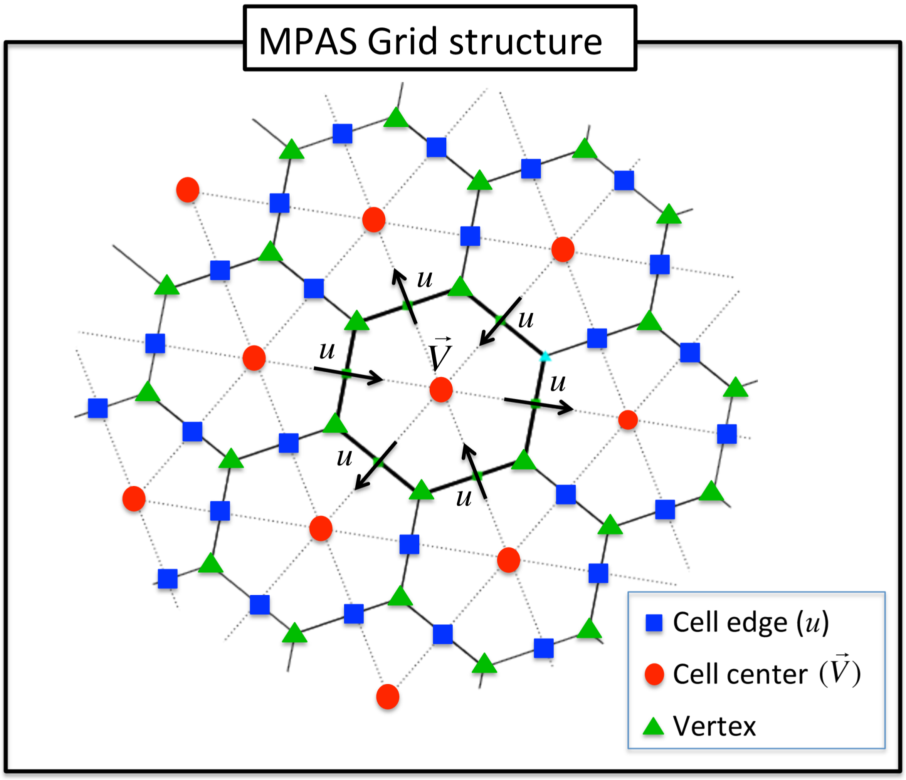
N/A

6.43 MPAS_ATM

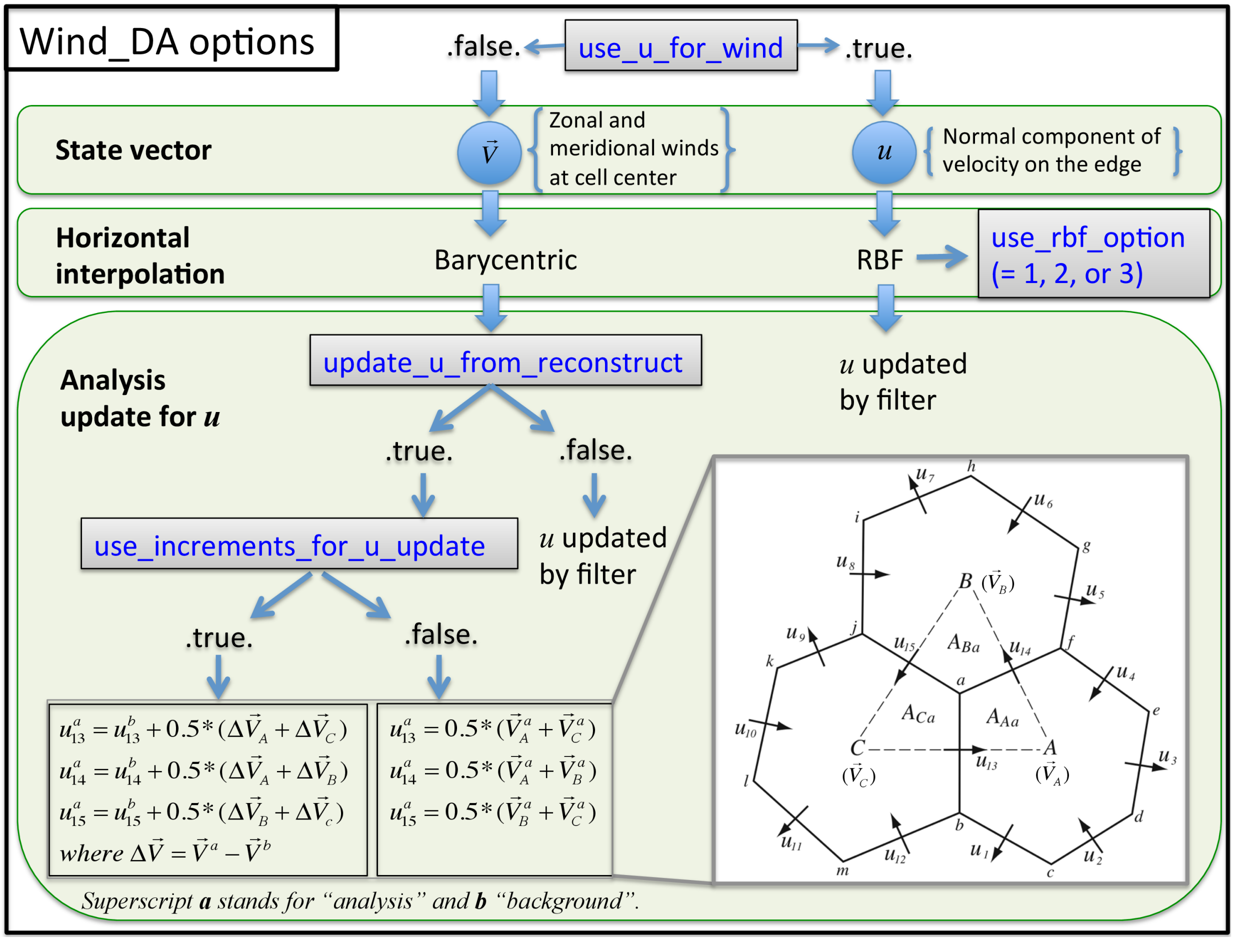
6.43.1 Overview

This document describes the DART interface module for the MPAS-Atmosphere (or briefly, MPAS-ATM) global model, which uses an unstructured Voronoi grid mesh, formally Spherical Centriodal Voronoi Tesselations (SCVTs). This allows for both quasi-uniform discretization of the sphere and local refinement. The MPAS/DART interface was built on the SCVT-dual mesh and does not regrid to regular lat/lon grids. In the C-grid discretization, the normal component of velocity on cell edges is prognosed; zonal and meridional wind components are diagnosed on the cell centers. We provide several options to choose from in the assimilation of wind observations as shown below.

The grid terminology used in MPAS is as shown in the figure below:



The wind options during a DART assimilation are controlled by combinations of 4 different namelist values. The values determine which fields the forward operator uses to compute expected observation values; how the horizontal interpolation is computed in that forward operator; and how the assimilation increments are applied to update the wind quantities in the state vector. Preliminary results based on real data assimilation experiments indicate that performance is better when the zonal and meridional winds are used as input to the forward operator that uses Barycentric interpolation, and when the prognostic u wind is updated by the incremental method described in the figure below. However there remain scientific questions about how best to handle the wind fields under different situations. Thus we have kept all implemented options available for use in experimental comparisons. See the figure below for a flow-chart representation of how the 4 namelist items interact:


Wind_DA options
.false.
use_u_for_wind
.true.
State vector
$\vec{V}$
Zonal and meridional winds at cell center
u
Normal component of velocity on the edge
Horizontal interpolation
Barycentric
RBF
use_rbf_option (= 1, 2, or 3)
Analysis update for u
update_u_from_reconstruct
u updated by filter
.true.
.false.
use_increments_for_u_update
u updated by filter
.true.
.false.
$u^a_{13} = u^b_{13} + 0.5*(\Delta\vec{V}_A + \Delta\vec{V}_C)$
$u^a_{14} = u^b_{14} + 0.5*(\Delta\vec{V}_A + \Delta\vec{V}_B)$
$u^a_{15} = u^b_{15} + 0.5*(\Delta\vec{V}_B + \Delta\vec{V}_C)$
where $\Delta\vec{V} = \vec{V}^a - \vec{V}^b$
$u^a_{13} = 0.5*(\vec{V}^a_A + \vec{V}^a_C)$
$u^a_{14} = 0.5*(\vec{V}^a_A + \vec{V}^a_B)$
$u^a_{15} = 0.5*(\vec{V}^a_B + \vec{V}^a_C)$
u_7 h u_6 i g u_8 B $(\vec{V}_B)$ u_5 u_9 j u_{15} A_{Ba} u_{14} f k a u_4 A_{Ca} A_{Aa} e u_{10} C A u_{13} u_3 $(\vec{V}_C)$ $(\vec{V}_A)$ l b d u_{11} u_{12} u_1 u_2 m c
Superscript $\boldsymbol{a}$ stands for "analysis" and $\boldsymbol{b}$ "background".

Cycling of MPAS/DART is run in a *restart* mode. As for all DART experiments, the overall design for an experiment is this: the DART program `filter` will read the initial condition file, the observation sequence file, and the DART namelist to decide whether or not to advance the MPAS-ATM model. All of the control of the execution of the MPAS model is done by DART directly. If the model needs to be advanced, `filter` makes a call to the shell to execute the script `advance_model.csh`, which is ENTIRELY responsible for getting all the input files, data files, namelists, etc. into a temporary directory, running the model, and copying the results back to the parent directory (which we call CENTRALDIR). The whole process hinges on setting the MPAS-ATM model namelist values such that it is doing a restart for every model advance. Unlike MPAS-ATM free forecast runs, the forecast step in MPAS/DART requires to set up one more namelist parameter called `config_do_DAcycling = .true.` in `&restart` section of `namelist.input` to recouple the state vectors (updated by filter) with the mass field for the restart mode. For more information, check the `advance_model.csh` script in `./shell_scripts/` directory.

Since DART is an ensemble algorithm, there are multiple analysis files for a single analysis time: one for each ensemble member. Because MPAS/DART is run in a restart mode, each member should keep its own MPAS restart file from the previous cycle (rather than having a single template file in CENTRALDIR). Creating the initial ensemble of states is an area of active research.

6.43.2 Namelist

This namelist is read from the file *input.nml*. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   init_template_filename        = 'mpas_init.nc',
   vert_localization_coord       = 3,
   assimilation_period_days      = 0,
   assimilation_period_seconds   = 21600,
   model_perturbation_amplitude = 0.0001,
   log_p_vert_interp             = .true.,
   calendar                      = 'Gregorian',
   use_u_for_wind                = .false.,
   use_rbf_option                = 2,
   update_u_from_reconstruct     = .true.,
   use_increments_for_u_update   = .true.,
   highest_obs_pressure_mb       = 100.0,
   sfc_elev_max_diff             = -1.0,
   outside_grid_level_tolerance = -1.0,
   extrapolate                   = .false.,
   debug                         = 0,
/
```

Item	Type	Description
init_template_filename	character(len=256) *[default: 'mpas_init.nc']*	The name of the MPAS analysis file to be read and/or written by the DART programs for the state data.
highest_obs_pressure_mb	real(r8) *[default: 100.0]*	Observations higher than this pressure are ignored. Set to -1.0 to ignore this test. For models with a prescribed top boundary layer, trying to assimilate very high observations results in problems because the model damps out any changes the assimilation tries to make. With adaptive algorithms this results in larger and larger coefficients as the assimilation tries to effect state vector change.
assimilation_period_days	integer *[default: 0]*	The number of days to advance the model for each assimilation. Even if the model is being advanced outside of the DART filter program, the assimilation period should be set correctly. Only observations with a time within +/- 1/2 this window size will be assimilated.
assimilation_period_seconds	integer *[default: 21600]*	In addition to `assimilation_period_days`, the number of seconds to advance the model for each assimilation.
vert_localization_coord	integer *[default: 3]*	Vertical coordinate for vertical localization. • 1 = model level • 2 = pressure (in pascals) • 3 = height (in meters) • 4 = scale height (unitless)
sfc_elev_max_diff	real(r8)*[default: -1.0]*	If > 0, the maximum difference, in meters, between an observation marked as a 'surface obs' as the vertical type (with the surface elevation, in meters, as the numerical vertical location), and the surface elevation as defined by the model. Observations further away from the surface than this threshold are rejected and not assimilated. If the value is negative, this test is skipped.
log_p_vert_interp	logical *[default: .true.]*	If `.true.`, vertical interpolation is done in log-pressure. Otherwise, linear.
use_u_for_wind	logical *[default: .false.]*	If `.false.`, zonal and meridional winds at cell centers are used for the wind observation operator [default]. In that case, triangular meshes are used for the barycentric (e.g., area-weighted) interpolation. If `.true.`, wind vectors at an arbitrary (e.g., observation) point are reconstructed from the normal component of velocity on cell edges *(u)* using radial

The &mpas_vars_nml namelist within input.nml contains the list of MPAS variables that make up the DART state vector. The order the items are specified controls the order of the data in the state vector, so it should not be changed without regenerating all DART initial condition or restart files. These variables are directly updated by the filter assimilation.

Any variables whose values cannot exceed a given minimum or maximum can be listed in mpas_state_bounds. When the data is written back into the MPAS NetCDF files values outside the allowed range will be detected and changed. Data inside the DART state vector and data written to the DART diagnostic files will not go through this test and values may exceed the allowed limits. Note that changing values at the edges of the distribution means it is no longer completely gaussian. In practice this technique has worked effectively, but if the assimilation is continually trying to move the values outside the permitted range the results may be of poor quality. Examine the diagnostics for these fields carefully when using bounds to restrict their values.

```
&mpas_vars_nml
   mpas_state_variables = 'theta',                 'QTY_POTENTIAL_TEMPERATURE',
                          'uReconstructZonal',     'QTY_U_WIND_COMPONENT',
                          'uReconstructMeridional','QTY_V_WIND_COMPONENT',
                          'qv',                    'QTY_VAPOR_MIXING_RATIO',
                          'qc',                    'QTY_CLOUDWATER_MIXING_RATIO',
                          'surface_pressure',      'QTY_SURFACE_PRESSURE'
   mpas_state_bounds    = 'qv','0.0','NULL','CLAMP',
                          'qc','0.0','NULL','CLAMP',
/
```

Item	Type	Description
mpas_vars_nml	character(len=NF90_MAX_NAME):: dimension(160)	The table that both specifies which MPAS-ATM variables will be placed in the state vector, and also relates those variables to the corresponding DART kinds. The first column in each pair must be the exact NetCDF name of a field in the MPAS file. The second column in each pair must be a KIND known to the DART system. See the obs_kind_mod.f90 file within assimilation_code/modules/observations/ for known names. This file is autogenerated when DART builds filter for a particular model, so run quickbuild.csh in the work directory first before examining this file. Use the generic kind list in the obs_kind_mod tables, not the specific type list.
mpas_state_bounds	character(len=NF90_MAX_NAME):: dimension(160)	List only MPAS-ATM variables that must restrict their values to remain between given lower and/or upper bounds. Columns are: NetCDF variable name, min value, max value, and action to take for out-of-range values. Either min or max can have the string 'NULL' to indicate no limiting will be done. If the action is 'CLAMP' out of range values will be changed to the corresponding bound and execution continues; 'FAIL' stops the executable if out of range values are detected.

6.43.3 Grid Information

As the forward operators use the unstructured grid meshes in MPAS-ATM, the DART/MPAS interface needs to read static variables related to the grid structure from the MPAS ATM 'history' file (specified in model_analysis_filename). These variables are used to find the closest cell to an observation point in the cartesian coordinate (to avoid the polar issues).

integer :: nCells	the number of cell centers
integer :: nEdges	the number of cell edges
integer :: nVertices	the number of cell vertices
integer :: nVertLevels	the number of vertical levels for mass fields
integer :: nVertLevelsP1	the number of vertical levels for vertical velocity
integer :: nSoilLevels	the number of soil levels
real(r8) :: latCell(:)	the latitudes of the cell centers [-90,90]
real(r8) :: lonCell(:)	the longitudes of the cell centers [0, 360]
real(r8) :: latEdge(:)	the latitudes of the edges [-90,90], if edge winds are used.
real(r8) :: lonEdge(:)	the longitudes of the edges [0, 360], if edge winds are used.
real(r8) :: xVertex(:)	The cartesian location in x-axis of the vertex
real(r8) :: yVertex(:)	The cartesian location in y-axis of the vertex
real(r8) :: zVertex(:)	The cartesian location in z-axis of the vertex
real(r8) :: xEdge(:)	The cartesian location in x-axis of the edge, if edge winds are used.
real(r8) :: yEdge(:)	The cartesian location in y-axis of the edge, if edge winds are used.
real(r8) :: zEdge(:)	The cartesian location in z-axis of the edge, if edge winds are used.
real(r8) :: zgrid(:,:)	geometric height at cell centers (nCells, nVertLevelsP1)
integer :: CellsOnVertex(:,:)	list of cell centers defining a triangle
integer :: edgesOnCell(:,:)	list of edges on each cell
integer :: verticesOnCell(:,:)	list of vertices on each cell
integer :: edgeNormalVectors(:,:)	unit direction vectors on the edges (only used if *use_u_for_wind* = .true.)

6.43.4 model_mod variable storage

The `&mpas_vars_nml` within `input.nml` defines the list of MPAS variables used to build the DART state vector. Combined with an MPAS analysis file, the information is used to determine the size of the DART state vector and derive the metadata. To keep track of what variables are contained in the DART state vector, an array of a user-defined type called “progvar” is available with the following components:

```
type progvartype
   private
   character(len=NF90_MAX_NAME) :: varname
   character(len=NF90_MAX_NAME) :: long_name
   character(len=NF90_MAX_NAME) :: units
   character(len=NF90_MAX_NAME), dimension(NF90_MAX_VAR_DIMS) :: dimname
   integer, dimension(NF90_MAX_VAR_DIMS) :: dimlens
   integer :: xtype         ! netCDF variable type (NF90_double, etc.)
   integer :: numdims       ! number of dimensions - excluding TIME
   integer :: numvertical   ! number of vertical levels in variable
   integer :: numcells      ! number of cell locations (typically cell centers)
   integer :: numedges      ! number of edge locations (edges for normal velocity)
   logical :: ZonHalf       ! vertical coordinate for mass fields (nVertLevels)
   integer :: varsize       ! variable size (dimlens(1:numdims))
   integer :: index1        ! location in dart state vector of first occurrence
   integer :: indexN        ! location in dart state vector of last  occurrence
   integer :: dart_kind
   character(len=paramname_length) :: kind_string
   logical  :: clamping     ! does variable need to be range-restricted before
   real(r8) :: range(2)     ! lower and upper bounds for the data range.
   logical  :: out_of_range_fail  ! is out of range fatal if range-checking?
end type progvartype

type(progvartype), dimension(max_state_variables) :: progvar
```

The variables are simply read from the MPAS analysis file and stored in the DART state vector such that all quantities for one variable are stored contiguously. Within each variable; they are stored vertically-contiguous for each horizontal location. From a storage standpoint, this would be equivalent to a Fortran variable dimensioned x(nVertical,nHorizontal,nVariables). The fastest-varying dimension is vertical, then horizontal, then variable … naturally, the DART state vector is 1D. Each variable is also stored this way in the MPAS analysis file.

6.43.5 Compilation

The DART interface for MPAS-ATM can be compiled with various fortran compilers such as (but not limited to) gfortran, pgf90, and intel. It has been tested on a Mac and NCAR IBM supercomputer (yellowstone).

Note: While MPAS requires the PIO (Parallel IO) and pNetCDF (Parallel NetCDF) libraries, DART uses only the plain NetCDF libraries. If an altered NetCDF library is required by the parallel versions, there may be incompatibilities between the run-time requirements of DART and MPAS. Static linking of one or the other executable, or swapping of modules between executions may be necessary.

6.43.6 Conversions

A Welcome Development

MPAS files no longer beed to be converted to DART formatted files, they can be read in directly from a input file list!

Analysis File NetCDF header

The header of an MPAS analysis file is presented below - simply for context. Keep in mind that **many** variables have been removed for clarity. Also keep in mind that the multi-dimensional arrays listed below have the dimensions reversed from the Fortran convention.

```
$ ncdump -h mpas_init.nc
netcdf mpas_analysis {
dimensions:
         StrLen = 64 ;
         Time = UNLIMITED ; // (1 currently)
         nCells = 10242 ;                                   available in DART
         nEdges = 30720 ;                                   available in DART
         maxEdges = 10 ;
         maxEdges2 = 20 ;
         nVertices = 20480 ;                                available in DART
         TWO = 2 ;
         THREE = 3 ;
         vertexDegree = 3 ;
         FIFTEEN = 15 ;
         TWENTYONE = 21 ;
         R3 = 3 ;
         nVertLevels = 41 ;                                 available in DART
         nVertLevelsP1 = 42 ;                               available in DART
         nMonths = 12 ;
         nVertLevelsP2 = 43 ;
         nSoilLevels = 4 ;                                  available in DART
variables:
         char xtime(Time, StrLen) ;                         available in DART
```

(continues on next page)

(continued from previous page)

```
         double latCell(nCells) ;                          available in DART
         double lonCell(nCells) ;                          available in DART
         double latEdge(nEdges) ;                          available in DART
         double lonEdge(nEdges) ;                          available in DART
         int indexToEdgeID(nEdges) ;
         double latVertex(nVertices) ;
         double lonVertex(nVertices) ;
      double xVertex(nVertices) ;                       available in DART
      double yVertex(nVertices) ;                       available in DART
      double zVertex(nVertices) ;                       available in DART
      double xEdge(nVertices) ;                         available in DART
      double yEdge(nVertices) ;                         available in DART
      double zEdge(nVertices) ;                         available in DART
         int indexToVertexID(nVertices) ;
         int cellsOnEdge(nEdges, TWO) ;
         int nEdgesOnCell(nCells) ;
         int nEdgesOnEdge(nEdges) ;
         int edgesOnCell(nCells, maxEdges) ;               available in DART
         int edgesOnEdge(nEdges, maxEdges2) ;
         double weightsOnEdge(nEdges, maxEdges2) ;
         double dvEdge(nEdges) ;
         double dcEdge(nEdges) ;
         double angleEdge(nEdges) ;
         double edgeNormalVectors(nEdges, R3) ;            available in DART
         double cellTangentPlane(nEdges, TWO, R3) ;
         int cellsOnCell(nCells, maxEdges) ;
         int verticesOnCell(nCells, maxEdges) ;            available in DART
         int verticesOnEdge(nEdges, TWO) ;
         int edgesOnVertex(nVertices, vertexDegree) ;
         int cellsOnVertex(nVertices, vertexDegree) ;      available in DART
         double kiteAreasOnVertex(nVertices, vertexDegree) ;
         double rainc(Time, nCells) ;
         double cuprec(Time, nCells) ;
         double cutop(Time, nCells) ;
         double cubot(Time, nCells) ;
         double relhum(Time, nCells, nVertLevels) ;
         double qsat(Time, nCells, nVertLevels) ;
         double graupelnc(Time, nCells) ;
         double snownc(Time, nCells) ;
         double rainnc(Time, nCells) ;
         double graupelncv(Time, nCells) ;
         double snowncv(Time, nCells) ;
         double rainncv(Time, nCells) ;
         double sr(Time, nCells) ;
         double surface_temperature(Time, nCells) ;
         double surface_pressure(Time, nCells) ;
         double coeffs_reconstruct(nCells, maxEdges, R3) ;
         double theta_base(Time, nCells, nVertLevels) ;
         double rho_base(Time, nCells, nVertLevels) ;
         double pressure_base(Time, nCells, nVertLevels) ;
         double exner_base(Time, nCells, nVertLevels) ;
         double exner(Time, nCells, nVertLevels) ;
         double h_divergence(Time, nCells, nVertLevels) ;
         double uReconstructMeridional(Time, nCells, nVertLevels) ;
         double uReconstructZonal(Time, nCells, nVertLevels) ;
         double uReconstructZ(Time, nCells, nVertLevels) ;
         double uReconstructY(Time, nCells, nVertLevels) ;
```

(continues on next page)

(continued from previous page)

```
        double uReconstructX(Time, nCells, nVertLevels) ;
        double pv_cell(Time, nCells, nVertLevels) ;
        double pv_vertex(Time, nVertices, nVertLevels) ;
        double ke(Time, nCells, nVertLevels) ;
        double rho_edge(Time, nEdges, nVertLevels) ;
        double pv_edge(Time, nEdges, nVertLevels) ;
        double vorticity(Time, nVertices, nVertLevels) ;
        double divergence(Time, nCells, nVertLevels) ;
        double v(Time, nEdges, nVertLevels) ;
        double rh(Time, nCells, nVertLevels) ;
        double theta(Time, nCells, nVertLevels) ;
        double rho(Time, nCells, nVertLevels) ;
        double qv_init(nVertLevels) ;
        double t_init(nCells, nVertLevels) ;
        double u_init(nVertLevels) ;
        double pressure_p(Time, nCells, nVertLevels) ;
        double tend_theta(Time, nCells, nVertLevels) ;
        double tend_rho(Time, nCells, nVertLevels) ;
        double tend_w(Time, nCells, nVertLevelsP1) ;
        double tend_u(Time, nEdges, nVertLevels) ;
        double qv(Time, nCells, nVertLevels) ;
        double qc(Time, nCells, nVertLevels) ;
        double qr(Time, nCells, nVertLevels) ;
        double qi(Time, nCells, nVertLevels) ;
        double qs(Time, nCells, nVertLevels) ;
        double qg(Time, nCells, nVertLevels) ;
        double tend_qg(Time, nCells, nVertLevels) ;
        double tend_qs(Time, nCells, nVertLevels) ;
        double tend_qi(Time, nCells, nVertLevels) ;
        double tend_qr(Time, nCells, nVertLevels) ;
        double tend_qc(Time, nCells, nVertLevels) ;
        double tend_qv(Time, nCells, nVertLevels) ;
        double qnr(Time, nCells, nVertLevels) ;
        double qni(Time, nCells, nVertLevels) ;
        double tend_qnr(Time, nCells, nVertLevels) ;
        double tend_qni(Time, nCells, nVertLevels) ;
```

6.43.7 Files

filename	purpose
input.nml	to read the namelist - model_mod_nml and mpas_vars_nml
mpas_init.nc	provides model state, and ‘valid_time’ of the model state
static.nc	provides grid dimensions
true_state.nc	the time-history of the “true” model state from an OSSE
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.43.8 References

The Data Assimilation section in the MPAS documentation found at http://mpas-dev.github.io.

6.44 MPAS OCN

6.44.1 Overview

The **MPAS OCN** interface for **Data Assimilation Research Testbed (DART)** is under development.

Since MPAS OCN uses netcdf files for their restart mechanism, a namelist-controlled set of variables is used to build the DART state vector. Each variable must also correspond to a DART "KIND"; required for the DART interpolate routines. For example:

```
&mpas_vars_nml
   mpas_state_variables = 'uReconstructZonal',      'QTY_U_WIND_COMPONENT',
                          'uReconstructMeridional', 'QTY_V_WIND_COMPONENT',
                          'w',                      'QTY_VERTICAL_VELOCITY',
                          'theta',                  'QTY_POTENTIAL_TEMPERATURE',
                          'qv',                     'QTY_VAPOR_MIXING_RATIO',
                          'qc',                     'QTY_CLOUDWATER_MIXING_RATIO',
                          'qr',                     'QTY_RAINWATER_MIXING_RATIO',
                          'qi',                     'QTY_ICE_MIXING_RATIO',
                          'qs',                     'QTY_SNOW_MIXING_RATIO',
                          'qg',                     'QTY_GRAUPEL_MIXING_RATIO',
                          'surface_pressure',       'QTY_SURFACE_PRESSURE'
   /
```

These variables are then adjusted to be consistent with observations and stuffed back into the same netCDF analysis files. Since DART is an ensemble algorithm, there are multiple analysis files for a single analysis time: one for each ensemble member. Creating the initial ensemble of states is an area of active research.

DART reads grid information from the MPAS OCN 'history' file, I have tried to keep the variable names the same. Internal to the DART code, the following variables exist:

integer :: nCells	the number of Cell Centers
integer :: nEdges	the number of Cell Edges
integer :: nVertices	the number of Cell Vertices
integer :: nVertLevels	the number of vertical level midpoints
integer :: nVertLevelsP1	the number of vertical level edges
integer :: nSoilLevels	the number of soil level ?midpoints?
real(r8) :: latCell(:)	the latitudes of the Cell Centers (-90,90)
real(r8) :: lonCell(:)	the longitudes of the Cell Centers [0, 360)
real(r8) :: zgrid(:,:)	cell center geometric height at cell centers (ncells,nvert)
integer :: CellsOnVertex(:,:)	list of cell centers defining a triangle

6.44.2 model_mod variable storage

`input.nml&mpas_vars_nml` defines the list of MPAS variables used to build the DART state vector. Combined with an MPAS analysis file, the information is used to determine the size of the DART state vector and derive the metadata. To keep track of what variables are contained in the DART state vector, an array of a user-defined type called "progvar" is available with the following components:

```
type progvartype
   private
   character(len=NF90_MAX_NAME) :: varname
   character(len=NF90_MAX_NAME) :: long_name
   character(len=NF90_MAX_NAME) :: units
   character(len=NF90_MAX_NAME), dimension(NF90_MAX_VAR_DIMS) :: dimname
   integer, dimension(NF90_MAX_VAR_DIMS) :: dimlens
   integer :: xtype         ! netCDF variable type (NF90_double, etc.)
   integer :: numdims       ! number of dims - excluding TIME
   integer :: numvertical   ! number of vertical levels in variable
   integer :: numcells      ! number of horizontal locations (typically cell centers)
   logical :: ZonHalf       ! vertical coordinate has dimension nVertLevels
   integer :: varsize       ! prod(dimlens(1:numdims))
   integer :: index1        ! location in dart state vector of first occurrence
   integer :: indexN        ! location in dart state vector of last  occurrence
   integer :: dart_kind
   character(len=paramname_length) :: kind_string
   logical  :: clamping     ! does variable need to be range-restricted before
   real(r8) :: range(2)     ! being stuffed back into MPAS analysis file.
end type progvartype

type(progvartype), dimension(max_state_variables) :: progvar
```

The variables are simply read from the MPAS analysis file and stored in the DART state vector such that all quantities for one variable are stored contiguously. Within each variable; they are stored vertically-contiguous for each horizontal location. From a storage standpoint, this would be equivalent to a Fortran variable dimensioned x(nVertical,nHorizontal,nVariables). The fastest-varying dimension is vertical, then horizontal, then variable … naturally, the DART state vector is 1D. Each variable is also stored this way in the MPAS analysis file.

The DART interface for MPAS (atm)

was compiled with the gfortran 4.2.3 compilers and run on a Mac.
The DART components were built with the following `mkmf.template` settings:

```
FC = gfortran
LD = gfortran
NETCDF = /Users/thoar/GNU
INCS = -I${NETCDF}/include
LIBS = -L${NETCDF}/lib -lnetcdf -lcurl -lhdf5_hl -lhdf5 -lz  -lm
FFLAGS = -O0 -fbounds-check -frecord-marker=4 -ffpe-trap=invalid $(INCS)
LDFLAGS = $(FFLAGS) $(LIBS)
```

Converting between DART files and MPAS analysis files

is relatively straighforward. Given the namelist mechanism for determining the state variables and the MPAS history netCDF files exist, - everything that is needed is readily determined.

There are two programs - both require the list of MPAS variables to use in the DART state vector: the `mpas_vars_nml` namelist in the `input.nml` file. The MPAS file name being read and/or written is - in all instances - specified by the `model_nml:model_analysis_filename` variable in the `input.nml` namelist file.

PROGRAM model_to_dart for MPAS OCN	converts an MPAS analysis file (nominally named `mpas_analysis.nc`) into a DART-compatible file normally called `dart_ics` . We usually wind up linking the actual analysis file to a static name that is used by DART.
dart_to_model.f90	inserts the DART output into an existing MPAS analysis netCDF file by overwriting the variables in the analysis netCDF file. There are two different types of DART output files, so there is a namelist option to specify if the DART file has two time records or just one (if there are two, the first one is the 'advance_to' time, followed by the 'valid_time' of the ensuing state). `dart_to_model` updates the MPAS analysis file specified in `input.nmlmodel_nml:model_analysis_filename`. If the DART file contains an 'advance_to' time, separate control information is written to an auxiliary file that is used by the `advance_model.csh` script.

The header of an MPAS analysis file is presented below - simply for context. Keep in mind that **many** variables have been removed for clarity. Also keep in mind that the multi-dimensional arrays listed below have the dimensions reversed from the Fortran convention.

```
366 mirage2:thoar% ncdump -h mpas_analysis.nc
netcdf mpas_analysis {
dimensions:
        StrLen = 64 ;
        Time = UNLIMITED ; // (1 currently)
        nCells = 10242 ;                                    available in DART
        nEdges = 30720 ;                                    available in DART
        maxEdges = 10 ;
        maxEdges2 = 20 ;
        nVertices = 20480 ;                                 available in DART
        TWO = 2 ;
        THREE = 3 ;
        vertexDegree = 3 ;                                  available in DART
        FIFTEEN = 15 ;
        TWENTYONE = 21 ;
        R3 = 3 ;
        nVertLevels = 41 ;                                  available in DART
        nVertLevelsP1 = 42 ;                                available in DART
        nMonths = 12 ;
        nVertLevelsP2 = 43 ;
        nSoilLevels = 4 ;                                   available in DART
variables:
        char xtime(Time, StrLen) ;                          available in DART
        double latCell(nCells) ;                            available in DART
        double lonCell(nCells) ;                            available in DART
        double latEdge(nEdges) ;
```

(continues on next page)

(continued from previous page)

```
        double lonEdge(nEdges) ;
        int indexToEdgeID(nEdges) ;
        double latVertex(nVertices) ;
        double lonVertex(nVertices) ;
        int indexToVertexID(nVertices) ;
        int cellsOnEdge(nEdges, TWO) ;
        int nEdgesOnCell(nCells) ;
        int nEdgesOnEdge(nEdges) ;
        int edgesOnCell(nCells, maxEdges) ;
        int edgesOnEdge(nEdges, maxEdges2) ;
        double weightsOnEdge(nEdges, maxEdges2) ;
        double dvEdge(nEdges) ;
        double dcEdge(nEdges) ;
        double angleEdge(nEdges) ;
        double edgeNormalVectors(nEdges, R3) ;
        double cellTangentPlane(nEdges, TWO, R3) ;
        int cellsOnCell(nCells, maxEdges) ;
        int verticesOnCell(nCells, maxEdges) ;
        int verticesOnEdge(nEdges, TWO) ;
        int edgesOnVertex(nVertices, vertexDegree) ;
        int cellsOnVertex(nVertices, vertexDegree) ;      available in DART
        double kiteAreasOnVertex(nVertices, vertexDegree) ;
        double rainc(Time, nCells) ;
        double cuprec(Time, nCells) ;
        double cutop(Time, nCells) ;
        double cubot(Time, nCells) ;
        double relhum(Time, nCells, nVertLevels) ;
        double qsat(Time, nCells, nVertLevels) ;
        double graupelnc(Time, nCells) ;
        double snownc(Time, nCells) ;
        double rainnc(Time, nCells) ;
        double graupelncv(Time, nCells) ;
        double snowncv(Time, nCells) ;
        double rainncv(Time, nCells) ;
        double sr(Time, nCells) ;
        double surface_temperature(Time, nCells) ;
        double surface_pressure(Time, nCells) ;
        double coeffs_reconstruct(nCells, maxEdges, R3) ;
        double theta_base(Time, nCells, nVertLevels) ;
        double rho_base(Time, nCells, nVertLevels) ;
        double pressure_base(Time, nCells, nVertLevels) ;
        double exner_base(Time, nCells, nVertLevels) ;
        double exner(Time, nCells, nVertLevels) ;
        double h_divergence(Time, nCells, nVertLevels) ;
        double uReconstructMeridional(Time, nCells, nVertLevels) ;
        double uReconstructZonal(Time, nCells, nVertLevels) ;
        double uReconstructZ(Time, nCells, nVertLevels) ;
        double uReconstructY(Time, nCells, nVertLevels) ;
        double uReconstructX(Time, nCells, nVertLevels) ;
        double pv_cell(Time, nCells, nVertLevels) ;
        double pv_vertex(Time, nVertices, nVertLevels) ;
        double ke(Time, nCells, nVertLevels) ;
        double rho_edge(Time, nEdges, nVertLevels) ;
        double pv_edge(Time, nEdges, nVertLevels) ;
        double vorticity(Time, nVertices, nVertLevels) ;
        double divergence(Time, nCells, nVertLevels) ;
        double v(Time, nEdges, nVertLevels) ;
```

(continues on next page)

(continued from previous page)

```
        double rh(Time, nCells, nVertLevels) ;
        double theta(Time, nCells, nVertLevels) ;
        double rho(Time, nCells, nVertLevels) ;
        double qv_init(nVertLevels) ;
        double t_init(nCells, nVertLevels) ;
        double u_init(nVertLevels) ;
        double pressure_p(Time, nCells, nVertLevels) ;
        double tend_theta(Time, nCells, nVertLevels) ;
        double tend_rho(Time, nCells, nVertLevels) ;
        double tend_w(Time, nCells, nVertLevelsP1) ;
        double tend_u(Time, nEdges, nVertLevels) ;
        double qv(Time, nCells, nVertLevels) ;
        double qc(Time, nCells, nVertLevels) ;
        double qr(Time, nCells, nVertLevels) ;
        double qi(Time, nCells, nVertLevels) ;
        double qs(Time, nCells, nVertLevels) ;
        double qg(Time, nCells, nVertLevels) ;
        double tend_qg(Time, nCells, nVertLevels) ;
        double tend_qs(Time, nCells, nVertLevels) ;
        double tend_qi(Time, nCells, nVertLevels) ;
        double tend_qr(Time, nCells, nVertLevels) ;
        double tend_qc(Time, nCells, nVertLevels) ;
        double tend_qv(Time, nCells, nVertLevels) ;
        double qnr(Time, nCells, nVertLevels) ;
        double qni(Time, nCells, nVertLevels) ;
        double tend_qnr(Time, nCells, nVertLevels) ;
        double tend_qni(Time, nCells, nVertLevels) ;
```

6.44.3 Namelist

We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input. Consider yourself forewarned that character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
namelist /model_nml/  model_analysis_filename, &
          assimilation_period_days, assimilation_period_seconds, &
          model_perturbation_amplitude, output_state_vector, calendar, debug
```

This namelist is read in a file called `input.nml`. This namelist provides control over the assimilation period for the model. All observations within (+/-) half of the assimilation period are assimilated. The assimilation period is the minimum amount of time the model can be advanced, and checks are performed to ensure that the assimilation window is a multiple of the model dynamical timestep. This also specifies the MPAS analysis file that will be read and/or written by the different program units.

Contents	Type	Description
model_analysis_filename	character(len=256) *[default: 'mpas_analysis.nc']*	Character string specifying the name of the MPAS analysis file to be read and/or written by the different program units.
output_state_vector	logical *[default: .true.]*	The switch to determine the form of the state vector in the output netCDF files. If `.true.` the state vector will be output exactly as DART uses it … one long array. If `.false.`, the state vector is parsed into prognostic variables and output that way – much easier to use with 'ncview', for example.
assimilation_period_days	integer *[default: 1]*	The number of days to advance the model for each assimilation.
assimilation_period_seconds	integer *[default: 0]*	In addition to `assimilation_period_days`, the number of seconds to advance the model for each assimilation.
model_perturbation_amplitude	real(r8) *[default: 0.2]*	Reserved for future use.
calendar	character(len=32) *[default: 'Gregorian']*	Character string specifying the calendar being used by MPAS.
debug	integer *[default: 0]*	The switch to specify the run-time verbosity. `0` is as quiet as it gets. > `1` provides more run-time messages. > `5` provides ALL run-time messages.

Example namelist

```
&model_nml
   model_analysis_filename      = 'mpas_restart.nc';
   assimilation_period_days     = 0,
   assimilation_period_seconds  = 60,
   model_perturbation_amplitude = 0.2,
   output_state_vector          = .true.,
   calendar                     = 'Gregorian',
   debug                        = 0
   /
```

```
namelist /mpas_vars_nml/ mpas_state_variables
```

This namelist is read from `input.nml` and contains the list of MPAS variables that make up the DART state vector.

Contents	Type	Description
mpas_vars_nml	character(len=NF90_MAX_NAME):: dimension(160) *[default: see example]*	The table that relates the GITM variables to use to build the DART state vector, and the corresponding DART kinds for those variables.

Example

The following mpas_vars_nml is just for demonstration purposes. You application will likely involve a different DART state vector.

```
&mpas_vars_nml
   mpas_state_variables = 'theta',                 'QTY_POTENTIAL_TEMPERATURE',
                          'uReconstructZonal',     'QTY_U_WIND_COMPONENT',
                          'uReconstructMeridional','QTY_V_WIND_COMPONENT',
                          'w',                     'QTY_VERTICAL_VELOCITY',
                          'qv',                    'QTY_VAPOR_MIXING_RATIO',
                          'qc',                    'QTY_CLOUDWATER_MIXING_RATIO',
                          'qr',                    'QTY_RAINWATER_MIXING_RATIO',
                          'qi',                    'QTY_ICE_MIXING_RATIO',
                          'qs',                    'QTY_SNOW_MIXING_RATIO',
                          'qg',                    'QTY_GRAUPEL_MIXING_RATIO'
                          'surface_pressure',      'QTY_SURFACE_PRESSURE'
   /
```

The variables are simply read from the MPAS analysis file and stored in the DART state vector such that all quantities for one variable are stored contiguously. Within each variable; they are stored vertically-contiguous for each horizontal location. From a storage standpoint, this would be equivalent to a Fortran variable dimensioned x(nVertical,nHorizontal,nVariables). The fastest-varying dimension is vertical, then horizontal, then variable … naturally, the DART state vector is 1D. Each variable is also stored this way in the MPAS analysis file.

6.44.4 Other modules used

```
types_mod
time_manager_mod
threed_sphere/location_mod
utilities_mod
obs_kind_mod
mpi_utilities_mod
random_seq_mod
```

> **Warning:** DAReS staff began creating the MPAS_OCN interface to DART in preparation for the model's inclusion as the ocean component of the Community Earth System Model (CESM). The plans for including MPAS_OCN in CESM were abandoned and the Modular Ocean Model version 6 (MOM6) was included instead. Thus, the documentation on this page after this point describes an incomplete interface. Please contact DAReS staff by emailing dart@ucar.edu if you want to use DART with MPAS_OCN.

6.44.5 Public interfaces

Only a select number of interfaces used are discussed here. Each module has its own discussion of their routines.

Required interface routines

use model_mod, only :	get_model_size
	adv_1step
	get_state_meta_data
	model_interpolate
	get_model_time_step
	static_init_model
	end_model
	init_time
	init_conditions
	nc_write_model_atts
	nc_write_model_vars
	pert_model_state
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	ens_mean_for_model

Unique interface routines

use model_mod, only :	get_gridsize
	restart_file_to_sv
	sv_to_restart_file
	get_gitm_restart_filename
	get_base_time
	get_state_time

use location_mod, only :	get_close_o bs

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

Interface routine descriptions

model_size = get_model_size( )

```
integer :: get_model_size
```

Returns the length of the model state vector. Required.

`model_size`	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

`adv_1step` is not used for the gitm model. Advancing the model is done through the `advance_model` script. This is a NULL_INTERFACE, provided only for compatibility with the DART requirements.

`x`	State vector of length model_size.
`time`	Specifies time of the initial model state.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,             intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,   intent(out) ::  var_type
```

`get_state_meta_data` returns metadata about a given element of the DART representation of the model state vector. Since the DART model state vector is a 1D array and the native model grid is multidimensional, `get_state_meta_data` returns information about the native model state vector representation. Things like the `location`, or the type of the variable (for instance: temperature, u wind component, . . .). The integer values used to indicate different variable types in `var_type` are themselves defined as public interfaces to model_mod if required.

`index_in`	Index of state vector element about which information is requested.
`location`	Returns the 3D location of the indexed state variable. The `location_ type` comes from `DART/assimilation_code/location/threed_sphere/location_mod.f90`. Note that the lat/lon are specified in degrees by the user but are converted to radians internally.
var_type	Returns the type of the indexed state variable as an optional argument. The type is one of the list of supported observation types, found in the block of code starting `! Integer definitions for DART TYPES` in `DART/assimilation_code/modules/observations/obs_kind_mod.f90`

The list of supported variables in `DART/assimilation_code/modules/observations/obs_kind_mod.f90` is created by `preprocess`.

call model_interpolate(x, location, itype, obs_val, istatus)

```
real(r8), dimension(:), intent(in)  :: x
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: itype
real(r8),               intent(out) :: obs_val
integer,                intent(out) :: istatus
```

Given a model state, `model_interpolate` returns the value of the desired observation type (which could be a state variable) that would be observed at the desired location. The interpolation method is either completely specified by the model, or uses some standard 2D or 3D scalar interpolation routines. Put another way, `model_interpolate` will apply the forward operator **H** to the model state to create an observation at the desired location.

If the interpolation is valid, `istatus = 0`. In the case where the observation operator is not defined at the given location (e.g. the observation is below the lowest model level, above the top level, or 'dry'), interp_val is returned as 0.0 and istatus = 1.

`x`	A model state vector.
`location`	Location to which to interpolate.
`itype`	Integer indexing which type of observation is desired.
`obs_val`	The interpolated value from the model.
`istatus`	Integer flag indicating the success of the interpolation. success == 0, failure == anything else

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

`get_model_time_step` returns the forecast length to be used as the "model base time step" in the filter. This is the minimum amount of time the model can be advanced by `filter`. *This is also the assimilation window*. All observations within (+/-) one half of the forecast length are used for the assimilation. In the `GITM` case, this is set from the namelist values for `input.nml&model_nml:assimilation_period_days, assimilation_period_seconds`.

`var`	Smallest time step of model.

call static_init_model()

`static_init_model` is called for runtime initialization of the model. The namelists are read to determine runtime configuration of the model, the grid coordinates, etc. There are no input arguments and no return values. The routine sets module-local private attributes that can then be queried by the public interface routines.

See the GITM documentation for all namelists in `gitm_in` . Be aware that DART reads the GITM `&grid_nml` namelist to get the filenames for the horizontal and vertical grid information as well as the topography information.

The namelists (all mandatory) are:

`input.nml&model_mod_nml`,

`gitm_in&time_manager_nml`,

`gitm_in&io_nml`,

`gitm_in&init_ts_nml`,

`gitm_in&restart_nml`,

`gitm_in&domain_nml`, and

`gitm_in&grid_nml`.

call end_model()

`end_model` is used to clean up storage for the model, etc. when the model is no longer needed. There are no arguments and no return values. The grid variables are deallocated.

call init_time(time)

```
type(time_type), intent(out) :: time
```

`init_time` returns the time at which the model will start if no input initial conditions are to be used. This is frequently used to spin-up models from rest, but is not meaningfully supported for the GITM model. The only time this routine would get called is if the `input.nml&perfect_model_obs_nml:start_from_restart` is .false., which is not supported in the GITM model.

`time`	the starting time for the model if no initial conditions are to be supplied. This is hardwired to 0.0

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

`init_conditions` returns default initial conditions for model; generally used for spinning up initial model states. For the GITM model it is just a stub because the initial state is always provided by the input files.

`x`	Initial conditions for state vector. This is hardwired to 0.0

ierr = nc_write_model_atts(ncFileID)

```
integer              :: nc_write_model_atts
integer, intent(in)  :: ncFileID
```

`nc_write_model_atts` writes model-specific attributes to an opened netCDF file: In the GITM case, this includes information like the coordinate variables (the grid arrays: ULON, ULAT, TLON, TLAT, ZG, ZC, KMT, KMU), information from some of the namelists, and either the 1D state vector or the prognostic variables (SALT,TEMP,UVEL,VVEL,PSURF). All the required information (except for the netCDF file identifier) is obtained from the scope of the `model_mod` module. Both the `input.nml` and `gitm_in` files are preserved in the netCDF file as variables `inputnml` and `gitm_in`, respectively.

`ncFileID`	Integer file descriptor to previously-opened netCDF file.
`ierr`	Returns a 0 for successful completion.

`nc_write_model_atts` is responsible for the model-specific attributes in the following DART-output netCDF files: `true_state.nc`, `preassim.nc`, and `analysis.nc`.

ierr = nc_write_model_vars(ncFileID, statevec, copyindex, timeindex)

```
integer,                 intent(in) :: ncFileID
real(r8), dimension(:),  intent(in) :: statevec
integer,                 intent(in) :: copyindex
integer,                 intent(in) :: timeindex
integer                             :: ierr
```

nc_write_model_vars writes a copy of the state variables to a NetCDF file. Multiple copies of the state for a given time are supported, allowing, for instance, a single file to include multiple ensemble estimates of the state. Whether the state vector is parsed into prognostic variables (SALT, TEMP, UVEL, VVEL, PSURF) or simply written as a 1D array is controlled by input.nml&model_mod_nml:output_state_vector. If output_state_vector = .true. the state vector is written as a 1D array (the simplest case, but hard to explore with the diagnostics). If output_state_vector = .false. the state vector is parsed into prognostic variables before being written.

ncFileID	file descriptor to previously-opened netCDF file.
statevec	A model state vector.
copyindex	Integer index of copy to be written.
timeindex	The timestep counter for the given state.
ierr	Returns 0 for normal completion.

call pert_model_state(state, pert_state, interf_provided)

```
real(r8), dimension(:), intent(in)  :: state
real(r8), dimension(:), intent(out) :: pert_state
logical,                intent(out) :: interf_provided
```

Given a model state, pert_model_state produces a perturbed model state. This is used to generate ensemble initial conditions perturbed around some control trajectory state when one is preparing to spin-up ensembles. Since the DART state vector for the GITM model contains both 'wet' and 'dry' cells, it is imperative to provide an interface to perturb **just** the wet cells (interf_provided == .true.).

The magnitude of the perturbation is wholly determined by input.nml&model_mod_nml:model_perturbation_amplitude and **utterly, completely fails**.

A more robust perturbation mechanism is needed. Until then, avoid using this routine by using your own ensemble of initial conditions. This is determined by setting input.nml&filter_nml:start_from_restart = .false.

state	State vector to be perturbed.
pert_state	The perturbed state vector.
interf_provided	Because of the 'wet/dry' issue discussed above, this is always .true., indicating a model-specific perturbation is available.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
real(r8),             intent(in)    :: maxdist
```

Pass-through to the 3-D sphere locations module. See get_close_maxdist_init() for the documentation of this subroutine.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(num)
```

Pass-through to the 3-D sphere locations module. See get_close_obs_init() for the documentation of this subroutine.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, & num_close, close_ind [, dist])

```
type(get_close_type),                intent(in ) :: gc
type(location_type),                 intent(in ) :: base_obs_loc
integer,                             intent(in ) :: base_obs_kind
type(location_type), dimension(:),   intent(in ) :: obs
integer,             dimension(:),   intent(in ) :: obs_kind
integer,                             intent(out) :: num_close
integer,             dimension(:),   intent(out) :: close_ind
real(r8), optional,  dimension(:),   intent(out) :: dist
```

Given a DART location (referred to as “base”) and a set of locations, and a definition of ‘close’ - return a subset of locations that are ‘close’, as well as their distances to the DART location and their indices. This routine intentionally masks a routine of the same name in `location_mod` because we want to be able to discriminate against selecting ‘dry land’ locations.

Given a single location and a list of other locations, returns the indices of all the locations close to the single one along with the number of these and the distances for the close ones. The list of locations passed in via the `obs` argument must be identical to the list of `obs` passed into the most recent call to `get_close_obs_init()`. If the list of locations of interest changes, `get_close_obs_destroy()` must be called and then the two initialization routines must be called before using `get_close_obs()` again.

For vertical distance computations, the general philosophy is to convert all vertical coordinates to a common coordinate. This coordinate type is defined in the namelist with the variable “vert_localization_coord”.

`gc`	Structure to allow efficient identification of locations ‘close’ to a given location.
`base_obs_loc`	Single given location.
`base_obs_kind`	Kind of the single location.
`obs`	List of candidate locations.
`obs_kind`	Kind associated with candidate locations.
`num_close`	Number of locations close to the given location.
`close_ind`	Indices of those locations that are close.
dist	Distance between given location and the close ones identified in close_ind.

call ens_mean_for_model(ens_mean)

```
real(r8), dimension(:), intent(in) :: ens_mean
```

`ens_mean_for_model` normally saves a copy of the ensemble mean to module-local storage. This is a NULL_INTERFACE for the GITM model. At present there is no application which requires module-local storage of the ensemble mean. No storage is allocated.

`ens_mean`	State vector containing the ensemble mean.

Unique interface routine descriptions

call get_gridsize(num_x, num_y, num_z)

```
integer, intent(out) :: num_x, num_y, num_z
```

`get_gridsize` returns the dimensions of the compute domain. The horizontal gridsize is determined from `gitm_restart.nc`.

`num_x`	The number of longitudinal gridpoints.
`num_y`	The number of latitudinal gridpoints.
`num_z`	The number of vertical gridpoints.

call restart_file_to_sv(filename, state_vector, model_time)

```
character(len=*),       intent(in)    :: filename
real(r8), dimension(:), intent(inout) :: state_vector
type(time_type),        intent(out)   :: model_time
```

`restart_file_to_sv` Reads a GITM netCDF format restart file and packs the desired variables into a DART state vector. The desired variables are specified in the `gitm_vars_nml` namelist.

`filename`	The name of the netCDF format GITM restart file.
`state_vector`	the 1D array containing the concatenated GITM variables.
`model_time`	the time of the model state. The last time in the netCDF restart file.

call sv_to_restart_file(state_vector, filename, statedate)

```
real(r8), dimension(:), intent(in) :: state_vector
character(len=*),       intent(in) :: filename
type(time_type),        intent(in) :: statedate
```

sv_to_restart_file updates the variables in the GITM restart file with values from the DART vector state_vector. The last time in the file must match the statedate.

filename	the netCDF-format GITM restart file to be updated.
state_vector	the 1D array containing the DART state vector.
statedate	the 'valid_time' of the DART state vector.

call get_gitm_restart_filename(filename)

```
character(len=*), intent(out) :: filename
```

get_gitm_restart_filename returns the name of the gitm restart file - the filename itself is in private module storage.

filename	The name of the GITM restart file.

time = get_base_time(filehandle)

```
integer,           intent(in) :: filehandle -OR-
character(len=*), intent(in) :: filehandle
type(time_type),  intent(out) :: time
```

get_base_time extracts the start time of the experiment as contained in the netCDF restart file. The file may be specified by either a character string or the integer netCDF fid.

time = get_state_time(filehandle)

```
integer,           intent(in) :: filehandle -OR-
character(len=*), intent(in) :: filehandle
type(time_type),  intent(out) :: time
```

get_state_time extracts the time of the model state as contained in the netCDF restart file. In the case of multiple times in the file, the last time is the time returned. The file may be specified by either a character string or the integer netCDF fid.

6.44.6 Files

filename	purpose
input.nml	to read the model_mod namelist
gitm_vars.nml	to read the `gitm_vars_nml` namelist
gitm_restart.nc	provides grid dimensions, model state, and 'valid_time' of the model state
true_state.nc	the time-history of the "true" model state from an OSSE
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.44.7 References

- none

6.44.8 Private components

N/A

6.45 NCOMMAS

6.45.1 Overview

NCOMMAS 7_1 may now be used with the **Data Assimilation Research Testbed (DART)**.
Since NCOMMAS uses netCDF files or their restart mechanisms, it was possible to make a namelist-controlled set of variables to be included in the DART state vector. Each variable must also correspond to a DART "KIND"; required for the DART interpolate routines. For example,

```
&ncommas_vars_nml
   ncommas_state_variables = 'U',   'QTY_U_WIND_COMPONENT',
                             'V',   'QTY_V_WIND_COMPONENT',
                             'W',   'QTY_VERTICAL_VELOCITY',
                             'TH',  'QTY_POTENTIAL_TEMPERATURE',
                             'DBZ', 'QTY_RADAR_REFLECTIVITY',
                             'WZ',  'QTY_VERTICAL_VORTICITY',
                             'PI',  'QTY_EXNER_FUNCTION',
                             'QV',  'QTY_VAPOR_MIXING_RATIO',
                             'QC',  'QTY_CLOUDWATER_MIXING_RATIO',
                             'QR',  'QTY_RAINWATER_MIXING_RATIO',
                             'QI',  'QTY_ICE_MIXING_RATIO',
                             'QS',  'QTY_SNOW_MIXING_RATIO',
                             'QH',  'QTY_GRAUPEL_MIXING_RATIO'   /
```

These variables are then adjusted to be consistent with observations and stuffed back into the same netCDF restart files. Since DART is an ensemble algorithm, there are multiple restart files for a single restart time: one for each ensemble member. Creating the initial ensemble of states is an area of active research.

DART reads the grid information for NCOMMAS from the restart file specified in the DART `input.nml&model_nml:ncommas_restart_filename` and checks for the existence and shape of the desired state variables. This not only determines the size of the DART state vector, but DART also inherits much of the metadata for the variables from the NCOMMAS restart file. When DART is responsible for starting/stopping NCOMMAS, the information is conveyed through the command line arguments to NCOMMAS.

NCOMMAS 7_1

was compiled with the Intel 10.1 compilers and run on a linux cluster running SLES10. Initially, DART simply runs 'end-to-end' at every assimilation time, while the NCOMMAS ensemble mechanism is responsible for slicing and dicing the observation sequences and running `correct_ensemble` at the desired times. This is a complete role-reversal from the normal DART operation.

The DART components were built with the following settings:

```
MPIFC = mpif90
MPILD = mpif90
FC = ifort
LD = ifort
INCS = -I/coral/local/netcdf-3.6.3_intel-10.1-64/include
LIBS = -L/coral/local/netcdf-3.6.3_intel-10.1-64/lib -lnetcdf
FFLAGS = -pc64 -fpe0 -mp -O0 -vec-report0 $(INCS)
LDFLAGS = $(FFLAGS) $(LIBS)
```

Converting between DART files and NCOMMAS restart files

is blissfully straighforward. Given the namelist mechanism for determining the state variables and the fact that the NCOMMAS netCDF file has all the grid and time information in it - everything that is needed can be readily determined.

There are two programs - both require the list of NCOMMAS variables to use in the DART state vector: the `ncommas_vars_nml` namelist in the `ncommas_vars.nml` file.

PROGRAM ncommas_to_dart	converts the ncommas restart file `ncommas_restart.nc` into a DART-compatible file normally called `dart_ics` . We usually wind up linking the restart file to a static name that is used by DART.
PROGRAM dart_to_ncommas	inserts the DART output into an existing ncommas restart netCDF file by overwriting the variables in the ncommas restart netCDF file. There are two different types of DART output files, so there is a namelist option to specify if the DART file has two time records or just one (if there are two, the first one is the 'advance_to' time, followed by the 'valid_time' of the ensuing state). `dart_to_ncommas` determines the ncommas restart file name from the `input.nml model_nml:ncommas_restart_filename`. If the DART file contains an 'advance_to' time, `dart_to_ncommas` creates a new `&time_manager_nml` for ncommas in a file called `ncommas_in.DART` which can be used to control the length of the ncommas integration.

Generating the initial ensemble

Creating the initial ensemble is an area of active research. The ncommas model cannot take one single model state and generate its own ensemble (typically done with pert_model_state).

The ensemble has to come from 'somewhere else'. At present, it may be sufficient to use a climatological ensemble; e.g., using the ncommas restarts for '1 January 00Z' from 50 consecutive years from a hindcast experiment.

There is **not yet** a `shell_scripts/MakeInitialEnsemble.csh` script to demonstrate how to convert a set of ncommas netCDF restart files into a set of DART files that have a consistent timestamp. If you simply convert each ncommas file to a DART file using `ncommas_to_dart`, each DART file will have a 'valid time' that reflects the ncommas time of that state - instead of an ensemble of states reflecting one single time. The restart_file_utility can be used to overwrite the timestep in the header of each DART initial conditions file. The namelist for this program must look something like:

```
&restart_file_tool_nml
  input_file_name              = "dart_input",
  output_file_name             = "dart_output",
  ens_size                     = 1,
  single_restart_file_in       = .true.,
  single_restart_file_out      = .true.,
  write_binary_restart_files   = .true.,
  overwrite_data_time          = .true.,
  new_data_days                = 145731,
  new_data_secs                = 0,
  input_is_model_advance_file  = .false.,
  output_is_model_advance_file = .false.,
  overwrite_advance_time       = .false.,
  new_advance_days             = -1,
  new_advance_secs             = -1,
  gregorian_cal                = .true.  /
```

The time of days = *145731* seconds = *0* relates to 00Z 1 Jan 2000 in the DART world.

6.45.2 Namelist

We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input. Consider yourself forewarned that character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
namelist /model_nml/  ncommas_restart_filename, &
          assimilation_period_days, assimilation_period_seconds, &
          model_perturbation_amplitude, output_state_vector, calendar, debug
```

This namelist is read in a file called `input.nml`. This namelist provides control over the assimilation period for the model. All observations within (+/-) half of the assimilation period are assimilated. The assimilation period is the minimum amount of time the model can be advanced, and checks are performed to ensure that the assimilation window is a multiple of the model dynamical timestep.

Contents	Type	Description
output_state_vector	logical *[default: .true.]*	The switch to determine the form of the state vector in the output netCDF files. If `.true.` the state vector will be output exactly as DART uses it … one long array. If `.false.`, the state vector is parsed into prognostic variables and output that way – much easier to use with 'ncview', for example.
assimilation_period_days	integer *[default: 1]*	The number of days to advance the model for each assimilation.
assimilation_period_seconds	integer *[default: 0]*	In addition to `assimilation_period_days`, the number of seconds to advance the model for each assimilation.
model_perturbation_amplitude	real(r8) *[default: 0.2]*	Reserved for future use.
calendar	character(len=32) *[default: 'Gregorian']*	Character string specifying the calendar being used by NCOMMAS.
debug	integer *[default: 0]*	The switch to specify the run-time verbosity. `0` is as quiet as it gets. > `1` provides more run-time messages. > `5` provides ALL run-time messages. All values above 0 will also write a netCDF file of the grid information and perform a grid interpolation test.

Example model namelist

```
&model_nml
   ncommas_restart_filename      = 'ncommas_restart.nc';
   assimilation_period_days      = 1,
   assimilation_period_seconds   = 0,
   model_perturbation_amplitude  = 0.2,
   output_state_vector           = .true.,
   calendar                      = 'Gregorian',
   debug                         = 0
   /
```

```
namelist /ncommas_vars_nml/ ncommas_state_variables
```

This namelist is read in a file called `ncommas_vars.nml` and contains the list of NCOMMAS variables that make up the DART state vector.

Contents	Type	Description
ncommas_state_variables	character(len=NF90_MAX_NAME):: dimension(160) *[default: see example]*	The table that relates the NCOMMAS variables to use to build the DART state vector, and the corresponding DART kinds for those variables.

Ncommas_vars namelist

```
&ncommas_vars_nml
   ncommas_state_variables = 'U',   'QTY_U_WIND_COMPONENT',
                             'V',   'QTY_V_WIND_COMPONENT',
                             'W',   'QTY_VERTICAL_VELOCITY',
                             'TH',  'QTY_POTENTIAL_TEMPERATURE',
                             'DBZ', 'QTY_RADAR_REFLECTIVITY',
                             'WZ',  'QTY_VERTICAL_VORTICITY',
                             'PI',  'QTY_EXNER_FUNCTION',
                             'QV',  'QTY_VAPOR_MIXING_RATIO',
                             'QC',  'QTY_CLOUDWATER_MIXING_RATIO',
                             'QR',  'QTY_RAINWATER_MIXING_RATIO',
                             'QI',  'QTY_ICE_MIXING_RATIO',
                             'QS',  'QTY_SNOW_MIXING_RATIO',
                             'QH',  'QTY_GRAUPEL_MIXING_RATIO'
  /
```

6.45.3 Other modules used

```
types_mod
time_manager_mod
threed_sphere/location_mod
utilities_mod
obs_kind_mod
mpi_utilities_mod
random_seq_mod
```

6.45.4 Public interfaces

Only a select number of interfaces used are discussed here. Each module has its own discussion of their routines.

Required interface routines

use model_mod, only :	get_model_size
	adv_1step
	get_state_meta_data
	model_interpolate
	get_model_time_step
	static_init_model
	end_model
	init_time
	init_conditions
	nc_write_model_atts
	nc_write_model_vars
	pert_model_state
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	ens_mean_for_model

Unique interface routines

use model_mod, only :	get_gridsize
	restart_file_to_sv
	sv_to_restart_file
	get_ncommas_restart_filename
	get_base_time
	get_state_time

use location_mod, only :	get_close_o bs

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

Required interface routines

model_size = get_model_size( )

```
integer :: get_model_size
```

Returns the length of the model state vector. Required.

`model_size`	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

`adv_1step` is not used for the ncommas model. Advancing the model is done through the `advance_model` script. This is a NULL_INTERFACE, provided only for compatibility with the DART requirements.

`x`	State vector of length model_size.
`time`	Specifies time of the initial model state.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,              intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,    intent(out) ::  var_type
```

`get_state_meta_data` returns metadata about a given element of the DART representation of the model state vector. Since the DART model state vector is a 1D array and the native model grid is multidimensional, `get_state_meta_data` returns information about the native model state vector representation. Things like the `location`, or the type of the variable (for instance: temperature, u wind component, . . .). The integer values used to indicate different variable types in `var_type` are themselves defined as public interfaces to model_mod if required.

`index_in`	Index of state vector element about which information is requested.
`location`	Returns the 3D location of the indexed state variable. The `location_ type` comes from `DART/assimilation_code/location/threed_sphere/location_mod.f90`. Note that the lat/lon are specified in degrees by the user but are converted to radians internally.
var_type	Returns the type of the indexed state variable as an optional argument. The type is one of the list of supported observation types, found in the block of code starting `! Integer definitions for DART TYPES` in `DART/assimilation_code/modules/observations/obs_kind_mod.f90`

The list of supported variables in `DART/assimilation_code/modules/observations/obs_kind_mod.f90` is created by `preprocess`.

call model_interpolate(x, location, itype, obs_val, istatus)

```
real(r8), dimension(:), intent(in)  :: x
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: itype
real(r8),               intent(out) :: obs_val
integer,                intent(out) :: istatus
```

Given a model state, `model_interpolate` returns the value of the desired observation type (which could be a state variable) that would be observed at the desired location. The interpolation method is either completely specified by the model, or uses some standard 2D or 3D scalar interpolation routines. Put another way, `model_interpolate` will apply the forward operator **H** to the model state to create an observation at the desired location.

If the interpolation is valid, `istatus = 0`. In the case where the observation operator is not defined at the given location (e.g. the observation is below the lowest model level, above the top level, or 'dry'), interp_val is returned as 0.0 and istatus = 1.

`x`	A model state vector.
`location`	Location to which to interpolate.
`itype`	Integer indexing which type of observation is desired.
`obs_val`	The interpolated value from the model.
`istatus`	Integer flag indicating the success of the interpolation. success == 0, failure == anything else

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

`get_model_time_step` returns the forecast length to be used as the "model base time step" in the filter. This is the minimum amount of time the model can be advanced by `filter`. *This is also the assimilation window*. All observations within (+/-) one half of the forecast length are used for the assimilation. In the `ncommas` case, this is set from the namelist values for `input.nml&model_nml:assimilation_period_days, assimilation_period_seconds`.

`var`	Smallest time step of model.

call static_init_model()

static_init_model is called for runtime initialization of the model. The namelists are read to determine runtime configuration of the model, the grid coordinates, etc. There are no input arguments and no return values. The routine sets module-local private attributes that can then be queried by the public interface routines.

See the ncommas documentation for all namelists in ncommas_in . Be aware that DART reads the ncommas &grid_nml namelist to get the filenames for the horizontal and vertical grid information as well as the topography information.

The namelists (all mandatory) are:

input.nml&model_mod_nml,

ncommas_in&time_manager_nml,

ncommas_in&io_nml,

ncommas_in&init_ts_nml,

ncommas_in&restart_nml,

ncommas_in&domain_nml, and

ncommas_in&grid_nml.

call end_model()

end_model is used to clean up storage for the model, etc. when the model is no longer needed. There are no arguments and no return values. The grid variables are deallocated.

call init_time(time)

```
type(time_type), intent(out) :: time
```

init_time returns the time at which the model will start if no input initial conditions are to be used. This is frequently used to spin-up models from rest, but is not meaningfully supported for the ncommas model. The only time this routine would get called is if the input.nml&perfect_model_obs_nml:start_from_restart is .false., which is not supported in the ncommas model.

time	the starting time for the model if no initial conditions are to be supplied. This is hardwired to 0.0

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

init_conditions returns default initial conditions for model; generally used for spinning up initial model states. For the ncommas model it is just a stub because the initial state is always provided by the input files.

x	Initial conditions for state vector. This is hardwired to 0.0

ierr = nc_write_model_atts(ncFileID)

```
integer              :: nc_write_model_atts
integer, intent(in)  :: ncFileID
```

`nc_write_model_atts` writes model-specific attributes to an opened netCDF file: In the ncommas case, this includes information like the coordinate variables (the grid arrays: ULON, ULAT, TLON, TLAT, ZG, ZC, KMT, KMU), information from some of the namelists, and either the 1D state vector or the prognostic variables (SALT,TEMP,UVEL,VVEL,PSURF). All the required information (except for the netCDF file identifier) is obtained from the scope of the `model_mod` module. Both the `input.nml` and `ncommas_in` files are preserved in the netCDF file as variables `inputnml` and `ncommas_in`, respectively.

`ncFileID`	Integer file descriptor to previously-opened netCDF file.
`ierr`	Returns a 0 for successful completion.

`nc_write_model_atts` is responsible for the model-specific attributes in the following DART-output netCDF files: `true_state.nc`, `preassim.nc`, and `analysis.nc`.

ierr = nc_write_model_vars(ncFileID, statevec, copyindex, timeindex)

```
integer,                intent(in) :: ncFileID
real(r8), dimension(:), intent(in) :: statevec
integer,                intent(in) :: copyindex
integer,                intent(in) :: timeindex
integer                            :: ierr
```

`nc_write_model_vars` writes a copy of the state variables to a NetCDF file. Multiple copies of the state for a given time are supported, allowing, for instance, a single file to include multiple ensemble estimates of the state. Whether the state vector is parsed into prognostic variables (SALT, TEMP, UVEL, VVEL, PSURF) or simply written as a 1D array is controlled by `input.nml&model_mod_nml:output_state_vector`. If `output_state_vector = .true.` the state vector is written as a 1D array (the simplest case, but hard to explore with the diagnostics). If `output_state_vector = .false.` the state vector is parsed into prognostic variables before being written.

`ncFileID`	file descriptor to previously-opened netCDF file.
`statevec`	A model state vector.
`copyindex`	Integer index of copy to be written.
`timeindex`	The timestep counter for the given state.
`ierr`	Returns 0 for normal completion.

call pert_model_state(state, pert_state, interf_provided)

```
real(r8), dimension(:), intent(in)  :: state
real(r8), dimension(:), intent(out) :: pert_state
logical,                intent(out) :: interf_provided
```

Given a model state, `pert_model_state` produces a perturbed model state. This is used to generate ensemble initial conditions perturbed around some control trajectory state when one is preparing to spin-up ensembles. Since the DART state vector for the ncommas model contains both 'wet' and 'dry' cells, it is imperative to provide an interface to perturb **just** the wet cells (`interf_provided == .true.`).

The magnitude of the perturbation is wholly determined by `input.nml&model_mod_nml:model_perturbation_amplitude` and **utterly, completely fails**.

A more robust perturbation mechanism is needed. Until then, avoid using this routine by using your own ensemble of initial conditions. This is determined by setting `input.nml&filter_nml:start_from_restart = .false.`

`state`	State vector to be perturbed.
`pert_state`	The perturbed state vector.
`interf_provided`	Because of the 'wet/dry' issue discussed above, this is always `.true.`, indicating a model-specific perturbation is available.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
real(r8),             intent(in)    :: maxdist
```

Pass-through to the 3-D sphere locations module. See get_close_maxdist_init() for the documentation of this subroutine.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(num)
```

Pass-through to the 3-D sphere locations module. See get_close_obs_init() for the documentation of this subroutine.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, & num_close, close_ind [, dist])

```
type(get_close_type),                intent(in ) :: gc
type(location_type),                 intent(in ) :: base_obs_loc
integer,                             intent(in ) :: base_obs_kind
type(location_type), dimension(:),   intent(in ) :: obs
integer,             dimension(:),   intent(in ) :: obs_kind
integer,                             intent(out) :: num_close
integer,             dimension(:),   intent(out) :: close_ind
real(r8), optional,  dimension(:),   intent(out) :: dist
```

Given a DART location (referred to as “base”) and a set of locations, and a definition of ‘close’ - return a subset of locations that are ‘close’, as well as their distances to the DART location and their indices. This routine intentionally masks a routine of the same name in `location_mod` because we want to be able to discriminate against selecting ‘dry land’ locations.

Given a single location and a list of other locations, returns the indices of all the locations close to the single one along with the number of these and the distances for the close ones. The list of locations passed in via the `obs` argument must be identical to the list of `obs` passed into the most recent call to `get_close_obs_init()`. If the list of locations of interest changes, `get_close_obs_destroy()` must be called and then the two initialization routines must be called before using `get_close_obs()` again.

For vertical distance computations, the general philosophy is to convert all vertical coordinates to a common coordinate. This coordinate type is defined in the namelist with the variable “vert_localization_coord”.

`gc`	Structure to allow efficient identification of locations ‘close’ to a given location.
`base_obs_loc`	Single given location.
`base_obs_kind`	Kind of the single location.
`obs`	List of candidate locations.
`obs_kind`	Kind associated with candidate locations.
`num_close`	Number of locations close to the given location.
`close_ind`	Indices of those locations that are close.
dist	Distance between given location and the close ones identified in close_ind.

call ens_mean_for_model(ens_mean)

```
real(r8), dimension(:), intent(in) :: ens_mean
```

`ens_mean_for_model` normally saves a copy of the ensemble mean to module-local storage. This is a NULL_INTERFACE for the ncommas model. At present there is no application which requires module-local storage of the ensemble mean. No storage is allocated.

`ens_mean`	State vector containing the ensemble mean.

Unique interface routines

call get_gridsize(num_x, num_y, num_z)

```
integer, intent(out) :: num_x, num_y, num_z
```

`get_gridsize` returns the dimensions of the compute domain. The horizontal gridsize is determined from `ncommas_restart.nc`.

`num_x`	The number of longitudinal gridpoints.
`num_y`	The number of latitudinal gridpoints.
`num_z`	The number of vertical gridpoints.

call restart_file_to_sv(filename, state_vector, model_time)

```
character(len=*),        intent(in)    :: filename
real(r8), dimension(:),  intent(inout) :: state_vector
type(time_type),         intent(out)   :: model_time
```

`restart_file_to_sv` Reads a NCOMMAS netCDF format restart file and packs the desired variables into a DART state vector. The desired variables are specified in the `ncommas_vars_nml` namelist.

`filename`	The name of the netCDF format NCOMMAS restart file.
`state_vector`	the 1D array containing the concatenated NCOMMAS variables.
`model_time`	the time of the model state. The last time in the netCDF restart file.

call sv_to_restart_file(state_vector, filename, statedate)

```
real(r8), dimension(:), intent(in) :: state_vector
character(len=*),       intent(in) :: filename
type(time_type),        intent(in) :: statedate
```

`sv_to_restart_file` updates the variables in the NCOMMAS restart file with values from the DART vector `state_vector`. The last time in the file must match the `statedate`.

`filename`	the netCDF-format ncommas restart file to be updated.
`state_vector`	the 1D array containing the DART state vector.
`statedate`	the 'valid_time' of the DART state vector.

call get_ncommas_restart_filename(filename)

```
character(len=*), intent(out) :: filename
```

`get_ncommas_restart_filename` returns the name of the NCOMMAS restart file - the filename itself is in private module storage.

`filename`	The name of the NCOMMAS restart file.

time = get_base_time(filehandle)

```
integer,          intent(in) :: filehandle -OR-
character(len=*), intent(in) :: filehandle
type(time_type),  intent(out) :: time
```

`get_base_time` extracts the start time of the experiment as contained in the netCDF restart file. The file may be specified by either a character string or the integer netCDF fid.

time = get_state_time(filehandle)

```
integer,          intent(in) :: filehandle -OR-
character(len=*), intent(in) :: filehandle
type(time_type),  intent(out) :: time
```

`get_state_time` extracts the time of the model state as contained in the netCDF restart file. In the case of multiple times in the file, the last time is the time returned. The file may be specified by either a character string or the integer netCDF fid.

6.45.5 Files

filename	purpose
input.nml	to read the model_mod namelist
ncommas_vars.nml	to read the `ncommas_vars_nml` namelist
ncommas_restart.nc	provides grid dimensions, model state, and ‘valid_time’ of the model state
true_state.nc	the time-history of the “true” model state from an OSSE
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.45.6 References

- none

6.45.7 Private components

N/A

6.46 NOAH, NOAH-MP

6.46.1 Overview

The Manhattan-compliant version of the NOAH (technically NOAH-MP) supports NOAH-MP V3.6 and was largely updated in support of the data assimilation efforts with **wrf_hydro**. Experiments to perform data assimilation strictly with the NOAH-MP model have been run at the University of Texas at Austin by **Jingjing Liang**. We know other people are using DART and NOAH-MP. *however, we have not had the chance to update the documentation for the Manhattan release.* Consequently, we readily welcome any advice on how to improve the documentation and heartily encourage participation.

The NOAH **Land Surface Model** and **Data Assimilation Research Testbed (DART)** may now be used for assimilation experiments. The Classic or Lanai version should be considered an 'alpha' release – the code has only been tested for a single column configuration of NOAH.

Any of the variables in the NOAH restart file are available to be adjusted by the assimilation. The list of variables is set though a simple namelist interface. Since we are testing in a column configuration, there is no practical reason not to include all the variables necessary for a bit-for-bit restart: *SOIL_T*, *SOIL_M*, *SOIL_W*, *SKINTEMP*, *SNODEP*, *WEASD*, *CANWAT*, and *QFX*. These variables are then adjusted to be consistent with real observations and stuffed back into the same netCDF restart files. Since DART is an ensemble algorithm there are multiple restart files for a single restart time; one for each ensemble member. Creating the initial ensemble of land surface states is an area of active research. At present, it may be sufficient to use a climatological ensemble; e.g., using the restarts for '1 January 00Z' from 50 consecutive years.

There is reason to believe that the ensemble system will benefit from having unique atmospheric forcing for each ensemble member. A reasonable ensemble size is 50 or 80 or so.

DART reads the NOAH namelist `&NOAHLSM_OFFLINE` from a file called `namelist.hrldas` for several pieces of information. DART is responsible for starting/stopping NOAH; the restart information is conveyed through the NOAH namelist. **Unpleasant Reality #1 :** managing the tremendous number of hourly forcing files for every ensemble member is tedious. To facilitate matters, the DART/NOAH system uses a *single* netCDF file for each ensemble member that contains ALL of the forcing for that ensemble member.

dart_to_noah.f90	**updates** some or all of a NOAH restart file with the posterior DART state vector. There is the ability to selectively avoid updating the NOAH variables. This allows one to include NOAH variables in the DART state vector to aid in the application of observation operators, etc., without having to modify those variables in the NOAH restart file. [dart_to_noah.html]

Running a "Perfect Model" experiment . . . OSSE

The example requires a basic knowledge of running NOAH. Four scripts are provided to demonstrate how to set up and run a perfect model experiment for a single site - with one caveat. You must provide your own initial ensemble for the experiment. The scripts are not intended to be black boxes. You are expected to read them and modify them to your own purpose.

The scripts assume the directory containing the DART executables is `${DARTDIR}/work`, and assume that the directory containing the NOAH executables is `${NOAHDIR}/Run`.

1. setup_pmo.csh	This script stages the run of *program perfect_model_obs*. The directory where you run the script is called `CENTRALDIR` and will be the working directory for the experiment. The required input observation sequence file must be created in the normal DART way. This `obs_seq.in` file must exist before running this script. All the necessary data files and exectuables for a perfect model experiment get copied to CENTRALDIR so that you may run multiple experiments at the same time - in separate `CENTRALDIRs`.
2. run_pmo.csh	very simply - it advances NOAH and applies the observation operator to put the "perfect" observations in an observation sequence file that can then be used for an assimilation.
3. setup_filter.csh	builds upon the work of `setup_pmo.csh` and stages a PRE-EXISTING initial ensemble.
4. run_filter.csh	Actually runs the filtering (assimilation) experiment.

Generating the initial ensemble

Creating the initial ensemble of soil moisture states is an area of active research. The ensemble must come from 'somewhere else'. At present, it may be sufficient to use a climatological ensemble; e.g., using the NOAH restarts for '1 January 00Z' from 50 consecutive years from a hindcast experiment. It may also be sufficient to take a single model state, replicate it N times and force each of the N instances with different atmospheric conditions for 'a long time'.

By The Way

Experience has shown that having a paired (unique) atmospheric forcing maintains the ensemble spread during an assimilation better than simply forcing all the ensemble members with one single atmospheric state.

DART has routines to perturb a single NOAH state and generate its own ensemble (typically done with `pert_model_state`), but this produces model states that are incompatible with NOAH. We are interested in adopting/adapting strategies to create sensible initial conditions for NOAH.

If you have an algorithm you believe will be useful, please contact us!

6.46.2 Observations

Some novel observations come from the Cosmic-ray Soil Moisture Observing System: COSMOS and are processed by DART routines in the $DARTROOT/observations/COSMOS directory.

DART has a very object-oriented approach to observation support. All observations that are intended to be supported must be preprocessed (see $DARTROOT/preprocess/ into a single obs_def_mod.f90 and obs_kind_mod.f90 in the standard DART way.

Exploring the Output

There are Matlab® scripts for exploring the performance of the assimilation in observation-space (after running obs_diag). See $DARTROOT/diagnostics/threed_sphere/obs_diag.html to explore the *obs_seq.final* file) - use the scripts starting with plot_, i.e. $DARTROOT/diagnostics/matlab/plot_*.m*. As always, there are some model-specific items Matlab® will need to know about in $DARTROOT/models/NOAH/matlab.

The Prior_Diag.nc and Posterior_Diag.nc (and possibly True_State.nc) netCDF files have the model prognostic variables before and after the assimilation. The ./matlab scripts for NOAH are under development.

It is also worthwhile to convert your obs_seq.final file to a netCDF format obs_sequence file with obs_seq_to_netcdf. See $DARTROOT/obs_sequence/obs_seq_to_netcdf.html and use any of the standard plots. Be aware that the COSMOS site-specific metadata will not get conveyed to the netCDF file.

6.46.3 Namelist

The &model_nml namelist is read from the input.nml file. Namelists start with an ampersand & and terminate with a slash /. Character strings that contain a / must be enclosed in quotes to prevent them from prematurely terminating the namelist. The standard values are shown below:

```
&model_nml
   lsm_model_choice             = 'noahMP_36'
   domain_shapefiles            = 'RESTART.2003051600_DOMAIN1_01'
   assimilation_period_days     =    0
   assimilation_period_seconds  = 3600
   model_perturbation_amplitude = 0.2
   perturb_distribution         = 'gaussian'
   debug                        = 0
   polar                        = .false.
   periodic_x                   = .false.
   periodic_y                   = .false.
   lsm_variables = 'SOIL_T',    'QTY_SOIL_TEMPERATURE',    '0.0',  'NA', 'UPDATE',
                   'SMC',       'QTY_SOIL_MOISTURE',       '0.0', '1.0', 'UPDATE',
                   'WA',        'QTY_AQUIFER_WATER',       '0.0',  'NA', 'UPDATE',
                   'SNEQV',     'QTY_SNOW_WATER',          '0.0',  'NA', 'UPDATE',
                   'FSNO',      'QTY_SNOWCOVER_FRAC',      '0.0', '1.0', 'UPDATE'
  /
```

This namelist is read from a file called input.nml. This namelist provides control over the assimilation period for the model. All observations within (+/-) half of the assimilation period are assimilated. The assimilation period is the minimum amount of time the model can be advanced, and checks are performed to ensure that the assimilation window is a multiple of the NOAH model dynamical timestep.

Item	Type	Description
lsm_model_choice	character(len=256)	The version of the NOAH namelist to read
domain_shapefiles	an array of character(len=256)	The name of the NOAH RESTART files to use to specify the shape of the variables and geographic metadata. One per domain.
assimilation_period_days	integer	The number of days to advance the model for each assimilation.
assimilation_period_seconds	integer	In addition to `assimilation_period_days`, the number of seconds to advance the model for each assimilation.
model_perturbation_amplitude	real(r8)	The amount of noise to add when trying to perturb a single state vector to create an ensemble. Only used when `input.nml` is set with `&filter_nml:start_from_restart = .false.`. See also *Generating the initial ensemble*. units: standard deviation of the specified distribution the mean at the value of the state vector element.
perturb_distribution	character(len=256)	The switch to determine the distribution of the perturbations used to create an initial ensemble from a single model state. Valid values are : `lognormal` or `gaussian`
periodic_x	logical	Switch to determine if the configuration has periodicity in the X direction.
periodic_y	logical	Switch to determine if the configuration has periodicity in the Y direction.
lsm_variables	character(len=32)::dimension(5,40)	The list of variable names in the NOAH restart file to use to create the DART state vector and their corresponding DART kind. [default: see example below]

The columns of `lsm_variables` needs some explanation. Starting with the column 5, `UPDATE` denotes whether or not to replace the variable with the Posterior (i.e. assimilated) value. Columns 3 and 4 denote lower and upper bounds that should be enforced when writing to the files used to restart the model. These limits are not enforced for the DART diagnostic files. Column 2 specifies the relationship between the netCDF variable name for the model and the corresponding DART QUANTITY.

The DART 'QTY's match what the `model_mod` knows how to interpolate, so you can't just add a new quantity and expect it to work. There is a complex interplay between `obs_def_mod` and `preprocess`, and `model_mod` that defines what QUANTITIES are supported. There is only a single QUANTITY that works with each variable and the example shows the current QUANTITYs. Support for these QUANTITYs was provided by running `preprocess` with the following namelist settings:

```
&preprocess_nml
    input_obs_kind_mod_file = '../../../assimilation_code/modules/observations/
↪DEFAULT_obs_kind_mod.F90'
   output_obs_kind_mod_file = '../../../assimilation_code/modules/observations/obs_
↪kind_mod.f90'
     input_obs_def_mod_file = '../../../observations/forward_operators/DEFAULT_obs_
↪def_mod.F90'
    output_obs_def_mod_file = '../../../observations/forward_operators/obs_def_mod.f90
↪'
   input_files              = '../../../observations/forward_operators/obs_def_land_
↪mod.f90',
                              '../../../observations/forward_operators/obs_def_COSMOS_
↪mod.f90',
                              '../../../observations/forward_operators/obs_def_GRACE_
↪mod.f90'
```

(continues on next page)

(continued from previous page)

```
  /
```

NOAHLSM_OFFLINE NAMELIST

```
namelist /NOAHLSM_OFFLINE/
   hrldas_constants_file, &
   indir, outdir,  &
   restart_filename_requested, &
   khour,  kday, &
   forcing_timestep, &
   noah_timestep,  &
   output_timestep, &
   restart_frequency_hours, &
   split_output_count, &
   nsoil, &
   zsoil
```

The remaining variables are not used by DART - but are used by NOAH. Since DART verifies namelist accuracy, any namelist entry in NOAHLSM_OFFLINE that is not in the following list will cause a FATAL DART ERROR.

```
zlvl, zlvl_wind, iz0tlnd, sfcdif_option, update_snow_from_forcing,
start_year, start_month, start_day, start_hour, start_min,
external_fpar_filename_template, external_lai_filename_template,
subwindow_xstart, subwindow_xend, subwindow_ystart, subwindow_yend
```

This namelist is read from a file called `namelist.hrldas`. This namelist is the same one that is used by NOAH. The values are explained in full in the NOAH documentation. Only the namelist variables of interest to DART are discussed. All other namelist variables are ignored by DART - but mean something to NOAH.

Item	Type	Description
hrldas_constants_file	character(len=256)	The name of the netCDF file containing the grid information. [default: `wrfinput`]
indir	character(len=256)	The DART/NOAH environment requires all the input files to be in the current working directory. [default: `'.'`]
outdir	character(len=256)	The DART/NOAH environment requires all output files are in the current working directory. [default: `'.'`]
restart_filename_requested	character(len=256)	The name of the file containing the grid information. The default value is implicitly used by the scripting examples. Change at your own risk. [default: `'restart.nc'`]
khour	integer	The duration (in hours) of the model integration. [default: `1`]
kday	integer	The duration (in days) of the model integration. [default: `0`]
forcing_timestep	integer	The timestep (in seconds) of the atmospheric forcing. [default: `3600`]
noah_timestep	integer	The internal (dynamical) timestep (in seconds). [default: `3600`]
output_timestep	integer	The output interval (in seconds). [default: `3600`]
restart_frequency_hours	integer	How often the NOAH restart files get written. [default: `1`]
split_output_count	integer	should be 1 or bad things happen. [default: `1`]
nsoil	integer	The number of soil interfaces. As I understand it, NOAH requires this to be 4. [default: `4`]
zsoil	integer(NSOLDX)	The depth (in meters) of the soil interfaces. [default: `-0.1, -0.4, -1.0, -2.4`]

Example

Note: the `FORCING_FILE_DIRECTORY` line is not required by NOAH but IS required by DART - specifically in the *advance_model.csh* script.

```
### THIS IS FOR DART ###
FORCING_FILE_DIRECTORY = "/path/to/your/forcing/files"

&NOAHLSM_OFFLINE
   HRLDAS_CONSTANTS_FILE = "wrfinput"
   INDIR  = "."
   OUTDIR = "."
   RESTART_FILENAME_REQUESTED = "restart.nc"
   KHOUR                   = 1
   FORCING_TIMESTEP        = 3600
   NOAH_TIMESTEP           = 3600
   OUTPUT_TIMESTEP         = 3600
   RESTART_FREQUENCY_HOURS = 1
   SPLIT_OUTPUT_COUNT      = 1
   NSOIL=4
   ZSOIL(1) = -0.10
   ZSOIL(2) = -0.40
   ZSOIL(3) = -1.00
   ZSOIL(4) = -2.00
/
```

6.46.4 Input Files

filename	purpose
input.nml	to read the model_mod namelist
namelist.hrldas	to read the NOAHLSM_OFLINE namelist
wrfinput	provides NOAH grid information
&model_nml:noah_netcdf_filename	the RESTART file containing the NOAH model state.

6.47 PBL_1D

6.47.1 Overview

The PBL_1D directory has been deprecated in favor of using the WRF/DART model interface. There is now support for WRF single column mode built into the standard model_mod in that directory.

If you are interested in more information on this configuration, please email us at dart@ucar.edu.

If you really want the files that used to be in this directory, check them out from the Kodiak release of DART.

6.48 pe2lyr

6.48.1 Overview

DART standard interfaces for a two-layer isentropic primitive equation model.

The 16 public interfaces are standardized for all DART compliant models. These interfaces allow DART to advance the model, get the model state and metadata describing this state, find state variables that are close to a given location, and do spatial interpolation for model state variables.

This model is a 2-layer, isentropic, primitive equation model on a sphere. TODO: add more detail here, including equations, etc.

Contact: Jeffrey.S.Whitaker@noaa.gov

6.48.2 Other modules used

```
types_mod
time_manager_mod
utilities_mod
random_seq_mod
threed_sphere/location_mod
```

6.48.3 Public interfaces

use model_mod, only :	get_model_size
	adv_1step
	get_state_meta_data
	model_interpolate
	get_model_time_step
	static_init_model
	end_model
	init_time
	init_conditions
	nc_write_model_atts
	nc_write_model_vars
	pert_model_state
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	ens_mean_for_model

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

model_size = get_model_size( )

```
integer :: get_model_size
```

Returns the size of the model as an integer. For this model the default grid size is 96 (lon) by 48 (lat) by 2 levels, and 3 variables (U, V, Z) at each grid location, for a total size of 27,648. There are alternative include files which, if included at compile time instead of the default file, defines a grid at twice and 4 times this resolution. They have corresponding truncation values of T63 and T127 (the default grid uses T31).

`model_size`	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

Advances the model for a single time step. The time associated with the initial model state is also input although it is not used for the computation.

`x`	State vector of length model_size.
`time`	Specifies time of the initial model state.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,             intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,   intent(out) ::  var_type
```

Returns metadata about a given element, indexed by `index_in`, in the model state vector. The `location` defines where the state variable is located.

For this model, the default grid is a global lat/lon grid, 96 (lon) by 48 (lat) by 2 levels. The variable types are U, V, and Z:

- 1 = TYPE_u
- 2 = TYPE_v
- 901 = TYPE_z

Grids at twice and 4 times the resolution can be compiled in instead by using one of the alternative header files (see `resolt31.h` (the default), `resolt63.h`, and `resolt127.h`).

`index_in`	Index of state vector element about which information is requested.
`location`	The location of state variable element.
var_type	The type of the state variable element.

call model_interpolate(x, location, itype, obs_val, istatus)

```
real(r8), dimension(:), intent(in)  :: x
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: itype
real(r8),               intent(out) :: obs_val
integer,                intent(out) :: istatus
```

Given a state vector, a location, and a model state variable type, interpolates the state variable field to that location and returns the value in obs_val. The istatus variable is always returned as 0 (OK).

`x`	A model state vector.
`location`	Location to which to interpolate.
`itype`	Type of state field to be interpolated.
`obs_val`	The interpolated value from the model.
`istatus`	Integer value returning 0 for successful, other values can be defined for various failures.

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

Returns the the time step of the model; the smallest increment in time that the model is capable of advancing the state in a given implementation. For this model the default value is 20 minutes (1200 seconds), but also comes with header files with times steps of 10 and 5 minutes (for higher grid resolution and truncation constants).

`var`	Smallest time step of model.

call static_init_model()

Used for runtime initialization of a model, for instance calculating storage requirements, initializing model parameters, etc. This is the first call made to a model by any DART compliant assimilation routines.

In this model, it allocates space for the grid, and initializes the grid locations, data values, and various parameters, including spherical harmonic weights.

call end_model()

A stub since the pe2lyr model does no cleanup.

call init_time(time)

```
type(time_type), intent(out) :: time
```

Returns the time at which the model will start if no input initial conditions are to be used. This model sets the time to 0.

time	Initial model time.

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

Returns default initial conditions for model; generally used for spinning up initial model states. This model sets the default state vector based on the initialized fields in the model. (TODO: which are what?)

x	Initial conditions for state vector.

ierr = nc_write_model_atts(ncFileID)

```
integer               :: nc_write_model_atts
integer, intent(in)  :: ncFileID
```

This routine writes the model-specific attributes to a netCDF file. This includes coordinate variables and any metadata, but NOT the model state vector. This model writes out the data as U, V, and Z arrays on a lat/lon/height grid, so the attributes are organized in the same way.

ncFileID	Integer file descriptor to previously-opened netCDF file.
ierr	Returns a 0 for successful completion.

ierr = nc_write_model_vars(ncFileID, statevec, copyindex, timeindex)

```
integer                                   :: nc_write_model_vars
integer,                  intent(in)  :: ncFileID
real(r8), dimension(:), intent(in)  :: statevec
integer,                  intent(in)  :: copyindex
integer,                  intent(in)  :: timeindex
```

This routine writes the model-specific state vector (data) to a netCDF file. This model writes out the data as U, V, and Z arrays on a lat/lon/height grid.

ncFileID	file descriptor to previously-opened netCDF file.
statevec	A model state vector.
copyindex	Integer index of copy to be written.
timeindex	The timestep counter for the given state.
ierr	Returns 0 for normal completion.

call pert_model_state(state, pert_state, interf_provided)

```
real(r8), dimension(:), intent(in)  :: state
real(r8), dimension(:), intent(out) :: pert_state
logical,                intent(out) :: interf_provided
```

Given a model state vector, perturbs this vector. Used to generate initial conditions for spinning up ensembles. This model has no code to generate these values, so it returns `interf_provided` as .false. and the default algorithms in filter are then used by the calling code.

`state`	State vector to be perturbed.
`pert_state`	Perturbed state vector
`interf_provided`	Returned false; interface is not implemented.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
real(r8),             intent(in)    :: maxdist
```

In distance computations any two locations closer than the given `maxdist` will be considered close by the `get_close_obs()` routine. Pass-through to the 3-D sphere locations module. See get_close_maxdist_init() for the documentation of this subroutine.

`gc`	The get_close_type which stores precomputed information about the locations to speed up searching
`maxdist`	Anything closer than this will be considered close.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(num)
```

Pass-through to the 3-D sphere locations module. See get_close_obs_init() for the documentation of this subroutine.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, num_close, close_ind [, dist])

```
type(get_close_type), intent(in)  :: gc
type(location_type),  intent(in)  :: base_obs_loc
integer,              intent(in)  :: base_obs_kind
```

(continues on next page)

(continued from previous page)

```
type(location_type),  intent(in)  :: obs(:)
integer,              intent(in)  :: obs_kind(:)
integer,              intent(out) :: num_close
integer,              intent(out) :: close_ind(:)
real(r8), optional,   intent(out) :: dist(:)
```

Given a location and kind, compute the distances to all other locations in the `obs` list. The return values are the number of items which are within maxdist of the base, the index numbers in the original obs list, and optionally the distances. The `gc` contains precomputed information to speed the computations.

Pass-through to the 3-D sphere locations module. See get_close_obs() for the documentation of this subroutine.

call ens_mean_for_model(ens_mean)

```
real(r8), dimension(:), intent(in) :: ens_mean
```

Stub only. Not needed by this model.

`ens_mean`	State vector containing the ensemble mean.

This model currently has no values settable by namelist.

6.48.4 Files

- The model source is in pe2lyr_mod.f90, and the spherical harmonic code is in spharmt_mod.f90. The various resolution settings are in resolt31.h, resolt63.h, and resolt127.h.

6.48.5 References

Zou, X., Barcilon, A., Navon, I.M., Whitaker, J., Cacuci, D.G.. 1993: An Adjoint Sensitivity Study of Blocking in a Two-Layer Isentropic Model. Monthly Weather Review: Vol. 121, No. 10, pp. 2833-2857.

6.48.6 Private components

N/A

6.49 POP

6.49.1 Overview

This document describes the DART interface to the Parallel Ocean Program (POP). It covers the *Development history* of the interface with two implementations of POP:

- the Los Alamos National Laboratory Parallel Ocean Program (LANL POP), and
- the Community Earth System Model Parallel Ocean Program 2 (CESM POP2; Smith et al. 2010[1]).

This document also provides *Detailed instructions for using DART and CESM POP2 on NCAR's supercomputer*, including information about the availability of restart files for *Creating an initial ensemble* of model states and *Observation sequence files* for assimilation.

6.49.2 Development History

When the DART interface to POP was originally developed circa 2009-2010, the interface worked with both the LANL POP and CESM POP2 implementations of POP.

LANL POP

In years subsequent to the initial development of the DART interface, the Computer, Computational, and Statistical Sciences Division at LANL transitioned from using POP as their primary ocean model to using the Model for Prediction Across Scales-Ocean (MPAS-Ocean). Thus it became difficult for staff in the Data Assimilation Research Section (DAReS) at NCAR to maintain access to the LANL POP source code. As a result, LANL POP has been tested using DART's Lanai framework but has not been tested using DART's Manhattan framework. If you intend to use LANL POP with DART Manhattan, contact DAReS staff for assistance by emailing dart@ucar.edu.

CESM POP2

The NCAR implementation of POP, CESM POP2, has been used extensively with DART throughout multiple generations of NCAR's supercomputer (Bluefire, Yellowstone & Cheyenne) and multiple iterations of NCAR's earth system model (CCSM4, CESM1 and CESM2). CESM POP2 is supported under DART's Manhattan framework.

For DART's CESM POP2 interface, the CESM Interactive Ensemble facility is used to manage the ensemble and the Flux Coupler is responsible for stopping POP2 at the times required to perform an assimilation. CESM runs continuously and all of the DART routines run at each assimilation time.

[1] Smith, R., and Coauthors, 2010: The Parallel Ocean Program (POP) Reference Manual Ocean Component of the Community Climate System Model (CCSM) and Community Earth System Model (CESM). National Center for Atmospheric Research, http://www.cesm.ucar.edu/ models/cesm1.0/pop2/doc/sci/POPRefManual.pdf.

6.49.3 Detailed instructions for using DART and CESM POP2 on NCAR's supercomputer

If you're using NCAR's supercomputer, you can run the setup scripts after making minor edits to set details that are specific to your project. The setup scripts create a CESM case in which POP is configured using a 1° horizontal grid, and uses the eddy-paremetrization of Gent and McWilliams (1990).[2] The CICE model is active and atmospheric forcing is provided by the CAM6 DART Reanalysis.

The filesystem attached to NCAR's supercomputer is known as the Globally Accessible Data Environment (GLADE). All filepaths on GLADE have the structure:

```
/glade/*
```

If you aren't using NCAR's supercomputer, take note of when the /glade/ filepath is present in the setup scripts, since this will indicate sections that you must alter in order to get the scripts to work on your supercomputer. Additionally, you'll need to generate your own initial condition and observation sequence files or you'll need to copy these files from GLADE. If you want to copy these files from GLADE and don't have access, contact DAReS staff by emailing dart@ucar.edu for assistance.

6.49.4 Summary

To use DART and CESM POP2 on NCAR's supercomputer, you will need to complete the following steps.

1. Configure the scripts for your specific experiment by editing DART_params.csh.
2. Stage your initial ensemble using copy_POP_JRA_restarts.py.
3. Run the appropriate DART setup script to create and build the CESM case.

If the DART setup script runs to completion, it will print instructions to the screen. Follow these instructions to submit your case.

6.49.5 Shell scripts

Since CESM requires many third-party modules in order to compile, it is often difficult to compile older versions of CESM because the older modules become unavailable. You should attempt to use the most recent setup scripts. The Discuss CESM bulletin board specifies which releases of CESM are supported.

The setup scripts are stored in:

```
DART/models/POP/shell_scripts
```

in subdirectories that correspond releases of CESM. For example:

```
DART/models/POP/shell_scripts/cesm2_1
```

contains scripts that should be used with CESM releases 2.1.0-2.1.3.

[2] Gent, P. R., and J. C. McWilliams, 1990: Isopycnal Mixing in Ocean Circulation Models. *Journal of Physical Oceanography*, **20**, 150–155, doi:10.1175/1520-0485(1990)020<0150:IMIOCM>2.0.CO;2.

copy_POP_JRA_restarts.py

This script stages an intial ensemble of POP2 restart files by copying files from a prior experiment run by *Who Kim*. Thanks Who!

These restart files can be used as an initial ensemble of model states. The files are kept in a directory on GLADE that is owned by the Climate and Global Dynamics (CGD) Ocean Section:

```
/glade/campaign/cgd/oce/people/whokim/csm/g210.G_JRA.v14.gx1v7.01
```

Unless you're already a member of the CGD Ocean Section, you must be granted access to this directory by CISL. Use the Service Desk to request permission. If you're unable to get permission, contact DAReS staff for assistance by emailing dart@ucar.edu.

Filepaths beginning with `/glade/campaign/*` can't be accessed from NCAR's supercomputer nodes. You must log on to NCAR's data visualization computer to copy files from `/glade/campaign/*`.

This python script was created by *Dan Amrhein*. Thanks Dan!

Script name	Description
`copy_POP_JRA_restarts.py`	This script copies restart files from the g210.G_JRA.v14.gx1v7.01 experiment that are saved in campaign storage. You must be granted access to the CGD Ocean Section campaign storage directory and be logged on to NCAR's data visualization computer in order to run this script. The assignment of the `stagedir` variable in this script should match the assignment of the `stagedir` variable in `DART_params.csh`.

In order to use this script, log in to NCAR's data visualization computer and use python to run the script. For example:

```
$ cd DART/models/POP/shell_scripts/cesm2_1
$ python copy_POP_JRA_restarts.py
```

DART_params.csh

This is the essential script you must edit to get your cases to build properly. While you need to configure this script, you don't need to run this script. It is run by the setup scripts.

Script name	Description
`DART_params.csh`	This script contains most, if not all, of the variables that you need to set in order to build and run cases. You must read this file carefully and configure the variables to match your needs. The assignment of the `stagedir` variable in this script should match the assignment of the `stagedir` variable in `copy_POP_JRA_restarts.py`.

Setup scripts

These are the primary scripts used to setup CESM cases in which data assimilation is enabled in POP2. The only variable that you might need to set in these scripts is the `extra_string` variable. It is appended to the end of the CESM case name. You can use it to differentiate experiments with the same configuration.

Script name	Description
`setup_CESM_perfect_model.csh`	This script creates a CESM case with a single model instance in order to run DART's `perfect_model_obs` program to collect observations from the model run.
`setup_CESM_hybrid_ensemble.csh`	This script creates a CESM case with multiple model instances in order to run DART's `filter` program to complete assimilation.

After configuring your experiment in `DART_params.csh`, you can setup a case by running these scripts. For example, to setup an assimilation experiment:

```
$ cd DART/models/POP/shell_scripts/cesm2_1
$ ./setup_CESM_hybrid_ensemble.csh
```

If the setup scripts run to completion, they will print instructions that you can follow to use CESM's case submit tool to begin a model integration.

CESM_DART_config.csh

This script is copied by the setup scripts into the CESM case directory. It configures CESM to run DART.

Script name	Description
`CESM_DART_config.csh`	This script is copied into the CESM case directory where it configures CESM to run DART.

Runtime scripts

These scripts are copied into the CESM case directory. They are called by CESM and contain the logic to run DART's `perfect_model_obs` or `filter` programs. You shouldn't need to run these scripts directly, unless they exit before completion and halt a CESM integration. In this case you may need to run the script directly to complete an assimilation in order to continue the integration.

Script name	Description
`perfect_model.csh`	This script runs `perfect_model_obs` to collect synthetic data in a single-instance CESM case.
`assimilate.csh`	This script runs `filter` to perform assimilation in a multi-instance CESM case.

6.49.6 Other files needed for assimilation

Creating an initial ensemble

Karspeck et al. (2013)[3] find that an ensemble of 1 January model states selected from a multi-decade free-running integration of POP2 can be used as an initial ensemble.

If you have access to CGD's Ocean Section directory on `/glade/campaign` you can use the *copy_POP_JRA_restarts.py* script to stage a collection of POP restart files from Who Kim's mulit-century `g210.G_JRA.v14.gx1v7.01` experiment to serve as an initial ensemble. This experiment uses the JRA-55 dataset for atmospheric forcing (Tsujino et al. 2018[4]).

Observation sequence files

When `setup_CESM_hybrid_ensemble.csh` is used to create an assimilation experiment, `DART_params.csh` configures the experiment to assimilate observation sequence files from the World Ocean Database 2013 (WOD13; Boyer et al. 2013[5]).

The WOD13 dataset comprises data from 2005-01-01 to 2016-12-31 and contains the following observation types:

FLOAT_SALINITY	FLOAT_TEMPERATURE
DRIFTER_SALINITY	DRIFTER_TEMPERATURE
GLIDER_SALINITY	GLIDER_TEMPERATURE
MOORING_SALINITY	MOORING_TEMPERATURE
BOTTLE_SALINITY	BOTTLE_TEMPERATURE
CTD_SALINITY	CTD_TEMPERATURE
XCTD_SALINITY	XCTD_TEMPERATURE
APB_SALINITY	APB_TEMPERATURE
XBT_TEMPERATURE	

The W0D13 observations have already been converted into DART's observation sequence file format by *Fred Castruccio*. Thanks Fred! The files are stored in the following directory on GLADE:

```
/glade/p/cisl/dares/Observations/WOD13
```

The subdirectories are formatted in `YYYYMM` order.

Observation sequence files converted from the World Ocean Database 2009 (WOD09; Johnson et al. 2009[6]), which comprises data from 1960-01-01 to 2008-12-31, are also stored in the following directory on GLADE:

```
/glade/p/cisl/dares/Observations/WOD09
```

These observation sequence files can be assimilated by changing the `BASEOBSDIR` variable in `DART_params.csh`.

DART extracts the following variables from the POP2 restart files and adjusts them to be consistent with the observations: `SALT_CUR`, `TEMP_CUR`, `UVEL_CUR`, `VVEL_CUR`, and `PSURF_CUR`.

[3] Karspeck, A., Yeager, S., Danabasoglu, G., Hoar, T. J., Collins, N. S., Raeder, K. D., Anderson, J. L, Tribbia, J. 2013: An ensemble adjustment Kalman filter for the CCSM4 ocean component. *Journal of Climate*, **26**, 7392-7413, doi:10.1175/JCLI-D-12-00402.1.

[4] Tsujino, H., Urakawa, S., Nakano, H., Small, R. J., Kim, W. M., Yeager, S. G., … Yamazaki, D., 2018: JRA-55 based surface dataset for driving ocean-sea-ice models (JRA55-do). *Ocean Modelling*, **130**, 79-139, doi:10.1016/j.ocemod.2018.07.002.

[5] Boyer, T.P., J. I. Antonov, O. K. Baranova, C. Coleman, H. E. Garcia, A. Grodsky, D. R. Johnson, R. A. Locarnini, A. V. Mishonov, T.D. O'Brien, C.R. Paver, J.R. Reagan, D. Seidov, I. V. Smolyar, and M. M. Zweng, 2013: World Ocean Database 2013, NOAA Atlas NESDIS 72, S. Levitus, Ed., A. Mishonov, Technical Ed.; Silver Spring, MD, 209 pp., doi:10.7289/V5NZ85MT.

[6] Johnson, D.R., T.P. Boyer, H.E. Garcia, R.A. Locarnini, O.K. Baranova, and M.M. Zweng, 2009. World Ocean Database 2009 Documentation. Edited by Sydney Levitus. NODC Internal Report 20, NOAA Printing Office, Silver Spring, MD, 175 pp., http://www.nodc.noaa.gov/OC5/WOD09/pr_wod09.html.

Data atmosphere streams files

The setup scripts configure the CESM case with atmospheric forcing from the CAM6 DART Reanalysis. The coupler history files from this reanalysis are referenced in `user_datm.streams*template` files. These `user_datm.streams*template` files are contained in the same directory as the setup scripts and are configured and copied into the CESM case directory by the setup scripts.

6.49.7 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand, `&`, and terminate with a slash, `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

The variables and their default values are listed here:

```
&model_nml
   assimilation_period_days      = -1
   assimilation_period_seconds   = -1
   model_perturbation_amplitude = 0.2
   binary_grid_file_format       = 'big_endian'
   debug                         = 0,
   model_state_variables         = 'SALT_CUR ', 'QTY_SALINITY              ', 'UPDATE',
                                   'TEMP_CUR ', 'QTY_POTENTIAL_TEMPERATURE', 'UPDATE',
                                   'UVEL_CUR ', 'QTY_U_CURRENT_COMPONENT  ', 'UPDATE',
                                   'VVEL_CUR ', 'QTY_V_CURRENT_COMPONENT  ', 'UPDATE',
                                   'PSURF_CUR', 'QTY_SEA_SURFACE_PRESSURE ', 'UPDATE'
/
```

This namelist provides control over the assimilation period for the model. All observations within (+/-) half of the assimilation period are assimilated. The assimilation period is the minimum amount of time the model can be advanced, and checks are performed to ensure that the assimilation window is a multiple of the ocean model dynamical timestep.

Item	Type	Description
assimilation_period_days	integer	The number of days to advance the model for each assimilation. If both assimilation_period_days and assimilation_period_seconds are 0; the value of the POP namelist variables restart_freq and restart_freq_opt are used to determine the assimilation period. *WARNING:* in the CESM framework, the restart_freq is set to a value that is not useful so DART defaults to 1 day - even if you are using POP in the LANL framework.
assimilation_period_seconds	integer	In addition to assimilation_period_days, the number of seconds to advance the model for each assimilation. Make sure you read the description of assimilation_period_days.
model_perturbation_amplitude	real(r8)	Reserved for future use.
binary_grid_file_format	character(len=32)	The POP grid files are in a binary format. Valid values are native, big_endian, or little_endian. Modern versions of Fortran allow you to specify the endianness of the file you wish to read when they are opened as opposed to needing to set a compiler switch or environment variable.
debug	integer	The switch to specify the run-time verbosity. • 0 is as quiet as it gets. • > 1 provides more run-time messages. • > 5 provides ALL run-time messages. All values above 0 will also write a netCDF file of the grid information and perform a grid interpolation test.
model_state_variables	character(:,3)	Strings that associate POP variables with a DART quantity and whether or not to write the updated values to the restart files. These variables will be read from the POP restart file and modified by the assimilation. Some (perhaps all) will be used by the forward observation operators. If the 3rd column is 'UPDATE', the output files will have the modified (assimilated,posterior) values. If the 3rd column is 'NO_COPY_BACK', that variable will not be written to the restart files. **The DART diagnostic files will always have the**

6.49.8 References

6.50 ROMS

There are several DART users who have working DART interface code to the Regional Ocean Modeling System (ROMS), as the model is a community ocean model funded by the Office of Naval Research. Please visit MyRoms for more information on the model.

The lead developers are at Rutgers and UCLA, but the list of associate developers is extensive. Please read ROMS developers for more information.

If you are interested in running DART with this model please contact the DART group at dart@ucar.edu for more information. We are currently working with collaborators to optimize the model_mod interface and associated scripting to run data assimilation experiments with this model. We may be able to put you in contact with the right people to get a copy of the code.

6.50.1 Overview

This document describes the relationship between ROMS and DART and provides an overview of how to perform ensemble data assimilation with ROMS to provide ocean states that are consistent with the information provided by various ocean observations.

Running ROMS is complicated. It is **strongly** recommended that you become very familiar with running ROMS before you attempt a ROMS-DART assimilation experiment. Running DART is complicated. It is **strongly** recommended that you become very familiar with running DART before you attempt a ROMS-DART assimilation experiment. Running ROMS-DART takes expertise in both areas.

We recommend working through the DART tutorial to learn the concepts of ensemble data assimilation and the capabilities of DART.

The ROMS code is not distributed with DART, it can be obtained from the ROMS website. There you will also find instructions on how to compile and run ROMS. DART can use the ‘verification observations’ from ROMS (basically the estimate of the observation at the location and time computed as the model advances) so it would be worthwhile to become familiar with that capability of ROMS.

DART calls these ‘precomputed forward operators’. DART can also use observations from the World Ocean Database - WOD. The conversion from the WOD formats to the DART observation sequence format is accomplished by the converters in the `DARTHOME/observations/obs_converters/WOD` directory.

The DART forward operators require interpolation from the ROMS terrain-following and horizontally curvilinear orthogonal coordinates to the observation location. Please contact us for more information about this interpolation.

6.50.2 A Note About Filenames

During the course of an experiment, many files are created. To make them unique, the *ocean_time* is converted from “seconds since 1900-01-01 00:00:00” to the equivalent number of DAYS. An *integer* number of days. The intent is to tag the filename to reflect the valid time of the model state. This could be used as the DSTART for the next cycle, so it makes sense to me. The confusion comes when applied to the observation files.

The input observation files for the ROMS 4DVAR system typically have a DSTART that designates the start of the forecast cycle and the file must contain observation from DSTART to the end of the forecast. Makes sense.

The model runs to the end of the forecast, harvesting the verification observations along the way. So then DART converts all those verification observations and tags that file … with the same time tag as all the other output files … which reflects the *ocean_time* (converted to days). The input observation file to ROMS will have a different DSTART time in the filename than the corresponding verification files. Ugh. You are free to come up with a better plan.

These are just examples. . . after all; hopefully good examples.

6.50.3 Procedure

The procedure to perform an assimilation experiment is outlined in the following steps:

1. Compile ROMS (as per the ROMS instructions).
2. Compile all the DART executables (in the normal fashion).
3. Stage a directory with all the files required to advance an ensemble of ROMS models and DART.
4. Modify the run-time controls in `ocean.in`, `s4dvar.in` and `input.nml`. Since ROMS has a *Bin/subsitute* command, it is used to replace temporary placeholders with actual values at various parts during the process.
5. Advance all the instances of ROMS; each one will produce a restart file and a verification observation file.
6. Convert all the verification observation files into a single DART observation sequence file with the `convert_roms_obs.f90` program in `DARTHOME/observations/obs_converters/ROMS/`.
7. Run filter to assimilate the data (DART will read and update the ROMS files directly - no conversion is necessary.)
8. Update the control files for ROMS in preparation for the next model advance.

6.50.4 Shell scripts

The `shell_scripts` directory has several scripts that are intended to provide examples. These scripts **WILL** need to be modified to work on your system and are heavily internally commented. It will be necessary to read through and understand the scripts. As mentioned before, the ROMS *Bin/subsitute* command is used to replace temporary placeholders with actual values at various parts during the process.

Script	Description
ensemble.sh	Was written by Hernan Arango to run an ensemble of ROMS models. It is an appropriate example of what is required from the ROMS perspective. It does no data assimilation.
stage_experiment.csh	prepares a directory for an assimilation experiment. The idea is basically that everything you need should be assembled by this script and that this should only be run ONCE per experiment. After everything is staged in the experiment directory, another script can be run to advance the model and perform the assimilation. *stage_experiment.csh* will also modify some of the template scripts and copy working versions into the experiment directory. This script may be run interactively, i.e. from the UNIX command line.
submit_multiple_cycles_lsf	is an executable script that submits a series of dependent jobs to an LSF queuing system. Each job runs *cycle.csh* in the experiment directory and only runs if the previous dependent job completes successfully.
cycle.csh.template	is a non-executable template that is modified by *stage_experiment.csh* and results in an exectuable *cycle.csh* in the experiment directory. *cycle.csh* is designed to be run as a batch job and advances the ROMS model states one-by-one for the desired forecast length. The assimilation is performed and the control information for the next ROMS forecast is updated. Each model execution and *filter* use the same set of MPI tasks.
submit_multiple_jobs_slurm.csh	is an executable script that submits a series of dependent jobs to an LSF queuing system. It is possible to submit **many** jobs the queue, but the jobs run one-at-a-time. Every assimilation cycle is divided into two scripts to be able to efficiently set the resources for each phase. *advance_ensemble.csh* is a job array that advances each ROMS instance in separate jobs. When the entire job array finishes - and only if they all finish correctly - will the next job start to run. *run_filter.csh* performs the assimilation and prepares the experiment directory for another assimilation cycle. *submit_multiple_jobs_slurm.csh* may be run from the command line in the experiment directory. Multiple assimilation cycles can be specified, so it is possible to put **many** jobs in the queue.
advance_ensemble.csh.template	is a non-executable template that is modified by *stage_experiment.csh* and results in an exectuable *advance_ensemble.csh* in the experiment directory. *advance_ensemble.csh* is designed to submit an job array to the queueing system (PBS,SLURM, or LSF) to advance the ensemble members in separate jobs.
run_filter.csh.template	is a non-executable template that is modified by *stage_experiment.csh* and results in an exectuable *run_filter.csh* in the experiment directory. *run_filter.csh* is very similar to *cycle.csh* but does not advance the ROMS model instances.

The variables from ROMS that are copied into the DART state vector are controlled by the *input.nml model_nml* namelist. See below for the documentation on the &model_nml entries. The state vector should include all variables needed to apply the forward observation operators as well as the prognostic variables important to restart ROMS.

The example *input.nml model_nml* demonstrates how to construct the DART state vector. The following table explains in detail each entry for the *variables* namelist item:

Variable name	This is the ROMS variable name as it appears in the ROMS netCDF file.
DART QUANTITY	This is the character string of the corresponding DART QUANTITY. The complete list of possible DART QUANTITY values is available in the `obs_def_mod` that is built by `preprocess`.
minimum	If the variable is to be updated in the ROMS restart file, this specifies the minimum value. If set to 'NA', there is no minimum value.
maximum	If the variable is to be updated in the ROMS restart file, this specifies the maximum value. If set to 'NA', there is no maximum value.
update	The updated variable may or may not be written to the ROMS restart file. *'UPDATE'* means the variable in the restart file is updated. This is case-insensitive. *'NO_COPY_BACK'* (or anything else) means the variable in the restart file remains unchanged.

6.50.5 Namelist

This namelist is read from the file *input.nml*. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist. The default namelist is presented below, a more realistic namelist is presented at the end of this section.

```
&model_nml
  roms_filename                = 'roms_input.nc'
  assimilation_period_days     = 1
  assimilation_period_seconds = 0
  vert_localization_coord      = 3
  debug                        = 0
  variables                    = ''
/
```

Item	Type	Description
roms_filename	character(len=256)	This is the name of the file used to provide information about the ROMS variable dimensions, etc.
assi milation_period_days, assimi lation_period_seconds	integer	Combined, these specify the width of the assimilation window. The current model time is used as the center time of the assimilation window. All observations in the assimilation window are assimilated. BEWARE: if you put observations that occur before the beginning of the assimilation_period, DART will error out because it cannot move the model 'back in time' to process these observations.
variables	character(:, 5)	A 2D array of strings, 5 per ROMS variable to be added to the dart state vector. 1. ROMS field name - must match netCDF variable name exactly 2. DART QUANTITY - must match a valid DART QTY_xxx exactly 3. minimum physical value - if none, use 'NA' 4. maximum physical value - if none, use 'NA' 5. case-insensitive string describing whether to copy the updated variable into the ROMS restart file ('UPDATE') or not (any other value). There is generally no point copying diagnostic variables into the restart file. Some diagnostic variables may be useful for computing forward operators, however.
ve rt_localization_coord	integer	Vertical coordinate for vertical localization. • 1 = model level • 2 = pressure (in pascals) • 3 = height (in meters) • 4 = scale height (unitless) Currently, only 3 (height) is supported for ROMS.

A more realistic ROMS namelist is presented here, along with one of the more unusual settings that is generally necessary when running ROMS. The *use_precomputed_FOs_these_obs_types* variable needs to list the observation types that are present in the ROMS verification observation file.

```
&model_nml
  roms_filename                  = 'roms_input.nc'
  assimilation_period_days       = 1
  assimilation_period_seconds    = 0
  vert_localization_coord        = 3
  debug                          = 1
  variables = 'temp',    'QTY_TEMPERATURE',          'NA', 'NA', 'update',
              'salt',    'QTY_SALINITY',            '0.0', 'NA', 'update',
              'u',       'QTY_U_CURRENT_COMPONENT',  'NA', 'NA', 'update',
              'v',       'QTY_V_CURRENT_COMPONENT',  'NA', 'NA', 'update',
              'zeta',    'QTY_SEA_SURFACE_HEIGHT'    'NA', 'NA', 'update'
/
&obs_kind_nml
  evaluate_these_obs_types = ''
  assimilate_these_obs_types =          'SATELLITE_SSH',
                                        'SATELLITE_SSS',
                                        'XBT_TEMPERATURE',
                                        'CTD_TEMPERATURE',
                                        'CTD_SALINITY',
                                        'ARGO_TEMPERATURE',
                                        'ARGO_SALINITY',
                                        'GLIDER_TEMPERATURE',
                                        'GLIDER_SALINITY',
                                        'SATELLITE_BLENDED_SST',
                                        'SATELLITE_MICROWAVE_SST',
                                        'SATELLITE_INFRARED_SST'
  use_precomputed_FOs_these_obs_types = 'SATELLITE_SSH',
                                        'SATELLITE_SSS',
                                        'XBT_TEMPERATURE',
                                        'CTD_TEMPERATURE',
                                        'CTD_SALINITY',
                                        'ARGO_TEMPERATURE',
                                        'ARGO_SALINITY',
                                        'GLIDER_TEMPERATURE',
                                        'GLIDER_SALINITY',
                                        'SATELLITE_BLENDED_SST',
                                        'SATELLITE_MICROWAVE_SST',
                                        'SATELLITE_INFRARED_SST'
/
```

6.51 ROSE

6.51.1 Overview

The rose model is an atmospheric model for the Mesosphere Lower-Thermosphere (MLT). The DART interface was developed by Tomoko Matsuo (now at CU-Boulder).

The source code for rose is not distributed with DART, thus the DART/models/rose/work/workshop_setup.csh script is SUPPOSED to fail without the rose code.

The rose model is a research model that is still being developed. The DART components here are simply to help the rose developers with the DART framework.

As of Mon Mar 22 17:23:20 MDT 2010 the rose project has been substantially streamlined. There is no need for the trans_time and build_nml routines. dart_to_model has assumed those responsibilities.

6.52 Simple advection

6.52.1 Overview

This simple advection model simulates a wind field using Burger's Equation with an upstream semi-lagrangian differencing on a periodic one-dimensional domain. This diffusive numerical scheme is stable and forcing is provided by adding in random gaussian noise to each wind grid variable independently at each timestep. The domain mean value of the wind is relaxed to a constant fixed value set by the namelist parameter `mean_wind`. The random forcing magnitude is set by namelist parameter `wind_random_amp` and the damping of the mean wind is controlled by parameter `wind_damping_rate`. An Eulerian option with centered in space differencing is also provided and can be used by setting namelist parameter `lagrangian_for_wind` to `.false.` The Eulerian differencing is both numerically unstable and subject to shock formation. However, it can sometimes be made stable in assimilation mode (see recent work by Majda and collaborators).

The model state includes a single passive tracer that is advected by the wind field using semi-lagrangian upstream differencing. The state also includes a tracer source value at each gridpoint. At each time step, the source is added into the concentration at each gridpoint. There is also a constant global destruction of tracer that is controlled by the namelist parameter destruction_rate. The appropriate percentage of tracer is destroyed at each gridpoint at each timestep.

The model also includes an associated model for the tracer source rate. At each gridpoint, there is a value of the time mean source rate and a value of the phase offset for a diurnal component of the source rate. The diurnal source rate has an amplitude that is proportional to the source rate (this proportion is controlled by namelist parameter `source_diurnal_rel_amp`). At each grid point, the source is the sum of the source rate plus the appropriate diurnally varying component. The phase_offset at the gridpoint controls the diurnal phase. The namelist parameter `source_phase_noise` controls the amplitude of random gaussian noise that is added into the source phase at each time step. If `source_phase_noise` is zero then the phase offset is fixed. Finally, the time mean source rate is constant in time in the present model version. The time mean source rate controls the amplitude of the diurnal cycle of the tracer source.

For the simple advection model, DART advances the model, gets the model state and metadata describing this state, finds state variables that are close to a given location, and does spatial interpolation for model state variables.

The simple advection model has a `work/workshop_setup.csh` script that compiles and runs an example. This example is referenced in Section 25 of the DART_tutorial and is intended to provide insight into model/assimilation behavior. The example **may or may not** result in good (*or even decent!*) results!

6.52.2 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   num_grid_points        = 10,
   grid_spacing_meters    = 100000.0,
   time_step_days         = 0,
   time_step_seconds      = 3600,
   mean_wind              = 20.0,
   wind_random_amp        = 0.00027778,
   wind_damping_rate      = 0.0000027878,
   lagrangian_for_wind    = .true.,
   destruction_rate       = 0.000055556,
   source_random_amp_frac = 0.00001,
```

(continues on next page)

(continued from previous page)

```
   source_damping_rate     = 0.0000027878,
   source_diurnal_rel_amp = 0.05,
   source_phase_noise     = 0.0,
   output_state_vector  = .false.
/
```

Description of each namelist entry

Item	Type	Description
num_grid_points	integer	Number of grid points in model. State vector size is 5 times this number.
grid_spacing_meters	integer	Grid spacing in meters.
time_step_days	real(r8)	Number of days for dimensional timestep, mapped to delta_t.
time_step_seconds	real(r8)	Number of seconds for dimensional timestep, mapped to delta_t.
mean_wind	real(r8)	Base wind velocity (expected value over time) in meters/second.
wind_random_amp	real(r8)	Random walk amplitude for wind in meters/second^2.
wind_damping_rate	real(r8)	Rate of damping towards mean wind value in fraction/second.
lagrangian_for_wind	logical	Can use Lagrangian (stable) or Eulerian (unstable) scheme for wind.
destruction_rate	real(r8)	Tracer destruction rate in fraction/second.
source_random_amp_frac	real(r8)	Random walk amplitude for source as a fraction of mean source (per second)^2.
source_damping_rate	real(r8)	Damping towards mean source rate in fraction/second.
source_diurnal_rel_amp	real(r8)	Relative amplitude of diurnal cycle of source (dimensionless).
source_phase_noise	real(r8)	Amplitude of gaussian noise to be added to source phase offset (per second).
output_state_vector	logical	Controls the output to netCDF files. If .true., output the raw dart state vector. If .false. output the prognostic version (gridded data) for easier plotting (recommended).

6.53 SQG

6.53.1 Overview

This is a uniform PV two-surface QG+1 spectral model contributed by Rahul Majahan.

The underlying model is described in: Hakim, Gregory J., 2000: Role of Nonmodal Growth and Nonlinearity in Cyclogenesis Initial-Value Problems. J. Atmos. Sci., 57, 2951-2967. doi: 10.1175/1520-0469(2000)057<2951:RONGAN>2.0.CO;2

6.53.2 Other modules used

```
types_mod
time_manager_mod
threed_sphere/location_mod
utilities_mod
```

6.53.3 Public interfaces

use model_mod, only :	get_model_size
	adv_1step
	get_state_meta_data
	model_interpolate
	get_model_time_step
	static_init_model
	end_model
	init_time
	init_conditions
	nc_write_model_atts
	nc_write_model_vars
	pert_model_state
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	ens_mean_for_model

Optional namelist interface `&model_nml` may be read from file `input.nml`.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

model_size = get_model_size( )

```
integer :: get_model_size
```

Returns the length of the model state vector.

model_size	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

Advances the model for a single time step. The time associated with the initial model state is also input although it is not used for the computation.

x	State vector of length model_size.
time	Specifies time of the initial model state.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,              intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,    intent(out) ::  var_type
```

Returns metadata about a given element, indexed by index_in, in the model state vector. The location defines where the state variable is located.

index_in	Index of state vector element about which information is requested.
location	The location of state variable element.
var_type	Returns the type (always 1) of the indexed state variable as an optional argument.

call model_interpolate(x, location, itype, obs_val, istatus)

```
real(r8), dimension(:), intent(in)  :: x
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: itype
real(r8),               intent(out) :: obs_val
integer,                intent(out) :: istatus
```

Given model state, returns the value interpolated to a given location.

x	A model state vector.
location	Location to which to interpolate.
itype	Not used.
obs_val	The interpolated value from the model.
istatus	Quality control information, always returned 0.

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

Returns the time step (forecast length) of the model;

var	Smallest time step of model.

call static_init_model()

Used for runtime initialization of model; reads namelist, initializes model parameters, etc. This is the first call made to the model by any DART-compliant assimilation routine.

call end_model()

A stub.

call init_time(time)

```
type(time_type), intent(out) :: time
```

Returns the time at which the model will start if no input initial conditions are to be used. This is used to spin-up the model from rest.

time	Initial model time.

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

Returns default initial conditions for the model; generally used for spinning up initial model states.

`x`	Initial conditions for state vector.

ierr = nc_write_model_atts(ncFileID)

```
integer             :: nc_write_model_atts
integer, intent(in) :: ncFileID
```

Function to write model specific attributes to a netCDF file. At present, DART is using the NetCDF format to output diagnostic information. This is not a requirement, and models could choose to provide output in other formats. This function writes the metadata associated with the model to a NetCDF file opened to a file identified by ncFileID.

`ncFileID`	Integer file descriptor to previously-opened netCDF file.
`ierr`	Returns a 0 for successful completion.

ierr = nc_write_model_vars(ncFileID, statevec, copyindex, timeindex)

```
integer                             :: nc_write_model_vars
integer,                intent(in)  :: ncFileID
real(r8), dimension(:), intent(in)  :: statevec
integer,                intent(in)  :: copyindex
integer,                intent(in)  :: timeindex
```

Writes a copy of the state variables to a netCDF file. Multiple copies of the state for a given time are supported, allowing, for instance, a single file to include multiple ensemble estimates of the state.

`ncFileID`	file descriptor to previously-opened netCDF file.
`statevec`	A model state vector.
`copyindex`	Integer index of copy to be written.
`timeindex`	The timestep counter for the given state.
`ierr`	Returns 0 for normal completion.

call pert_model_state(state, pert_state, interf_provided)

```
real(r8), dimension(:), intent(in)  :: state
real(r8), dimension(:), intent(out) :: pert_state
logical,                intent(out) :: interf_provided
```

Given a model state, produces a perturbed model state.

state	State vector to be perturbed.
pert_state	Perturbed state vector: NOT returned.
interf_provided	Returned false; interface is not implemented.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
real(r8),             intent(in)    :: maxdist
```

Pass-through to the 3D Sphere locations module. See get_close_maxdist_init() for the documentation of this subroutine.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(num)
```

Pass-through to the 3D Sphere locations module. See get_close_obs_init() for the documentation of this subroutine.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, num_close, close_ind [, dist])

```
type(get_close_type), intent(in)  :: gc
type(location_type),  intent(in)  :: base_obs_loc
integer,              intent(in)  :: base_obs_kind
type(location_type),  intent(in)  :: obs(:)
integer,              intent(in)  :: obs_kind(:)
integer,              intent(out) :: num_close
integer,              intent(out) :: close_ind(:)
real(r8), optional,   intent(out) :: dist(:)
```

Pass-through to the 3D Sphere locations module. See get_close_obs() for the documentation of this subroutine.

call ens_mean_for_model(ens_mean)

```
real(r8), dimension(:), intent(in) :: ens_mean
```

A NULL INTERFACE in this model.

ens_mean	State vector containing the ensemble mean.

6.53.4 Namelist

We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input.

```
&model_nml
  output_state_vector = .false.
  channel_center = 45.0
  channel_width = 40.0
  assimilation_period_days = 0
  assimilation_period_seconds = 21600
  debug = .false.
/
```

This namelist is read in a file called `input.nml`

Contents	Type	Description
output_state_vector	logical	If .true. write state vector as a 1D array to the diagnostic output file. If .false. break state vector up into fields before writing to the outputfile.
channel_center	real(r8)	Channel center
channel_width	real(r8)	Channel width
assimilation_period_days	integer	Number of days for timestep
assimilation_period_seconds	integer	Number of seconds for timestep
debug	logical	Set to .true. for more output

6.53.5 Files

filename	purpose
input.nml	to read the model_mod namelist
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.53.6 References

The underlying model is described in:

Hakim, Gregory J., 2000: Role of Nonmodal Growth and Nonlinearity in Cyclogenesis Initial-Value Problems. J. Atmos. Sci., 57, 2951-2967. doi: 10.1175/1520-0469(2000)057<2951:RONGAN>2.0.CO;2

6.53.7 Private components

N/A

6.54 TIEGCM

6.54.1 Overview

This is the DART interface to the Thermosphere Ionosphere Electrodynamic General Circulation Model (TIEGCM), which is a community model developed at the NCAR High Altitude Observatory. TIEGCM is widely used by the space physics and aeronomy community and is one of the most well-validated models of the Earth's upper atmosphere. DART/TIEGCM has been used to assimilate neutral mass density retrieved from satellite-borne accelerometers and electon density obtained from ground-based and space-based GNSS signals. Unlike other ionospheric data assimilation applications, this approach allows simultaneous assimilation of thermospheric and ionospheric parameters by taking advantage of the coupling of plasma and neutral constituents described in TIEGCM. DART/TIEGCM's demonstrated capability to infer under-observed thermospheric parameters from abundant electron density observations has important implications for the future of upper atmosphere research.

DART is designed so that the TIEGCM source code can be used with no modifications, as DART runs TIEGCM as a completely separate executable. The TIEGCM source code and restart files are **not** included in DART, so you must obtain them from the NCAR High Altitude Observatory (download website). It is **strongly** recommended that you become familiar with running TIEGCM **before** you try to run DART/TIEGCM (See the TIEGCM User's Guide). Some assumptions are made about the mannner in which TIEGCM is run: (1) There can only be 1 each of the TIEGCM primary (restart) and secondary NetCDF history files. The TIEGCM primary history files contain the prognostic variables necessary to restart the model, while the secondary history files contain diagnostic variables; (2) The last timestep in the restart file is the only timestep which is converted to a DART state vector, and only the last timestep in the TIEGCM primary file is ever modified by DART. The TIEGCM variables to be included in a DART state vector, and possibly updated by the assimilation, are specified in the DART namelist. (Some of the TIEGCM variables used to compute observation priors need not to be updated.) It is required to associate the TIEGCM variable name with a 'generic' DART counterpart (e.g., `NE` is `QTY_ELECTRON_DENSITY`). The composition of the DART state vector and which variables get updated in the TIEGCM primary file are under complete user control.

In the course of a filtering experiment, it is necessary to make a short forecast with TIEGCM. DART writes out an ancillary file with the information necessary to advance TIEGCM to the required time. The DART script `advance_model.csh` reads this information and modifies the TIEGCM namelist `tiegcm.nml` such that TIEGCM runs upto the requested time when DART assimilates the next set of observations. The run scripts `run_filter.csh` and `run_perfect_model_obs.csh` are configured to run under the LSF queueing system. The scripting examples exploit an 'embarassingly-simple' parallel paradigm in that each TIEGCM instance is a single-threaded executable and all ensemble members may run simultaneously. To use these run scripts, the TIECGM executable needs to be compiled with no MPI option. As such, there is an advantage to matching the ensemble size to the number of tasks. Requesting more tasks than the number of ensemble members may speed up the DART portion of an assimilation (i.e., `filter`) but will not make the model advance faster. The `filter` may be compiled with MPI and can exploit all available tasks.

6.54.2 Quickstart guide to running

It is important to understand basic DART nomenclature and mechanisms. Please take the time to read and run the DART tutorial.

Both `run_filter.csh` and `run_perfect_model_obs.csh` are heavily internally commented. Please read and understand the scripts. The overall process is to

1. Specify resources (wall-clock time, number of nodes, tasks that sort of thing).
2. Set shell variables to identify the location of the DART exectuables, the TIEGCM executables, initial ensemble, etc.
3. Establish a temporary working directory for the experiment.
4. Populate that directory with the initial ensemble and required namelists.
5. Convert each TIEGCM ensemble member to a DART initial conditions file.
6. Run either `filter` or `run_perfect_model_obs.csh`.
7. `perfect_model_obs` will
8. Check for any desired observations at the current time of the model state and create the synthetic observations for all observation times in the specified assimilation window. If the model needs to be advanced, it then
9. creates a unique run-time directory for the model advance,
10. copies the required information into that directory,
11. conveys the desired forecast stopping time to TIEGCM via the `tiegcm.nml` and
12. runs a single executable of TIEGCM.
13. Steps 1-5 are repeated until the input DART observation sequence file has been exhausted.
14. `filter` will
15. Check for any desired observations at the current time of the model state and assimilates all the observations in the specified assimilation window. If the model needs to be advanced, it then
16. creates a set of run-time directories, one for each task. A single task may be responsible for advancing more than one TIEGCM instance. If so, each instance is done serially, one after another. See the documentation for *Filter async modes*.
17. Copy the required information into that directory.
18. Update the TIEGCM restart file with the most current DART-modified state and convey the desired forecast stopping time to TIEGCM via the unique `tiegcm.nml` for this ensemble member.
19. Runs a single executable of TIEGCM.
20. Steps 1-5 are repeated until the input DART observation sequence file

What to check when things go wrong

The scripts are designed to send email to the user that contains the run-time output from the script. Check that first. If that does not provide the information needed, go to the run directory (i.e. CENTRALDIR) and check the `dart_log.out`. It usually provides the same information as the email, but sometimes it can help. If that does not help, go to any of the CENTRALDIR/*advance_tempnnnn* directories and read the *log_advance.nnnn.txt* file.

6.54.3 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   output_state_vector         = .false.
   tiegcm_restart_file_name    = 'tiegcm_restart_p.nc'
   tiegcm_secondary_file_name  = 'tiegcm_s.nc'
   tiegcm_namelist_file_name   = 'tiegcm.nml'
   assimilation_period_seconds = 3600
   estimate_f10_7              = .false.
   debug                       = 1
   variables = 'NE',    'QTY_ELECTRON_DENSITY',          '1000.0',  'NA',
↪'restart',     'UPDATE'
               'OP',    'QTY_DENSITY_ION_OP',            'NA',      'NA',
↪'restart',     'UPDATE',
               'TI',    'QTY_TEMPERATURE_ION',           'NA',      'NA',
↪'restart',     'UPDATE',
               'TE',    'QTY_TEMPERATURE_ELECTRON',      'NA',      'NA',
↪'restart',     'UPDATE',
               'OP_NM', 'QTY_DENSITY_ION_OP',            'NA',      'NA',
↪'restart',     'UPDATE',
               'O1',    'QTY_ATOMIC_OXYGEN_MIXING_RATIO','0.00001', '0.99999',
↪'secondary',   'NO_COPY_BACK',
               'O2',    'QTY_MOLEC_OXYGEN_MIXING_RATIO', '0.00001', '0.99999',
↪'secondary',   'NO_COPY_BACK',
               'TN',    'QTY_TEMPERATURE',               '0.0',     '6000.0',
↪'secondary',   'NO_COPY_BACK',
               'ZG',    'QTY_GEOMETRIC_HEIGHT',          'NA',      'NA',
↪'secondary',   'NO_COPY_BACK',
               'VTEC',  'QTY_VERTICAL_TEC',              'NA',      'NA',
↪'calculate',   'NO_COPY_BACK'
   /
```

Item	Type	Description
output_state_vector	logical	If .true. write state vector as a 1D array to the DART diagnostic output files. If .false. break state vector up into variables before writing to the output files.
tiegcm_restart_file_name	character(len=256)	The TIEGCM restart file name.
tiegcm_secondary_file_name	character(len=256)	The TIEGCM secondary file name.
tiegcm_namelist_file_name	character(len=256)	The TIEGCM namelist file name.
assimilation_period_seconds	integer	This specifies the width of the assimilation window. The current model time is used as the center time of the assimilation window. All observations in the assimilation window are assimilated. BEWARE: if you put observations that occur before the beginning of the assimilation_period, DART will error out because it cannot move the model 'back in time' to process these observations. `assimilation_period_seconds` must be an integer number of TIEGCM dynamical timesteps (as specified by tiegcm.nml:STEP) AND be able to be expressed by tiegcm.nml:STOP. Since STOP has three components: day-of-year, hour, and minute, the `assimilation_period_seconds` must be an integer number of minutes.
estimate_f10_7	logical	Switch to specify that the f10.7 index should be estimated by augmenting the DART state vector with a scalar. The location of the f10.7 index is taken to be longitude of local noon and latitude zero. WARNING: this is provided with no guarantees. Please read the comments in `model_mod.f90` and act accordingly.
debug	integer	Set to 0 (zero) for minimal output. Successively larger values generate successively more output.
variables	character(:,6)	Strings that identify the TIEGCM variables, their DART kind, the min & max values, what file to read from, and whether or not the file should be updated after the assimilation. The DART kind must be one found in the `DART/assimilation_code/mo dules/observations/obs_kind_mod.f90` AFTER it gets built by `preprocess`. Most of the upper atmosphere observation kinds are specified by `DART/observations/`

6.54.4 Other modules used

```
adaptive_inflate_mod.f90
assim_model_mod.f90
assim_tools_mod.f90
types_mod.f90
cov_cutoff_mod.f90
ensemble_manager_mod.f90
filter.f90
location/threed_sphere/location_mod.f90
[null_,]mpi_utilities_mod.f90
obs_def_mod.f90
obs_kind_mod.f90
obs_model_mod.f90
obs_sequence_mod.f90
random_seq_mod.f90
reg_factor_mod.f90
smoother_mod.f90
sort_mod.f90
time_manager_mod.f90
utilities_mod.f90
```

6.54.5 Public interfaces - required

use model_mod, only :	get_model_size
	adv_1step
	get_state_meta_data
	model_interpolate
	get_model_time_step
	static_init_model
	end_model
	init_time
	init_conditions
	nc_write_model_atts
	nc_write_model_vars
	pert_model_state
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	ens_mean_for_model

6.54.6 Public interfaces - optional

use model_mod, only :	tiegcm_to_dart_vector
	dart_vector_to_tiegcm
	get_f107_value
	test_interpolate

A namelist interface `&model_nml` is defined by the module, and is read from file `input.nml`.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

model_size = get_model_size( )

```
integer :: get_model_size
```

Returns the length of the model state vector. Required.

`model_size`	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

Since TIEGCM is not called as a subroutine, this is a NULL interface. TIEGCM is advanced as a separate executable - i.e. `async == 2`. *adv_1step* only gets called if `async == 0`. The subroutine must still exist, but contains no code and will not be called. An error message is issued if an unsupported value of `filter`, `perfect_model_obs`:`async` is used.

call get_state_meta_data (index_in, location, [, var_kind])

```
integer,             intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,   intent(out) ::  var_kind
```

Given an integer index into the state vector structure, returns the associated location. A second intent(out) optional argument returns the generic kind of this item, e.g. QTY_MOLEC_OXYGEN_MIXING_RATIO, QTY_ELECTRON_DENSITY, . . . This interface is required to be functional for all applications.

`index_in`	Index of state vector element about which information is requested.
`location`	The location of state variable element.
var_kind	The generic kind of the state variable element.

call model_interpolate(x, location, ikind, obs_val, istatus)

```
real(r8), dimension(:), intent(in)  :: x
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: ikind
real(r8),               intent(out) :: obs_val
integer,                intent(out) :: istatus
```

Given a state vector, a location, and a model state variable kind interpolates the state variable field to that location and returns the value in obs_val. The istatus variable should be returned as 0 unless there is some problem in computing the interpolation in which case a positive value should be returned. The ikind variable is one of the KIND parameters defined in the *MODULE obs_kind_mod* file and defines which generic kind of item is being interpolated.

`x`	A model state vector.
`location`	Location to which to interpolate.
`itype`	Kind of state field to be interpolated.
`obs_val`	The interpolated value from the model.
`istatus`	Integer value returning 0 for success. Other values can be defined for various failures.

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

Returns the smallest useful forecast length (time step) of the model. This is set by `input.nml:assimilation_period_seconds` and must be an integer number of TIEGCM dynamical timesteps (as specified by `tiegcm.nml:STEP`) AND be able to be expressed by `tiegcm.nml:STOP`. Since `STOP` has three components: day-of-year, hour, and minute, the `assimilation_period_seconds` must be an integer number of minutes.

`var`	Smallest forecast step of model.

call static_init_model()

Called to do one-time initialization of the model. There are no input arguments. `static_init_model` reads the DART and TIEGCM namelists and reads the grid geometry and constructs the shape of the DART vector given the TIEGCM variables specified in the DART namelist.

call end_model()

Does all required shutdown and clean-up needed.

call init_time(time)

```
type(time_type), intent(out) :: time
```

This is a NULL INTERFACE for TIEGCM. If `input.nml:start_from_restart == .FALSE.`, this routine is called and will generate a fatal error.

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

This is a NULL INTERFACE for TIEGCM. If `input.nml:start_from_restart == .FALSE.`, this routine is called and will generate a fatal error.

ierr = nc_write_model_atts(ncFileID)

```
integer              :: nc_write_model_atts
integer, intent(in)  :: ncFileID
```

This routine writes the model-specific attributes to a netCDF file. This includes the coordinate variables and any metadata, but NOT the model state vector. We do have to allocate SPACE for the model state vector, but that variable gets filled as the model advances. If `input.nml:model_nml:output_state_vector == .TRUE.`, the DART state vector is written as one long vector. If `input.nml:model_nml:output_state_vector == .FALSE.`, the DART state vector is reshaped into the original TIEGCM variables and those variables are written.

`ncFileID`	Integer file descriptor to previously-opened netCDF file.
`ierr`	Returns a 0 for successful completion.

ierr = nc_write_model_vars(ncFileID, statevec, copyindex, timeindex)

```
integer                                :: nc_write_model_vars
integer,                    intent(in) :: ncFileID
real(r8), dimension(:),     intent(in) :: statevec
integer,                    intent(in) :: copyindex
integer,                    intent(in) :: timeindex
```

This routine writes the DART state vector to a netCDF file. If `input.nml:model_nml:output_state_vector == .TRUE.`, the DART state vector is written as one long vector. If `input.nml:model_nml:output_state_vector == .FALSE.`, the DART state vector is reshaped into the original TIEGCM variables and those variables are written.

`ncFileID`	file descriptor to previously-opened netCDF file.
`statevec`	A model state vector.
`copyindex`	Integer index of copy to be written.
`timeindex`	The timestep counter for the given state.
`ierr`	Returns 0 for normal completion.

call pert_model_state(state, pert_state, interf_provided)

```
real(r8), dimension(:), intent(in)  :: state
real(r8), dimension(:), intent(out) :: pert_state
logical,                intent(out) :: interf_provided
```

`pert_model_state` is intended to take a single model state vector and perturbs it in some way to generate initial conditions for spinning up ensembles. TIEGCM does this is a manner that is different than most other models. The F10_7 parameter must be included in the DART state vector as a QTY_1D_PARAMETER and gaussian noise is added to it. That value must be conveyed to the tiegcm namelist and used to advance the model.

Most other models simply add noise with certain characteristics to the model state.

`state`	State vector to be perturbed.
`pert_state`	Perturbed state vector.
`interf_provided`	This is returned as .TRUE. since the routine exists. A value of .FALSE. would indicate that the default DART routine should just add noise to every element of state.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
real(r8),             intent(in)    :: maxdist
```

This is a PASS-THROUGH routine, the actual routine is the default one in `location_mod`. In distance computations any two locations closer than the given `maxdist` will be considered close by the `get_close_obs()` routine. `get_close_maxdist_init` is listed on the `use` line for the locations_mod, and in the public list for this module, but has no subroutine declaration and no other code in this module.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(num)
```

This is a PASS-THROUGH routine. The default routine in the location module precomputes information to accelerate the distance computations done by `get_close_obs()`. Like the other PASS-THROUGH ROUTINES it is listed on the use line for the locations_mod, and in the public list for this module, but has no subroutine declaration and no other code in this module:

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs_loc, obs_kind, num_close, close_ind [, dist])

```
type(get_close_type), intent(in)  :: gc
type(location_type),  intent(in)  :: base_obs_loc
integer,              intent(in)  :: base_obs_kind
type(location_type),  intent(in)  :: obs_loc(:)
integer,              intent(in)  :: obs_kind(:)
integer,              intent(out) :: num_close
integer,              intent(out) :: close_ind(:)
real(r8), optional,   intent(out) :: dist(:)
```

Given a location and kind, compute the distances to all other locations in the `obs_loc` list. The return values are the number of items which are within maxdist of the base, the index numbers in the original obs_loc list, and optionally the distances. The `gc` contains precomputed information to speed the computations.

This is different than the default `location_mod:get_close_obs()` in that it is possible to modify the 'distance' based on the DART 'kind'. This allows one to apply specialized localizations.

`gc`	The get_close_type which stores precomputed information about the locations to speed up searching
`base_obs_loc`	Reference location. The distances will be computed between this location and every other location in the obs list
`base_obs_kind`	The kind of base_obs_loc
`obs_loc`	Compute the distance between the base_obs_loc and each of the locations in this list
`obs_kind`	The corresponding kind of each item in the obs list
`num_close`	The number of items from the obs_loc list which are within maxdist of the base location
`close_ind`	The list of index numbers from the obs_loc list which are within maxdist of the base location
dist	If present, return the distance between each entry in the close_ind list and the base location. If not present, all items in the obs_loc list which are closer than maxdist will be added to the list but the overhead of computing the exact distances will be skipped.

call ens_mean_for_model(ens_mean)

```
real(r8), dimension(:), intent(in) :: ens_mean
```

A model-size vector with the means of the ensembles for each of the state vector items. The model should save a local copy of this data if it needs to use it later to compute distances or other values. This routine is called after each model advance and contains the updated means.

`ens_mean`	State vector containing the ensemble mean.

TIEGCM public routines

call tiegcm_to_dart_vector(statevec, model_time)

```
real(r8), dimension(:), intent(out) :: statevec
type(time_type),        intent(out) :: model_time
```

Read TIEGCM fields from the TIEGCM restart file and/or TIEGCM secondary file and pack them into a DART vector.

`statevec`	variable that contains the DART state vector
`model_time`	variable that contains the LAST TIME in the TIEGCM restart file.

call dart_vector_to_tiegcm(statevec, dart_time)

```
real(r8), dimension(:), intent(in) :: statevec
type(time_type),        intent(in) :: dart_time
```

Unpacks a DART vector and updates the TIEGCM restart file variables. Only those variables designated as 'UPDATE' are put into the TIEGCM restart file. All variables are written to the DART diagnostic files **prior** to the application of any "clamping". The variables **are "clamped"** before being written to the TIEGCM restart file. The clamping limits are specified in columns 3 and 4 of `&model_nml:variables`.
The time of the DART state is compared to the time in the restart file to ensure that we are not improperly updating a restart file.

`statevec`	Variable containing the DART state vector.
`dart_time`	Variable containing the time of the DART state vector.

var = get_f107_value(x)

```
real(r8)                           :: get_f107_value
real(r8), dimension(:), intent(in) :: x
```

If the F10_7 value is part of the DART state, return that value. If it is not part of the DART state, just return the F10_7 value from the TIEGCM namelist.

`x`	Variable containing the DART state vector.
`var`	The f10_7 value.

call test_interpolate(x, locarray)

```
real(r8), dimension(:), intent(in) :: x
real(r8), dimension(3), intent(in) :: locarray
```

This function is **only** used by *program model_mod_check* and can be modified to suit your needs. `test_interpolate()` exercises `model_interpolate()`, `get_state_meta_data()`, `static_init_model()` and a host of supporting routines.

`x`	variable containing the DART state vector.
`locarray`	variable containing the location of interest. locarray(1) is the longitude (in degrees East) locarray(2) is the latitude (in degrees North) locarray(3) is the height (in meters).

6.54.7 Files

`filename`	purpose
`tiegcm.nml`	TIEGCM control file modified to control starting and stopping.
`input.nml`	to read the model_mod namelist
`tiegcm_restart_p.nc`	both read and modified by the TIEGCM model_mod
`tiegcm_s.nc`	read by the GCOM model_mod for metadata purposes.
`namelist_update`	DART file containing information useful for starting and stopping TIEGCM. `advance_model.csh` uses this to update the TIEGCM file `tiegcm.nml`
`dart_log.out`	the run-time diagnostic output
`dart_log.nml`	the record of all the namelists (and their values) actually USED
log_advance.nnnn.txt	the run-time output of everything that happens in `advance_model.csh`. This file will be in the *advance_tempnnnn* directory.

6.54.8 References

- Matsuo, T., and E. A. Araujo-Pradere (2011), Role of thermosphere-ionosphere coupling in a global ionosphere specification, *Radio Science*, **46**, RS0D23, doi:10.1029/2010RS004576
-
- Lee, I. T., T, Matsuo, A. D. Richmond, J. Y. Liu, W. Wang, C. H. Lin, J. L. Anderson, and M. Q. Chen (2012), Assimilation of FORMOSAT-3/COSMIC electron density profiles into thermosphere/Ionosphere coupling model by using ensemble Kalman filter, *Journal of Geophysical Research*, **117**, A10318, doi:10.1029/2012JA017700
-
- Matsuo, T., I. T. Lee, and J. L. Anderson (2013), Thermospheric mass density specification using an ensemble Kalman filter, *Journal of Geophysical Research*, **118**, 1339-1350, doi:10.1002/jgra.50162
-
- Lee, I. T., H. F. Tsai, J. Y. Liu, Matsuo, T., and L. C. Chang (2013), Modeling impact of FORMOSAT-7/COSMIC-2 mission on ionospheric space weather monitoring, *Journal of Geophysical Research*, **118**, 6518-6523, doi:10.1002/jgra.50538

-
- Matsuo, T. (2014), Upper atmosphere data assimilation with an ensemble Kalman filter, in Modeling the Ionosphere-Thermosphere System, *Geophys. Monogr. Ser.*, vol. 201, edited by J. Huba, R. Schunk, and G. Khazanov, pp. 273-282, John Wiley & Sons, Ltd, Chichester, UK, doi:10.1002/9781118704417
-
- Hsu, C.-H., T. Matsuo, W. Wang, and J. Y. Liu (2014), Effects of inferring unobserved thermospheric and ionospheric state variables by using an ensemble Kalman filter on global ionospheric specification and forecasting, *Journal of Geophysical Research*, **119**, 9256-9267, doi:10.1002/2014JA020390
-
- Chartier, A., T. Matsuo, J. L. Anderson, G. Lu, T. Hoar, N. Collins, A. Coster, C. Mitchell, L. Paxton, G. Bust (2015), Ionospheric Data Assimilation and Forecasting During Storms, *Journal of Geophysical Research*, under review
-

6.55 WRF-Hydro

6.55.1 Overview

The Weather Research and Forecasting Hydrologic Model (WRF-Hydro) is a community modeling system and framework for hydrologic modeling and model coupling. WRF-Hydro is configured to use the Noah-MP Land Surface Model to simulate land surface processes. Combined with DART, the facility is called *HydroDART*.

The development of HydroDART was a collaboration between **James McCreight** of the Research Applications Laboratory of NCAR and **Moha Gharamti** of the Data Assimilation Research Section of NCAR.

Streamflow assimilation is an active area of research and provides many interesting research challenges.

6.55.2 Description of this directory within the DART repository

Contents of the `$DARTROOT/models/wrf_hydro/`:

```
├── ensemble_config_files/
    # Files which configure ensembles in wrfhydropy.
├── experiment_config_files/
    # File which configure hydro_dart_py experiments.
├── hydro_dart_py/
    # Python package/library for configuring and executing experiments.
├── python/
    # Python scripts for various purposes.
├── R/
    # R scripts for various purposes.
├── shell_scripts/
    # Shell scripts for various purposes.
├── templates/
    # Obsolete?
├── work/
    # Dart executables build directory and other testing.
├── model_mod.html
    # The model_mod documentation.
├── model_mod.nml
```

(continues on next page)

(continued from previous page)

```
│      # The model_mod namelist (subsumed by work/input.nml)
├── model_mod.f90
│      # The model_mod code.
├── noah_hydro_mod.f90
│      # Some model_mod interfaces more specific to Noah?
├── create_identity_streamflow_obs.f90
│      # For creating identity streamflow obs for the NHDPlus-based
│      # channel-network configuration of WRF-Hydro.
├── README.rst
       # This file.
```

6.55.3 To set up an experiment

To set up an experiment, consult the `./python/experiment` directory.

6.55.4 Description of external directories on GLADE

The gridded version of the model has bits/bobs in these directories:

- `/gpfs/fs1/work/jamesmcc/domains/public/croton_NY/Gridded/DOMAIN`
- `/gpfs/fs1/work/jamesmcc/domains/public/croton_NY/Gridded/RESTART`

Only the gridcells with flow are retained in the `qlink[1,2]`, `hlink` variables, so they must be unpacked in EXACTLY the same way as wrfHydo packs them from the grid to their 'sparse' representation.

6.55.5 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
     assimilation_period_days       = 0
     assimilation_period_seconds    = 3600
     lsm_model_choice               = 'noahMP'
     model_perturbation_amplitude   = 0.5
     perturb_distribution           = 'lognormal'
     max_link_distance              = 2000.0
     streamflow_4_local_multipliers = 0.0001
     debug                          = 0
     domain_order                   = 'hydro'
     domain_shapefiles              = 'restart.hydro.nc'
     lsm_variables     = 'SH2O',               'QTY_SOIL_LIQUID_WATER',  '0.0',    'NA',
↪'NOUPDATE',
                         'SUBSURFACE_FLUX',    'QTY_SUBSURFACE',         '0.0',    'NA',
↪'NOUPDATE',
                         'OVERLAND_FLUX',      'QTY_OVERLAND_FLOW',      '0.0',    'NA',
↪'NOUPDATE'
     hydro_variables   = 'qlink1',             'QTY_STREAM_FLOW',        '0.0',    'NA',
↪'UPDATE',
                         'z_gwsubbas',         'QTY_AQUIFER_WATER',      'NA',     'NA',
↪'UPDATE'
```

(continues on next page)

(continued from previous page)

```
     parameters       = 'qBucketMult',       'QTY_BUCKET_MULTIPLIER', '0.001', '50',
↪'UPDATE',
                        'qSfcLatRunoffMult', 'QTY_RUNOFF_MULTIPLIER', '0.001', '50',
↪'UPDATE'
/
```

This namelist is read from a file called `input.nml`. This namelist provides control over the assimilation period for the model. All observations within (+/-) half of the assimilation period are assimilated. The assimilation period is the minimum amount of time the model can be advanced, and checks are performed to ensure that the assimilation window is a multiple of the NOAH model dynamical timestep.

Item	Type	Description
assimilation_period_days	integer	The number of days to advance the model for each assimilation. [default: `1`]
assimilation_period_seconds	integer	In addition to `assimilation_period_days`, the number of seconds to advance the model for each assimilation. [default: `0`]
lsm_model_choice	character(len=128)	case-insensitive specification of the Land Surface model. Valid values are `noahmp` and `noahmp_36`
model_perturbation_amplitude	real(r8)	The amount of noise to add when trying to perturb a single state vector to create an ensemble. Only used when `input.nml` is set with `&filter_nml:start_from_restart = .false.`. See also Generating the initial ensemble. units: standard deviation of the specified distribution the mean at the value of the state vector element.
perturb_distribution	character(len=256)	The switch to determine the distribution of the perturbations used to create an initial ensemble from a single model state. Valid values are : `lognormal` or `gaussian`
max_link_distance	real(r8)	The along-the-stream localization distance. In meters.
streamflow_4_local_multipliers	real(r8)	
debug	integer	The switch to specify the run-time verbosity. • `0` is as quiet as it gets • `> 1` provides more run-time messages • `> 5` provides ALL run-time messages All values above 0 will also write a netCDF file of the grid information and perform a grid interpolation test. [default: `0`]
domain_order	character(len=256):: dimension(3)	There are three possible domains to include in the HydroDART state: `hydro`, `parameters`, `lsm` This variable specifies the ordering of the domains.
domain_shapefiles	character(len=256):: dimension(3)	There are input files used to determine the shape of the input variables and any geographic metadata. They must be specified in the same order as listed in `domain_order`
lsm_variables	character(len=32):: dimension(5,40)	The list of variable names in the NOAH restart file to use to create the DART state vector and their corresponding DART QUANTITY. [see example below]
hydro_variables	character(len=32):: dimension(5,40)	The list of variable names in the channel model file to use to create the DART state vector and their

The columns of `lsm_variables`, `hydro_variables`, and `parameters` needs some explanation. Starting with the column 5, `UPDATE` denotes whether or not to replace the variable with the Posterior (i.e. assimilated) value. Columns 3 and 4 denote lower and upper bounds that should be enforced when writing to the files used to restart the model. These limits are not enforced for the DART diagnostic files. Column 2 specifies the relationship between the netCDF variable name for the model and the corresponding DART QUANTITY.

Support for these QUANTITYs is provided by running `preprocess` with the following namelist settings:

```
&preprocess_nml
           overwrite_output = .true.
    input_obs_kind_mod_file = '../../../assimilation_code/modules/observations/
↪DEFAULT_obs_kind_mod.F90'
   output_obs_kind_mod_file = '../../../assimilation_code/modules/observations/obs_
↪kind_mod.f90'
     input_obs_def_mod_file = '../../../observations/forward_operators/DEFAULT_obs_
↪def_mod.F90'
    output_obs_def_mod_file = '../../../observations/forward_operators/obs_def_mod.f90
↪'
   input_files              = '../../../observations/forward_operators/obs_def_
↪streamflow_mod.f90',
                              '../../../observations/forward_operators/obs_def_land_
↪mod.f90',
                              '../../../observations/forward_operators/obs_def_COSMOS_
↪mod.f90'
  /
```

6.56 WRF

6.56.1 Overview

DART interface module for the WRF model. This page documents the details of the module compiled into DART that interfaces with the WRF data in the state vector.

6.56.2 WRF+DART Tutorial

There is additional overview and tutorial documentation for running a WRF/DART assimilation in `./tutorial/README.md`.

Please work through the tutorial in order to learn how to run WRF and DART.

Items of Note

- The `model_mod` reads WRF netCDF files directly to acquire the model state data. The `wrf_to_dart` and `dart_to_wrf` programs are no longer necessary.
- A netCDF file named `wrfinput_d01` is required and must be at the same resolution and have the same surface elevation data as the files converted to create the DART initial conditions. No data will be read from this file, but the grid information must match exactly.

The model interface code supports WRF configurations with multiple domains. Data for all domains is read into the DART state vector. During the computation of the forward operators (getting the estimated observation values from each ensemble member), the search starts in the domain with the highest number, which is generally the finest nest or one of multiple finer nests. The search stops as soon as a domain contains the observation location, working its way from largest number to smallest number domain, ending with domain 1. For example, in a 4 domain case the data in the state vector that came from `wrfinput_d04` is searched first, then `wrfinput_d03`, `wrfinput_d02`, and finally `wrfinput_d01`.

The forward operator is computed from the first domain grid that contains the lat/lon of the observation. During the assimilation phase, when the state values are adjusted based on the correlations and assimilation increments, all points in all domains that are within the localization radius are adjusted, regardless of domain. The impact of an observation on the state depends only on the distance between the observation and the state vector point, and the regression coefficient based on the correlation between the distributions of the ensemble of state vector points and the ensemble of observation forward operator values.

The fields from WRF that are copied into the DART state vector are controlled by namelist. See below for the documentation on the &model_nml entries. The state vector should include all fields needed to restart a WRF run. There may be additional fields needed depending on the microphysics scheme selected. See the ascii file `wrf_state_variables_table` in the `models/wrf` directory for a list of fields that are often included in the DART state.

6.56.3 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   default_state_variables     = .true.
   wrf_state_variables         = 'NULL'
   wrf_state_bounds            = 'NULL'
   num_domains                 = 1
   calendar_type               = 3
   assimilation_period_seconds = 21600
   allow_obs_below_vol         = .false.
   vert_localization_coord     = 3
   center_search_half_length   = 500000.
   center_spline_grid_scale    = 10
   circulation_pres_level      = 80000.0
   circulation_radius          = 108000.0
   sfc_elev_max_diff           = -1.0
   polar                       = .false.
   periodic_x                  = .false.
   periodic_y                  = .false.
   scm                         = .false.
   allow_perturbed_ics         = .false.   # testing purposes only
/

   # Notes for model_nml:
   # (1) vert_localization_coord must be one of:
   #     1 = model level
   #     2 = pressure
   #     3 = height
   #     4 = scale height
   # (2) see bottom of this file for explanations of polar, periodic_x,
   #     periodic_y, and scm
   # (3) calendar = 3 is GREGORIAN, which is what WRF uses.
   # (4) if 'default_state_variables' is .true. the model_mod.f90 code will
   #     fill the state variable table with the following wrf vars:
   #        U, V, W, PH, T, MU
   #     you must set it to false before you change the value
```

(continues on next page)

(continued from previous page)

```
   #     of 'wrf_state_variables' and have it take effect.
   # (5) the format for 'wrf_state_variables' is an array of 5 strings:
   #     wrf netcdf variable name, dart QTY_xxx string, type string (must be
   #     unique, will soon be obsolete, we hope), 'UPDATE', and '999' if the
   #     array is part of all domains.  otherwise, it is a string with the domain
   #     numbers (e.g. '12' for domains 1 and 2, '13' for domains 1 and 3).
   #   example:
   # wrf_state_variables='U','QTY_U_WIND_COMPONENT','TYPE_U','UPDATE','999',
   #                     'V','QTY_V_WIND_COMPONENT','TYPE_V','UPDATE','999',
   #                     'W','QTY_VERTICAL_VELOCITY','TYPE_W','UPDATE','999',
   #                     'T','QTY_POTENTIAL_TEMPERATURE','TYPE_T','UPDATE','999',
   #                     'PH','QTY_GEOPOTENTIAL_HEIGHT','TYPE_GZ','UPDATE','999',
   #                     'MU','QTY_PRESSURE','TYPE_MU','UPDATE','999',
   #                     'QVAPOR','QTY_VAPOR_MIXING_RATIO','TYPE_QV','UPDATE','999',
   #                     'QCLOUD','QTY_CLOUD_LIQUID_WATER','TYPE_QC','UPDATE','999',
   #                     'QRAIN','QTY_RAINWATER_MIXING_RATIO','TYPE_QR','UPDATE','999
↪',
   #                     'U10','QTY_U_WIND_COMPONENT','TYPE_U10','UPDATE','999',
   #                     'V10','QTY_V_WIND_COMPONENT','TYPE_V10','UPDATE','999',
   #                     'T2','QTY_TEMPERATURE','TYPE_T2','UPDATE','999',
   #                     'TH2','QTY_POTENTIAL_TEMPERATURE','TYPE_TH2','UPDATE','999',
   #                     'Q2','QTY_SPECIFIC_HUMIDITY','TYPE_Q2','UPDATE','999',
   #                     'PSFC','QTY_PRESSURE','TYPE_PS','UPDATE','999',
   # (6) the format for 'wrf_state_bounds' is an array of 4 strings:
   #     wrf netcdf variable name, minimum value, maximum value, and either
   #     FAIL or CLAMP.  FAIL will halt the program if an out of range value
   #     is detected.  CLAMP will set out of range values to the min or max.
   #     The special string 'NULL' will map to plus or minus infinity and will
   #     not change the values.  arrays not listed in this table will not
   #     be changed as they are read or written.
   #
   #
   # polar and periodic_x are used in global wrf.  if polar is true, the
   # grid interpolation routines will wrap over the north and south poles.
   # if periodic_x is true, when the east and west edges of the grid are
   # reached the interpolation will wrap.  note this is a separate issue
   # from regional models which cross the GMT line; those grids are marked
   # as having a negative offset and do not need to wrap; this flag controls
   # what happens when the edges of the grid are reached.

   # the scm flag is used for the 'single column model' version of WRF.
   # it needs the periodic_x and periodic_y flags set to true, in which
   # case the X and Y directions are periodic; no collapsing of the grid
   # into a single location like the 3d-spherical polar flag implies.
```

Description of each namelist entry

Item	Type	Description
default_state_variables	logical	If *.true.*, the dart state vector contains the fields U, V, W, PH, T, MU, in that order, and only those. Any values listed in the *wrf_state_variables* namelist item will be ignored.
wrf_state_variables	character(:, 5)	A 2D array of strings, 5 per wrf array to be added to the dart state vector. If *default_state_variables* is *.true.*, this is ignored. When *.false.*, this list of array names controls which arrays and the order that they are added to the state vector. The 5 strings are: 1. WRF field name - must match netcdf name exactly 2. DART KIND name - must match a valid DART QTY_xxx exactly 3. TYPE_NN - will hopefully be obsolete, but for now NN should match the field name. 4. the string UPDATE. at some future point, non-updatable fields may become part of the state vector. 5. A numeric string listing the domain numbers this array is part of. The specical string 999 means all domains. For example, '12' means domains 1 and 2, '13' means 1 and 3.
wrf_state_bounds	character(:, 4)	A 2D array of strings, 4 per wrf array. During the copy of data to and from the wrf netcdf file, variables listed here will have minimum and maximum values enforced. The 4 strings are: 1. WRF field name - must match netcdf name exactly 2. Minimum – specified as a string but must be a numeric value (e.g. '0.1') Can be 'NULL' to allow any minimum value. 3. Maximum – specified as a string but must be a numeric value (e.g. '0.1') Can be 'NULL' to allow any maximum value. 4. Action – valid strings are 'CLAMP', 'FAIL'. 'FAIL' means if a value is found outside the range, the code fails with an error. 'CLAMP'

The following items used to be in the WRF namelist but have been removed. The first 4 are no longer needed, and the last one was moved to the `&dart_to_wrf_nml` namelist in 2010. In the Lanai release having these values in the namelist does not cause a fatal error, but more recent versions of the code will fail if any of these values are specified. Remove them from your namelist to avoid errors.

Item	Type	Description
`surf_obs`	logical	OBSOLETE – now an error to specify this.
`soil_data`	logical	OBSOLETE – now an error to specify this.
`h_diab`	logical	OBSOLETE – now an error to specify this.
`num_moist_vars`	integer	OBSOLETE – now an error to specify this.
`adv_mod_command`	character(len=32)	OBSOLETE – now an error to specify this.

6.56.4 Files

- model_nml in input.nml
- wrfinput_d01, wrfinput_d02, … (one file for each domain)
- netCDF output state diagnostics files

6.56.5 References

http://www2.mmm.ucar.edu/wrf/users/docs/user_guide_V3/contents.html

6.57 Contributors' guide

6.57.1 Contributing to DART

This section describes how you can contribute your work to DART. Because DART is an open-source project, your contributions are welcome. Many user-provided contributions have widely benefited the earth science community.

To ensure you aren't duplicating efforts, contact DAReS staff by emailing dart@ucar.edu before you expend considerable development time.

All of the source code is hosted in the DART GitHub repository.

Before you start developing, you should be familiar with the GitHub workflow. The GitHub worflow involves:

1. Creating a *fork* of the DART project. A fork is a publically visible copy of the repository that is stored in your GitHub account.
2. Creating a *branch* for your feature with an appropriate name for your project, and when you are finished with your changes you can commit them to your fork. After testing locally on your machine, you can push them to your fork.

   Important: At this point, everyone can see the changes you made on your fork.

   When you are ready to begin the conversation about merging your work into the original project (called the DART repository master), you can create a pull request, which will show your changes. After reviewing and testing your changes, the pull request will be addressed appropriately by the DART development team.

6.57.2 Keeping your work private until you publish

You may want to keep your work private until it is ready for publication or public viewing.

Follow these steps to hide sensitive code until you are ready to contribute it to DART your work has been published.

1. First, create a public fork of the DART repository by following the steps listed above.
2. Next, create a private repository on GitHub.com. The name of your private repository is arbitrary, since only you and your private collaborators can see it.
3. Add your public fork as a remote repository of your private repository. Your remote repository can be named “public_fork” or “upstream.”
4. Add additional team members, if necessary.
5. Instead of pulling and pushing from your public fork, develop on your private repository.

Note: Only three collaborators are allowed on a free non-institutional private repository. DAReS staff can collaborate with you on your private repository, but keep this three collaborator limit in mind if you using a free GitHub account.

6.58 Requesting features and reporting bugs

DAReS staff uses GitHub’s project management tools to track development.

To request a feature or to request a bug fix, use the GitHub issue tracker on the DART repository.

6.59 Mailing list

DAReS staff send periodic updates to DART users. These updates summarize changes to the DART repository, including recent bug fixes.

The mailing list is not generally used for discussion so emails are infrequent.

To subscribe to the DART users mailing list, see Dart-users.

6.60 DART Manhattan Release Notes

6.60.1 Getting started

What’s required

1. a Fortran 90 compiler
2. a NetCDF library including the F90 interfaces
3. the C shell
4. (optional, to run in parallel) an MPI library

DART has been tested on many Fortran compilers and platforms. We don’t have any platform-dependent code sections and we use only the parts of the language that are portable across all the compilers we have access to. We explicitly set the Fortran 'kind' for all real values and do not rely on autopromotion or other compile-time flags to set the default byte size for numbers. It is possible that some model-specific interface code from outside sources may have specific compiler flag requirements; see the documentation for each model. The low-order models and all common portions of the DART code compile cleanly.

DART uses the NetCDF self-describing data format with a particular metadata convention to describe output that is used to analyze the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools.

Since most of the models being used with DART are written in Fortran and run on various UNIX or *nix platforms, the development environment for DART is highly skewed to these machines. We do most of our development on a small linux workstation and a mac laptop running OSX 10.x, and we have an extensive test network.

What's nice to have

ncview: DART users have used ncview to create graphical displays of output data fields.

NCO: The NCO tools are able to perform operations on NetCDF files like concatenating, slicing, and dicing.

Matlab®: A set of Matlab® scripts designed to produce graphical diagnostics from DART.

MPI: The DART system includes an MPI option. MPI stands for 'Message Passing Interface', and is both a library and run-time system that enables multiple copies of a single program to run in parallel, exchange data, and combine to solve a problem more quickly. DART does **NOT** require MPI to run; the default build scripts do not need nor use MPI in any way. However, for larger models with large state vectors and large numbers of observations, the data assimilation step will run much faster in parallel, which requires MPI to be installed and used. However, if multiple ensembles of your model fit comfortably (in time and memory space) on a single processor, you need read no further about MPI.

Types of input

DART programs can require three different types of input. First, some of the DART programs, like those for creating synthetic observational datasets, require interactive input from the keyboard. For simple cases this interactive input can be made directly from the keyboard. In more complicated cases a file containing the appropriate keyboard input can be created and this file can be directed to the standard input of the DART program. Second, many DART programs expect one or more input files in DART specific formats to be available. For instance, `perfect_model_obs`, which creates a synthetic observation set given a particular model and a description of a sequence of observations, requires an input file that describes this observation sequence. At present, the observation files for DART are in a custom format in either human-readable ascii or more compact machine-specific binary. Third, many DART modules (including main programs) make use of the Fortran90 namelist facility to obtain values of certain parameters at run-time. All programs look for a namelist input file called `input.nml` in the directory in which the program is executed. The `input.nml` file can contain a sequence of individual Fortran90 namelists which specify values of particular parameters for modules that compose the executable program.

6.60.2 Installation

This document outlines the installation of the DART software and the system requirements. The entire installation process is summarized in the following steps:

1. Determine which F90 compiler is available.
2. Determine the location of the `NetCDF` library.
3. Download the DART software into the expected source tree.
4. Modify certain DART files to reflect the available F90 compiler and location of the appropriate libraries.
5. Build the executables.

We have tried to make the code as portable as possible, but we do not have access to all compilers on all platforms, so there are no guarantees. We are interested in your experience building the system, so please email me (Tim Hoar) thoar 'at' ucar 'dot' edu (trying to cut down on the spam).

After the installation, you might want to peruse the following.

- Running the Lorenz_63 Model.
- Using the Matlab® diagnostic scripts.
- A short discussion on bias, filter divergence and covariance inflation.
- And another one on synthetic observations.

You should *absolutely* run the DARTLAB interactive tutorial (if you have Matlab available) and look at the DARTLAB presentation slides Website *DART_LAB Tutorial* in the `DART_LAB` directory, and then take the tutorial in the `DART/tutorial` directory.

Requirements: an F90 compiler

The DART software has been successfully built on many Linux, OS/X, and supercomputer platforms with compilers that include GNU Fortran Compiler ("gfortran") (free), Intel Fortran Compiler for Linux and Mac OS/X, Portland Group Fortran Compiler, Lahey Fortran Compiler, Pathscale Fortran Compiler, and the Cray native compiler. Since recompiling the code is a necessity to experiment with different models, there are no binaries to distribute.

DART uses the NetCDF self-describing data format for the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. In particular, DART also makes use of the F90 interface to the library which is available through the `netcdf.mod` and `typesizes.mod` modules. *IMPORTANT*: different compilers create these modules with different "case" filenames, and sometimes they are not **both** installed into the expected directory. It is required that both modules be present. The normal place would be in the `netcdf/include` directory, as opposed to the `netcdf/lib` directory.

If the NetCDF library does not exist on your system, you must build it (as well as the F90 interface modules). The library and instructions for building the library or installing from an RPM may be found at the NetCDF home page: http://www.unidata.ucar.edu/packages/netcdf/

The location of the NetCDF library, `libnetcdf.a`, and the locations of both `netcdf.mod` and `typesizes.mod` will be needed by the makefile template, as described in the compiling section. Depending on the NetCDF build options, the Fortran 90 interfaces may be built in a separate library named `netcdff.a` and you may need to add `-lnetcdff` to the library flags.

6.60.3 Downloading the distribution

HURRAY! The DART source code is now distributed through an anonymous Subversion server! The **big** advantage is the ability to patch or update existing code trees at your discretion. Subversion (the client-side app is '**svn**') allows you to compare your code tree with one on a remote server and selectively update individual files or groups of files. Furthermore, now everyone has access to any version of any file in the project, which is a huge help for developers. I have a brief summary of the svn commands I use most posted at: http://www.image.ucar.edu/~thoar/svn_primer.html

The resources to develop and support DART come from our ability to demonstrate our growing user base. We ask that you register at our download site http://www.image.ucar.edu/DAReS/DART/DART_download and promise that the information will only be used to notify you of new DART releases and shown to our sponsers in an aggregated form: "Look - we have three users from Tonawanda, NY". After filling in the form, you will be directed to a website that has instructions on how to download the code.

svn has adopted the strategy that "disk is cheap". In addition to downloading the code, it downloads an additional copy of the code to store locally (in hidden .svn directories) as well as some administration files. This allows svn to perform some commands even when the repository is not available. It does double the size of the code tree for the initial download, but then future updates download just the changes, so they usually happen very quickly.

If you follow the instructions on the download site, you should wind up with a directory named `DART`. Compiling the code in this tree (as is usually the case) will necessitate much more space.

The code tree is very “bushy”; there are many directories of support routines, etc. but only a few directories involved with the customization and installation of the DART software. If you can compile and run ONE of the low-order models, you should be able to compile and run ANY of the low-order models. For this reason, we can focus on the Lorenz `63 model. Subsequently, the only directories with files to be modified to check the installation are: `DART/build_templates`, `DART/models/lorenz_63/work`, and `DART/diagnostics/matlab` (but only for analysis).

6.60.4 Customizing the build scripts – overview

DART executable programs are constructed using two tools: `make` and `mkmf`. The `make` utility is a very common piece of software that requires a user-defined input file that records dependencies between different source files. `make` then performs a hierarchy of actions when one or more of the source files is modified. The `mkmf` utility is a custom preprocessor that generates a `make` input file (named `Makefile`) and an example namelist *input.nml.program_default* with the default values. The `Makefile` is designed specifically to work with object-oriented Fortran90 (and other languages) for systems like DART.

`mkmf` requires two separate input files. The first is a `template’ file which specifies details of the commands required for a specific Fortran90 compiler and may also contain pointers to directories containing pre-compiled utilities required by the DART system. **This template file will need to be modified to reflect your system**. The second input file is a `path_names’ file which includes a complete list of the locations (either relative or absolute) of all Fortran90 source files that are required to produce a particular DART program. Each ‘path_names’ file must contain a path for exactly one Fortran90 file containing a main program, but may contain any number of additional paths pointing to files containing Fortran90 modules. An `mkmf` command is executed which uses the ‘path_names’ file and the mkmf template file to produce a `Makefile` which is subsequently used by the standard `make` utility.

Shell scripts that execute the mkmf command for all standard DART executables are provided as part of the standard DART software. For more information on `mkmf` see the FMS mkmf description.

One of the benefits of using `mkmf` is that it also creates an example namelist file for each program. The example namelist is called *input.nml.program_default*, so as not to clash with any exising `input.nml` that may exist in that directory.

Building and customizing the ‘mkmf.template’ file

A series of templates for different compilers/architectures exists in the `DART/build_templates/` directory and have names with extensions that identify the compiler, the architecture, or both. This is how you inform the build process of the specifics of your system. Our intent is that you copy one that is similar to your system into `mkmf.template` and customize it. For the discussion that follows, knowledge of the contents of one of these templates (i.e. `mkmf.template.gfortran`) is needed. Note that only the LAST lines are shown here, the head of the file is just a big comment (worth reading, btw).

. . . MPIFC = mpif90 MPILD = mpif90 FC = gfortran LD = gfortran NETCDF = /usr/local INCS = ${NETCDF}/include FFLAGS = -O2 -I$(INCS) LIBS = -L${NETCDF}/lib -lnetcdf LDFLAGS = -I$(INCS) $(LIBS)

Essentially, each of the lines defines some part of the resulting `Makefile`. Since `make` is particularly good at sorting out dependencies, the order of these lines really doesn’t make any difference. The `FC = gfortran` line ultimately defines the Fortran90 compiler to use, etc. The lines which are most likely to need site-specific changes start with `FFLAGS` and `NETCDF`, which indicate where to look for the NetCDF F90 modules and the location of the NetCDF library and modules.

If you have MPI installed on your system `MPIFC, MPILD` dictate which compiler will be used in that instance. If you do not have MPI, these variables are of no consequence.

Netcdf

Modifying the `NETCDF` value should be relatively straightforward.

Change the string to reflect the location of your NetCDF installation containing `netcdf.mod` and `typesizes.mod`. The value of the `NETCDF` variable will be used by the `FFLAGS, LIBS,` and `LDFLAGS` variables.

FFLAGS

Each compiler has different compile flags, so there is really no way to exhaustively cover this other than to say the templates as we supply them should work – depending on the location of your NetCDF. The low-order models can be compiled without a `-r8` switch, but the `bgrid_solo` model cannot.

Libs

The Fortran 90 interfaces may be part of the default `netcdf.a` library and `-lnetcdf` is all you need. However it is also common for the Fortran 90 interfaces to be built in a separate library named `netcdff.a`. In that case you will need `-lnetcdf` and also `-lnetcdff` on the **LIBS** line. This is a build-time option when the NetCDF libraries are compiled so it varies from site to site.

Customizing the 'path_names_*' file

Several `path_names_*` files are provided in the `work` directory for each specific model, in this case: `DART/models/lorenz_63/work`. Since each model comes with its own set of files, the `path_names_*` files need no customization.

6.60.5 Building the Lorenz_63 DART project

DART executables are constructed in a `work` subdirectory under the directory containing code for the given model. From the top-level DART directory change to the L63 work directory and list the contents:

cd DART/models/lorenz_63/work ls -1

With the result:

```
filter_input.cdl
filter_input_list.txt
filter_output_list.txt
input.nml
input.workshop.nml
mkmf_create_fixed_network_seq
```

(continues on next page)

(continued from previous page)

```
mkmf_create_obs_sequence
mkmf_filter
mkmf_obs_diag
mkmf_obs_sequence_tool
mkmf_perfect_model_obs
mkmf_preprocess
obs_seq.final
obs_seq.in
obs_seq.out
obs_seq.out.average
obs_seq.out.x
obs_seq.out.xy
obs_seq.out.xyz
obs_seq.out.z
path_names_create_fixed_network_seq
path_names_create_obs_sequence
path_names_filter
path_names_obs_diag
path_names_obs_sequence_tool
path_names_perfect_model_obs
path_names_preprocess
perfect_input.cdl
quickbuild.csh
set_def.out
workshop_setup.csh
```

In all the `work` directories there will be a `quickbuild.csh` script that builds or rebuilds the executables. The following instructions do this work by hand to introduce you to the individual steps, but in practice running quickbuild will be the normal way to do the compiles.

There are seven `mkmf_`*xxxxxx* files for the programs

1. `preprocess`,
2. `create_obs_sequence`,
3. `create_fixed_network_seq`,
4. `perfect_model_obs`,
5. `filter`,
6. `obs_sequence_tool`, and
7. `obs_diag`,

along with the corresponding `path_names_`*xxxxxx* files. There are also files that contain initial conditions, NetCDF output, and several observation sequence files, all of which will be discussed later. You can examine the contents of one of the `path_names_`*xxxxxx* files, for instance `path_names_filter`, to see a list of the relative paths of all files that contain Fortran90 modules required for the program `filter` for the L63 model. All of these paths are relative to your `DART` directory. The first path is the main program (`filter.f90`) and is followed by all the Fortran90 modules used by this program (after preprocessing).

The `mkmf_`*xxxxxx* scripts are cryptic but should not need to be modified – as long as you do not restructure the code tree (by moving directories, for example). The function of the `mkmf_`*xxxxxx* script is to generate a `Makefile` and an *input.nml.program_default* file. It does not do the compile; `make` does that:

csh mkmf_preprocess make

The first command generates an appropriate Makefile and the input.nml.preprocess_default file. The second command results in the compilation of a series of Fortran90 modules which ultimately produces an executable file: preprocess. Should you need to make any changes to the DART/build_templates/mkmf.template, you will need to regenerate the Makefile.

The preprocess program actually builds source code to be used by all the remaining modules. It is **imperative** to actually **run** preprocess before building the remaining executables. This is how the same code can assimilate state vector 'observations' for the Lorenz_63 model and real radar reflectivities for WRF without needing to specify a set of radar operators for the Lorenz_63 model!

preprocess reads the &preprocess_nml namelist to determine what observations and operators to incorporate. For this exercise, we will use the values in input.nml. preprocess is designed to abort if the files it is supposed to build already exist. For this reason, it is necessary to remove a couple files (if they exist) before you run the preprocessor. (The quickbuild.csh script will do this for you automatically.)

```
\rm -f ../../../observations/forward_operators/obs_def_mod.f90
\rm -f ../../../assimilation_code/modules/observations/obs_kind_mod.f90
./preprocess
ls -l  ../../../observations/forward_operators/obs_def_mod.f90
ls -l  ../../../assimilation_code/modules/observations/obs_kind_mod.f90
```

This created DART/observations/forward_operators/obs_def_mod.f90 from DART/assimilation_code/modules/observations/DEFAULT_obs_kind_mod.F90 and several other modules. DART/assimilation_code/modules/observations/obs_kind_mod.f90 was created similarly. Now we can build the rest of the project.

A series of object files for each module compiled will also be left in the work directory, as some of these are undoubtedly needed by the build of the other DART components. You can proceed to create the other programs needed to work with L63 in DART as follows:

csh mkmf_create_obs_sequence make csh mkmf_create_fixed_network_seq make csh mkmf_perfect_model_obs make csh mkmf_filter make csh mkmf_obs_diag make

The result (hopefully) is that six executables now reside in your work directory. The most common problem is that the NetCDF libraries and include files (particularly typesizes.mod) are not found. Edit the DART/build_templates/mkmf.template, recreate the Makefile, and try again.

program	purpose
preprocess	creates custom source code for just the observation types of interest
create_obs_sequence	specify a (set) of observation characteristics taken by a particular (set of) instruments
create_fixed_network_seq	repeat a set of observations through time to simulate observing networks where observations are taken in the same location at regular (or irregular) intervals
perfect_model_obs	generate "true state" for synthetic observation experiments. Can also be used to 'spin up' a model by running it for a long time.
filter	does the assimilation
obs_diag	creates observation-space diagnostic files to be explored by the Matlab® scripts.
obs_sequence_tool	manipulates observation sequence files. It is not generally needed (particularly for low-order models) but can be used to combine observation sequences or convert from ASCII to binary or vice-versa. We will not cover its use in this document.

6.60.6 Running Lorenz_63

This initial sequence of exercises includes detailed instructions on how to work with the DART code and allows investigation of the basic features of one of the most famous dynamical systems, the 3-variable Lorenz-63 model. The remarkable complexity of this simple model will also be used as a case study to introduce a number of features of a simple ensemble filter data assimilation system. To perform a synthetic observation assimilation experiment for the L63 model, the following steps must be performed (an overview of the process is given first, followed by detailed procedures for each step):

6.60.7 Experiment overview

1. Integrate the L63 model for a long time starting from arbitrary initial conditions to generate a model state that lies on the attractor. The ergodic nature of the L63 system means a 'lengthy' integration always converges to some point on the computer's finite precision representation of the model's attractor.
2. Generate a set of ensemble initial conditions from which to start an assimilation. Since L63 is ergodic, the ensemble members can be designed to look like random samples from the model's 'climatological distribution'. To generate an ensemble member, very small perturbations can be introduced to the state on the attractor generated by step 1. This perturbed state can then be integrated for a very long time until all memory of its initial condition can be viewed as forgotten. Any number of ensemble initial conditions can be generated by repeating this procedure.
3. Simulate a particular observing system by first creating an 'observation set definition' and then creating an 'observation sequence'. The 'observation set definition' describes the instrumental characteristics of the observations and the 'observation sequence' defines the temporal sequence of the observations.
4. Populate the 'observation sequence' with 'perfect' observations by integrating the model and using the information in the 'observation sequence' file to create simulated observations. This entails operating on the model state at the time of the observation with an appropriate forward operator (a function that operates on the model state vector to produce the expected value of the particular observation) and then adding a random sample from the observation error distribution specified in the observation set definition. At the same time, diagnostic output about the 'true' state trajectory can be created.
5. Assimilate the synthetic observations by running the filter; diagnostic output is generated.

1. Integrate the L63 model for a 'long' time

`perfect_model_obs` integrates the model for all the times specified in the 'observation sequence definition' file. To this end, begin by creating an 'observation sequence definition' file that spans a long time. Creating an 'observation sequence definition' file is a two-step procedure involving `create_obs_sequence` followed by `create_fixed_network_seq`. After they are both run, it is necessary to integrate the model with `perfect_model_obs`.

1.1 Create an observation set definition

`create_obs_sequence` creates an observation set definition, the time-independent part of an observation sequence. An observation set definition file only contains the `location, type,` and `observational error characteristics` (normally just the diagonal observational error variance) for a related set of observations. There are no actual observations, nor are there any times associated with the definition. For spin-up, we are only interested in integrating the L63 model, not in generating any particular synthetic observations. Begin by creating a minimal observation set definition.

In general, for the low-order models, only a single observation set need be defined. Next, the number of individual scalar observations (like a single surface pressure observation) in the set is needed. To spin-up an initial condition for the L63 model, only a single observation is needed. Next, the error variance for this observation must be entered. Since we do not need (nor want) this observation to have any impact on an assimilation (it will only be used for spinning up the model and the ensemble), enter a very large value for the error variance. An observation with a very large error variance has essentially no impact on deterministic filter assimilations like the default variety implemented in DART. Finally, the location and type of the observation need to be defined. For all types of models, the most elementary form of synthetic observations are called 'identity' observations. These observations are generated simply by adding a random sample from a specified observational error distribution directly to the value of one of the state variables. This defines the observation as being an identity observation of the first state variable in the L63 model. The program will respond by terminating after generating a file (generally named `set_def.out`) that defines the single identity observation of the first state variable of the L63 model. The following is a screenshot (much of the verbose logging has been left off for clarity), the user input looks *like this*.

```
[unixprompt]$ ./create_obs_sequence
 Starting program create_obs_sequence
 Initializing the utilities module.
 Trying to log to unit   10
 Trying to open file dart_log.out

 Registering module :
 $url: http:/build_templatessquish/DART/trunk/utilities/utilities_mod.f90 $
 $revision: 2713 $
 $date: 2007-03-25 22:09:04 -0600 (Sun, 25 Mar 2007) $
 Registration complete.

 &UTILITIES_NML
 TERMLEVEL= 2,LOGFILENAME=dart_log.out

 /

 Registering module :
 $url: http://squish/DART/trunk/obs_sequence/create_obs_sequence.f90 $
 $revision: 2713 $
 $date: 2007-03-25 22:09:04 -0600 (Sun, 25 Mar 2007) $
 Registration complete.

 { ... }

 Input upper bound on number of observations in sequence
10

 Input number of copies of data (0 for just a definition)
0

 Input number of quality control values per field (0 or greater)
0

 input a -1 if there are no more obs
0

 Registering module :
 $url: http://squish/DART/trunk/obs_def/DEFAULT_obs_def_mod.F90 $
 $revision: 2820 $
 $date: 2007-04-09 10:37:47 -0600 (Mon, 09 Apr 2007) $
 Registration complete.
```

(continues on next page)

(continued from previous page)

```
 Registering module :
 $url: http://squish/DART/trunk/obs_kind/DEFAULT_obs_kind_mod.F90 $
 $revision: 2822 $
 $date: 2007-04-09 10:39:08 -0600 (Mon, 09 Apr 2007) $
 Registration complete.

 ------------------------------------------------------

 initialize_module obs_kind_nml values are

 -------------- ASSIMILATE_THESE_OBS_TYPES --------------
 RAW_STATE_VARIABLE
 -------------- EVALUATE_THESE_OBS_TYPES --------------
 ------------------------------------------------------

      Input -1 * state variable index for identity observations
      OR input the name of the observation kind from table below:
      OR input the integer index, BUT see documentation...
        1 RAW_STATE_VARIABLE

-1

 input time in days and seconds
1 0

 Input error variance for this observation definition
1000000

 input a -1 if there are no more obs
-1

 Input filename for sequence (  set_def.out   usually works well)
 set_def.out
 write_obs_seq  opening formatted file set_def.out
 write_obs_seq  closed file set_def.out
```

1.2 Create an observation sequence definition

`create_fixed_network_seq` creates an 'observation sequence definition' by extending the 'observation set definition' with the temporal attributes of the observations.

The first input is the name of the file created in the previous step, i.e. the name of the observation set definition that you've just created. It is possible to create sequences in which the observation sets are observed at regular intervals or irregularly in time. Here, all we need is a sequence that takes observations over a long period of time - indicated by entering a 1. Although the L63 system normally is defined as having a non-dimensional time step, the DART system arbitrarily defines the model timestep as being 3600 seconds. If we declare that we have one observation per day for 1000 days, we create an observation sequence definition spanning 24000 'model' timesteps; sufficient to spin-up the model onto the attractor. Finally, enter a name for the 'observation sequence definition' file. Note again: there are no observation values present in this file. Just an observation type, location, time and the error characteristics. We are going to populate the observation sequence with the `perfect_model_obs` program.

```
[unixprompt]$ ./create_fixed_network_seq

 ...

 Registering module :
 $url: http://squish/DART/trunk/obs_sequence/obs_sequence_mod.f90 $
 $revision: 2749 $
 $date: 2007-03-30 15:07:33 -0600 (Fri, 30 Mar 2007) $
 Registration complete.

 static_init_obs_sequence obs_sequence_nml values are
 &OBS_SEQUENCE_NML
 WRITE_BINARY_OBS_SEQUENCE =  F,
 /
 Input filename for network definition sequence (usually  set_def.out  )
set_def.out

 ...

 To input a regularly repeating time sequence enter 1
 To enter an irregular list of times enter 2
1
 Input number of observations in sequence
1000
 Input time of initial ob in sequence in days and seconds
1, 0
 Input period of obs in days and seconds
1, 0
           1
           2
           3
...
         997
         998
         999
        1000
What is output file name for sequence (  obs_seq.in   is recommended )
obs_seq.in
 write_obs_seq  opening formatted file obs_seq.in
 write_obs_seq closed file obs_seq.in
```

1.3 Initialize the model onto the attractor

`perfect_model_obs` can now advance the arbitrary initial state for 24,000 timesteps to move it onto the attractor. `perfect_model_obs` uses the Fortran90 namelist input mechanism instead of (admittedly gory, but temporary) interactive input. All of the DART software expects the namelists to found in a file called `input.nml`. When you built the executable, an example namelist was created `input.nml.perfect_model_obs_default` that contains all of the namelist input for the executable. If you followed the example, each namelist was saved to a unique name. We must now rename and edit the namelist file for `perfect_model_obs`. Copy `input.nml.perfect_model_obs_default` to `input.nml` and edit it to look like the following: (just worry about the highlighted stuff - and whitespace doesn't matter)

cp input.nml.perfect_model_obs_default input.nml

```
&perfect_model_obs_nml
   start_from_restart    = .false.,
   output_restart        = .true.,
   async                 = 0,
   init_time_days        = 0,
   init_time_seconds     = 0,
   first_obs_days        = -1,
   first_obs_seconds     = -1,
   last_obs_days         = -1,
   last_obs_seconds      = -1,
   output_interval       = 1,
   restart_in_file_name  = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   obs_seq_in_file_name  = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   adv_ens_command       = "./advance_ens.csh"   /

&ensemble_manager_nml
   single_restart_file_in  = .true.,
   single_restart_file_out = .true.,
   perturbation_amplitude  = 0.2   /

&assim_tools_nml
   filter_kind                     = 1,
   cutoff                          = 0.2,
   sort_obs_inc                    = .false.,
   spread_restoration              = .false.,
   sampling_error_correction       = .false.,
   adaptive_localization_threshold = -1,
   print_every_nth_obs             = 0   /

&cov_cutoff_nml
   select_localization = 1   /

&reg_factor_nml
   select_regression    = 1,
   input_reg_file       = "time_mean_reg",
   save_reg_diagnostics = .false.,
   reg_diagnostics_file = "reg_diagnostics"   /

&obs_sequence_nml
   write_binary_obs_sequence = .false.   /

&obs_kind_nml
   assimilate_these_obs_types = 'RAW_STATE_VARIABLE'   /

&assim_model_nml
   write_binary_restart_files = .true. /

&model_nml
   sigma  = 10.0,
   r      = 28.0,
   b      = 2.6666666666667,
   deltat = 0.01,
   time_step_days = 0,
   time_step_seconds = 3600   /
```

(continues on next page)

(continued from previous page)

```
&utilities_nml
   TERMLEVEL = 1,
   logfilename = 'dart_log.out'  /
```

For the moment, only two namelists warrant explanation. Each namelists is covered in detail in the html files accompanying the source code for the module.

perfect_model_obs_nml

namelist variable	description
start_from_restart	When set to 'false', perfect_model_obs generates an arbitrary initial condition (which cannot be guaranteed to be on the L63 attractor). When set to 'true', a restart file (specified by restart_in_file_name) is read.
output_restart	When set to 'true', perfect_model_obs will record the model state at the end of this integration in the file named by restart_out_file_name.
async	The lorenz_63 model is advanced through a subroutine call - indicated by async = 0. There is no other valid value for this model.
init_time_*xxx*	the start time of the integration.
first_obs_*xxx*	the time of the first observation of interest. While not needed in this example, you can skip observations if you want to. A value of -1 indicates to start at the beginning.
last_obs_*xxx*	the time of the last observation of interest. While not needed in this example, you do not have to assimilate all the way to the end of the observation sequence file. A value of -1 indicates to use all the observations.
output_interval	interval at which to save the model state (in True_State.nc).
restart_in_file_name	is ignored when 'start_from_restart' is 'false'.
restart_out_file_name	if output_restart is 'true', this specifies the name of the file containing the model state at the end of the integration.
obs_seq_in_file_name	specifies the file name that results from running create_fixed_network_seq, i.e. the 'observation sequence definition' file.
obs_seq_out_file_name	specifies the output file name containing the 'observation sequence', finally populated with (perfect?) 'observations'.
advance_ens_command	specifies the shell commands or script to execute when async /= 0.

utilities_nml

namelist variable	description
TERMLEVEL	When set to '1' the programs terminate when a 'warning' is generated. When set to '2' the programs terminate only with 'fatal' errors.
logfilename	Run-time diagnostics are saved to this file. This namelist is used by all programs, so the file is opened in APPEND mode. Subsequent executions cause this file to grow.

Executing perfect_model_obs will integrate the model 24,000 steps and output the resulting state in the file perfect_restart. Interested parties can check the spinup in the True_State.nc file.

./perfect_model_obs

2. Generate a set of ensemble initial conditions

The set of initial conditions for a 'perfect model' experiment is created in several steps. 1) Starting from the spun-up state of the model (available in `perfect_restart`), run `perfect_model_obs` to generate the 'true state' of the experiment and a corresponding set of observations. 2) Feed the same initial spun-up state and resulting observations into `filter`.
The first step is achieved by changing a perfect_model_obs namelist parameter, copying `perfect_restart` to `perfect_ics`, and rerunning `perfect_model_obs`. This execution of `perfect_model_obs` will advance the model state from the end of the first 24,000 steps to the end of an additional 24,000 steps and place the final state in `perfect_restart`. The rest of the namelists in `input.nml` should remain unchanged.

```
&perfect_model_obs_nml
   start_from_restart    = .true.,
   output_restart        = .true.,
   async                 = 0,
   init_time_days        = 0,
   init_time_seconds     = 0,
   first_obs_days        = -1,
   first_obs_seconds     = -1,
   last_obs_days         = -1,
   last_obs_seconds      = -1,
   output_interval       = 1,
   restart_in_file_name  = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   obs_seq_in_file_name  = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   adv_ens_command       = "./advance_ens.csh"   /
```

cp perfect_restart perfect_ics ./perfect_model_obs

A `True_State.nc` file is also created. It contains the 'true' state of the integration.

Generating the ensemble

This step (#2 from above) is done with the program `filter`, which also uses the Fortran90 namelist mechanism for input. It is now necessary to copy the `input.nml.filter_default` namelist to `input.nml`.

cp input.nml.filter_default input.nml

You may also build one master namelist containting all the required namelists. Having unused namelists in the `input.nml` does not hurt anything, and it has been so useful to be reminded of what is possible that we made it an error to NOT have a required namelist. Take a peek at any of the other models for examples of a "fully qualified" `input.nml`.
HINT: if you used `svn` to get the project, try 'svn revert input.nml' to restore the namelist that was distributed with the project - which DOES have all the namelist blocks. Just be sure the values match the examples here.

```
&filter_nml
   async                    = 0,
   adv_ens_command          = "./advance_model.csh",
   ens_size                 = 100,
   start_from_restart       = .false.,
   output_restart           = .true.,
   obs_sequence_in_name     = "obs_seq.out",
   obs_sequence_out_name    = "obs_seq.final",
   restart_in_file_name     = "perfect_ics",
   restart_out_file_name    = "filter_restart",
   init_time_days           = 0,
   init_time_seconds        = 0,
   first_obs_days           = -1,
   first_obs_seconds        = -1,
   last_obs_days            = -1,
   last_obs_seconds         = -1,
   num_output_state_members = 20,
   num_output_obs_members   = 20,
   output_interval          = 1,
   num_groups               = 1,
   input_qc_threshold       =  4.0,
   outlier_threshold        = -1.0,
   output_forward_op_errors = .false.,
   output_timestamps        = .false.,
   output_inflation         = .true.,

   inf_flavor               = 0,                       0,
   inf_start_from_restart   = .false.,                 .false.,
   inf_output_restart       = .false.,                 .false.,
   inf_deterministic        = .true.,                  .true.,
   inf_in_file_name         = 'not_initialized',       'not_initialized',
   inf_out_file_name        = 'not_initialized',       'not_initialized',
   inf_diag_file_name       = 'not_initialized',       'not_initialized',
   inf_initial              = 1.0,                     1.0,
   inf_sd_initial           = 0.0,                     0.0,
   inf_lower_bound          = 1.0,                     1.0,
   inf_upper_bound          = 1000000.0,               1000000.0,
   inf_sd_lower_bound       = 0.0,                     0.0
/

&smoother_nml
   num_lags              = 0,
   start_from_restart    = .false.,
   output_restart        = .false.,
   restart_in_file_name  = 'smoother_ics',
   restart_out_file_name = 'smoother_restart'  /

&ensemble_manager_nml
   single_restart_file_in  = .true.,
   single_restart_file_out = .true.,
   perturbation_amplitude  = 0.2  /

&assim_tools_nml
   filter_kind                     = 1,
   cutoff                          = 0.2,
   sort_obs_inc                    = .false.,
   spread_restoration              = .false.,
```

(continues on next page)

(continued from previous page)

```
   sampling_error_correction       = .false.,
   adaptive_localization_threshold = -1,
   print_every_nth_obs             = 0   /

&cov_cutoff_nml
   select_localization = 1  /

&reg_factor_nml
   select_regression    = 1,
   input_reg_file       = "time_mean_reg",
   save_reg_diagnostics = .false.,
   reg_diagnostics_file = "reg_diagnostics"  /

&obs_sequence_nml
   write_binary_obs_sequence = .false.  /

&obs_kind_nml
   assimilate_these_obs_types = 'RAW_STATE_VARIABLE'   /

&assim_model_nml
   write_binary_restart_files = .true. /

&model_nml
   sigma  = 10.0,
   r      = 28.0,
   b      = 2.6666666666667,
   deltat = 0.01,
   time_step_days = 0,
   time_step_seconds = 3600  /

&utilities_nml
   TERMLEVEL = 1,
   logfilename = 'dart_log.out'  /
```

Only the non-obvious(?) entries for `filter_nml` will be discussed.

namelist variable	description
`ens_size`	Number of ensemble members. 100 is sufficient for most of the L63 exercises.
`start_from_restart`	when ‘.false.’, `filter` will generate its own ensemble of initial conditions. It is important to note that the filter still makes use of the file named by `restart_in_file_name` (i.e. `perfect_ics`) by randomly perturbing these state variables.
`num_output_state_members`	specifies the number of state vectors contained in the NetCDF diagnostic files. May be a value from 0 to `ens_size`.
`num_output_obs_members`	specifies the number of ‘observations’ (derived from applying the forward operator to the state vector) are contained in the `obs_seq.final` file. May be a value from 0 to `ens_size`
`inf_flavor`	A value of 0 results in no inflation.(spin-up)

The filter is told to generate its own ensemble initial conditions since `start_from_restart` is ‘.false.’. However, it is important to note that the filter still makes use of `perfect_ics` which is set to be the `restart_in_file_name`. This is the model state generated from the first 24,000 step model integration by `perfect_model_obs`. `Filter` generates its ensemble initial conditions by randomly perturbing the state variables of this state.

`num_output_state_members` are ‘.true.’ so the state vector is output at every time for which there are observations (once a day here). `Posterior_Diag.nc` and `Prior_Diag.nc` then contain values for 20 ensemble members once a day. Once the namelist is set, execute `filter` to integrate the ensemble forward for 24,000 steps with the final ensemble state written to the `filter_restart`. Copy the `perfect_model_obs` restart file `perfect_restart` (the `true state') to `perfect_ics`, and the `filter` restart file `filter_restart` to `filter_ics` so that future assimilation experiments can be initialized from these spun-up states.

```
./filter
cp perfect_restart perfect_ics
cp filter_restart filter_ics
```

The spin-up of the ensemble can be viewed by examining the output in the NetCDF files `True_State.nc` generated by `perfect_model_obs` and `Posterior_Diag.nc` and `Prior_Diag.nc` generated by `filter`. To do this, see the detailed discussion of matlab diagnostics in Appendix I.

3. Simulate a particular observing system

Begin by using `create_obs_sequence` to generate an observation set in which each of the 3 state variables of L63 is observed with an observational error variance of 1.0 for each observation. To do this, use the following input sequence (the text including and after # is a comment and does not need to be entered):

4	# upper bound on num of observations in sequence
0	# number of copies of data (0 for just a definition)
0	# number of quality control values per field (0 or greater)
0	# -1 to exit/end observation definitions
-1	# observe state variable 1
0 0	# time – days, seconds
1.0	# observational variance
0	# -1 to exit/end observation definitions
-2	# observe state variable 2
0 0	# time – days, seconds
1.0	# observational variance
0	# -1 to exit/end observation definitions
-3	# observe state variable 3
0 0	# time – days, seconds
1.0	# observational variance
-1	# -1 to exit/end observation definitions
set_def.out	# Output file name

Now, generate an observation sequence definition by running `create_fixed_network_seq` with the following input sequence:

set_def.out	# Input observation set definition file
1	# Regular spaced observation interval in time
1000	# 1000 observation times
0, 43200	# First observation after 12 hours (0 days, 12 * 3600 seconds)
0, 43200	# Observations every 12 hours
obs_seq.in	# Output file for observation sequence definition

4. Generate a particular observing system and true state

An observation sequence file is now generated by running `perfect_model_obs` with the namelist values (unchanged from step 2):

```
&perfect_model_obs_nml
   start_from_restart    = .true.,
   output_restart        = .true.,
   async                 = 0,
   init_time_days        = 0,
   init_time_seconds     = 0,
   first_obs_days        = -1,
   first_obs_seconds     = -1,
   last_obs_days         = -1,
   last_obs_seconds      = -1,
   output_interval       = 1,
   restart_in_file_name  = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   obs_seq_in_file_name  = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   adv_ens_command       = "./advance_ens.csh"   /
```

This integrates the model starting from the state in `perfect_ics` for 1000 12-hour intervals outputting synthetic observations of the three state variables every 12 hours and producing a NetCDF diagnostic file, `True_State.nc`.

5. Filtering

Finally, `filter` can be run with its namelist set to:

```
&filter_nml
   async                    = 0,
   adv_ens_command          = "./advance_model.csh",
   ens_size                 = 100,
   start_from_restart       = .true.,
   output_restart           = .true.,
   obs_sequence_in_name     = "obs_seq.out",
   obs_sequence_out_name    = "obs_seq.final",
   restart_in_file_name     = "filter_ics",
   restart_out_file_name    = "filter_restart",
   init_time_days           = 0,
   init_time_seconds        = 0,
   first_obs_days           = -1,
   first_obs_seconds        = -1,
   last_obs_days            = -1,
   last_obs_seconds         = -1,
   num_output_state_members = 20,
   num_output_obs_members   = 20,
   output_interval          = 1,
   num_groups               = 1,
   input_qc_threshold       =  4.0,
   outlier_threshold        = -1.0,
   output_forward_op_errors = .false.,
   output_timestamps        = .false.,
   output_inflation         = .true.,

   inf_flavor               = 0,                       0,
```

(continues on next page)

(continued from previous page)

```
  inf_start_from_restart    = .false.,                 .false.,
  inf_output_restart        = .false.,                 .false.,
  inf_deterministic         = .true.,                  .true.,
  inf_in_file_name          = 'not_initialized',       'not_initialized',
  inf_out_file_name         = 'not_initialized',       'not_initialized',
  inf_diag_file_name        = 'not_initialized',       'not_initialized',
  inf_initial               = 1.0,                     1.0,
  inf_sd_initial            = 0.0,                     0.0,
  inf_lower_bound           = 1.0,                     1.0,
  inf_upper_bound           = 1000000.0,               1000000.0,
  inf_sd_lower_bound        = 0.0,                     0.0
/
```

filter produces two output diagnostic files, Prior_Diag.nc which contains values of the ensemble mean, ensemble spread, and ensemble members for 12- hour lead forecasts before assimilation is applied and Posterior_Diag.nc which contains similar data for after the assimilation is applied (sometimes referred to as analysis values).

Now try applying all of the matlab diagnostic functions described in the Matlab® Diagnostics section.

6.60.8 The tutorial

The DART/tutorial documents are an excellent way to kick the tires on DART and learn about ensemble data assimilation. If you have gotten this far, you can run anything in the tutorial.

6.60.9 Matlab® diagnostics

The output files are NetCDF files and may be examined with many different software packages. We use Matlab®, and provide our diagnostic scripts in the hopes that they are useful.

The diagnostic scripts and underlying functions reside in two places: DART/diagnostics/matlab and DART/matlab. They are reliant on the public-domain MEXNC/SNCTOOLS NetCDF interface from http://mexcdf.sourceforge.net. If you do not have them installed on your system and want to use Matlab to peruse NetCDF, you must follow their installation instructions. The 'interested reader' may want to look at the DART/matlab/startup.m file I use on my system. If you put it in your $HOME/matlab directory it is invoked every time you start up Matlab.

Once you can access the nc_varget function from within Matlab you can use our diagnostic scripts. It is necessary to prepend the location of the DART/matlab scripts to the matlabpath. Keep in mind the location of the Netcdf operators on your system WILL be different from ours … and that's OK.

```
[models/lorenz_63/work]$ matlab -nodesktop

                                             < M A T L A B >
                                 Copyright 1984-2002 The MathWorks, Inc.
                                     Version 6.5.0.180913a Release 13
                                               Jun 18 2002

  Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.

  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.
```

(continues on next page)

(continued from previous page)

```
>> which nc_varget
/contrib/matlab/snctools/4024/nc_varget.m
>>ls *.nc

ans =

Posterior_Diag.nc  Prior_Diag.nc  True_State.nc


>>path('../../../matlab',path)
>>path('../../../diagnostics/matlab',path)
>>which plot_ens_err_spread
../../../matlab/plot_ens_err_spread.m
>>help plot_ens_err_spread

  DART : Plots summary plots of the ensemble error and ensemble spread.
                         Interactively queries for the needed information.
                         Since different models potentially need different
                         pieces of information ... the model types are
                         determined and additional user input may be queried.

  Ultimately, plot_ens_err_spread will be replaced by a GUI.
  All the heavy lifting is done by PlotEnsErrSpread.

  Example 1 (for low-order models)

  truth_file = 'True_State.nc';
  diagn_file = 'Prior_Diag.nc';
  plot_ens_err_spread

>>plot_ens_err_spread
```

And the matlab graphics window will display the spread of the ensemble error for each state variable. The scripts are designed to do the “obvious” thing for the low-order models and will prompt for additional information if needed. The philosophy of these is that anything that starts with a lower-case *plot_some_specific_task* is intended to be user-callable and should handle any of the models. All the other routines in `DART/matlab` are called BY the high-level routines.

Matlab script	description
`plot_bins`	plots ensemble rank histograms
`plot_correl`	Plots space-time series of correlation between a given variable at a given time and other variables at all times in a n ensemble time sequence.
`plot_ens_err_spread`	Plots summary plots of the ensemble error and ensemble spread. Interactively queries for the needed information. Since different models potentially need different pieces of information . . . the model types are determined and additional user input may be queried.
`plot_ens_mean_time_series`	Queries for the state variables to plot.
`plot_ens_time_series`	Queries for the state variables to plot.
`plot_phase_space`	Plots a 3D trajectory of (3 state variables of) a single ensemble member. Additional trajectories may be superimposed.
`plot_total_err`	Summary plots of global error and spread.
`plot_var_var_correl`	Plots time series of correlation between a given variable at a given time and another variable at all times in an ensemble time sequence.

6.60.10 Bias, filter divergence and covariance inflation (with the l63 model)

One of the common problems with ensemble filters is filter divergence, which can also be an issue with a variety of other flavors of filters including the classical Kalman filter. In filter divergence, the prior estimate of the model state becomes too confident, either by chance or because of errors in the forecast model, the observational error characteristics, or approximations in the filter itself. If the filter is inappropriately confident that its prior estimate is correct, it will then tend to give less weight to observations than they should be given. The result can be enhanced overconfidence in the model's state estimate. In severe cases, this can spiral out of control and the ensemble can wander entirely away from the truth, confident that it is correct in its estimate. In less severe cases, the ensemble estimates may not diverge entirely from the truth but may still be too confident in their estimate. The result is that the truth ends up being farther away from the filter estimates than the spread of the filter ensemble would estimate. This type of behavior is commonly detected using rank histograms (also known as Talagrand diagrams). You can see the rank histograms for the L63 initial assimilation by using the matlab script `plot_bins`.

A simple, but surprisingly effective way of dealing with filter divergence is known as covariance inflation. In this method, the prior ensemble estimate of the state is expanded around its mean by a constant factor, effectively increasing the prior estimate of uncertainty while leaving the prior mean estimate unchanged. The program `filter` has a group of namelist parameters that controls the application of covariance inflation. For a simple set of inflation values, you will set `inf_flavor`, and `inf_initial`. These values come in pairs; the first value controls inflation of the prior ensemble values, while the second controls inflation of the posterior values. Up to this point `inf_flavor` has been set to 0 indicating that the prior ensemble is left unchanged. Setting the first value of `inf_flavor` to 3 enables one variety of inflation. Set `inf_initial` to different values (try 1.05 and 1.10 and other values). In each case, use the diagnostic matlab tools to examine the resulting changes to the error, the ensemble spread (via rank histogram bins, too), etc. What kind of relation between spread and error is seen in this model?

There are many more options for inflation, including spatially and temporarily varying values, with and without damping. See the discussion of all inflation-related namelist items local file.

6.60.11 Synthetic observations

Synthetic observations are generated from a `perfect' model integration, which is often referred to as the `truth' or a `nature run'. A model is integrated forward from some set of initial conditions and observations are generated as $y = H(x) + e$ where H is an operator on the model state vector, x, that gives the expected value of a set of observations, y, and e is a random variable with a distribution describing the error characteristics of the observing instrument(s) being simulated. Using synthetic observations in this way allows students to learn about assimilation algorithms while being isolated from the additional (extreme) complexity associated with model error and unknown observational error characteristics. In other words, for the real-world assimilation problem, the model has (often substantial) differences from what happens in the real system and the observational error distribution may be very complicated and is certainly not well known. Be careful to keep these issues in mind while exploring the capabilities of the ensemble filters with synthetic observations.

6.61 RMA notes

In the RMA version of DART, the state vector is not required to be stored completely on any process. This is achieved using Remote Memory Access (RMA). The RMA programing model allows processes to read (and write) memory on other processors asynchronously. RMA DART supported models:

Before bitwise testing with Lanai please read *Bitwise Considerations*

6.61.1 NetCDF restarts

The programs filter and perfect_model_obs now read/write directly from NetCDF files, rather than having to run converters (`model_to_dart` and `dart_to_model`). To facilitate this, there is a new required call `add_domain` which must be called during `static_init_model`. It can be called multiple times in static_model_mod, e.g. once for each NetCDF file that contains state variables. There are three ways to add a domain:

- **From Blank** : This is for small models such as lorenz_96 and no NetCDF restarts
  - `dom_id = add_domain(model_size)`
- **From File** : This is for models which have NetCDF restart files
  - `dom_id = add_domain(template_file, num_vars, var_names, ... )`
- **From Spec** : Creates a skeleton structure for a domain (currently only used in bgrid_solo)
  - `dom_id = add_domain(num_vars, var_names, ... )` `call add_dimension_to_variable(dom_id, var_id, dim_nam, dim_size)` `call finished_adding_domain`

For models without NetCDF restarts, use `add_domain(model_size)`. This is the minimum amount of information needed by DART to create a netdcf file. For models with NetCDF restarts use `add_domain(info_file, num_vars, var_names)` which lets DART read the NetCDF dimensions for a list of variables from a file (`info_file`). There are several routines that can be used together to create a domain from a description: `add_domain, add_dimension_to_variable, finished_adding_domain`. This can be used in models such as bgrid_solo where the model is spun up in perfect_model_obs, but the model itself has variable structure (3D variables with names). See Additions/Changes to existing namelists for how to use NetCDF IO.

Note when using NetCDF restarts, inflation files are NetCDF also. The inflation mean and inflation standard deviation are in separate files when you use NetCDF restarts. See *Netcdf Inflation Files* for details.

6.61.2 Calculation of forward operators

The forward operator code in model_mod now operates on an array of state values. See *Forward Operator* for more detail about distributed vs. non-distributed forward operators.

In distributed mode the forward operators for all ensemble members are calculated in the same `model_interpolate` call. In non-distributed mode, the forward oeprators for all ensemble members a task owns (1-ens_size) are calculated at once.

6.61.3 Vertical conversion of observation locations

The vertical conversion of observation locations is done before the assimilation. In Lanai this calculation is done in the assimilation as part of `get_close_obs` if a model_mod does vertical conversion. See *Vertical Conversion of Observations* for details about this change. Note that not all models do vertical conversion or even have a concept of vertical location, but every model_mod must have the following routines:

- `query_vert_localization_coord` - returns the vertical localization coordiate (or does nothing if there is no vertical conversion)
- `vert_convert` - converts location to required vertical (or does nothing if there is no vertical conversion)

6.61.4 Diagnostic file changes

For large models DART format diagnostic files (Prior_Diag.nc and Posterior_Diag.nc) have been replaced with separate files for each copy that would have gone into Prior_Diag.nc and Posterior_Diag.nc.

For Prior_Diag.nc:

- **Mean and standard deviation**: preassim_mean.nc preassim_sd.nc
- **Inflation mean and standard deviation** (if state space inflation is used): preassim_priorinf_mean.nc preassim_priorinf_sd.nc
- **The number of ensemble members specifed** in filter_nml (num_output_state_members): preassim_member_####.nc

For Posterior_Diag.nc:

- **Mean and standard deviation**: postassim_mean.nc postassim_sd.nc
- **Inflation mean and standard deviation** (if state space inflation is used): postassim_priorinf_mean.nc postassim_priorinf_sd.nc
- **The number of ensemble members specifed** in filter_nml (num_output_state_members): postassim_member_####.nc

The `num_output_state_members` are not written separately from the restarts. Note that restarts will have been clamped if any clamping is applied (given as an arguement to add_domain). This is *different* to Posterior_Diag.nc which contains unclamped values. Note also that there are 2 more "stages" which might be output, in addition to the preassim and postassim discussed here.

For models with multiple domains the filenames above are appended with the domain number, e.g. preassim_mean.nc becomes preassim_mean_d01.nc, preassim_mean_d02.nc, etc.

Changes to nc_write_model_atts

`nc_write_model_atts` has an new argument '`model_mod_writes_state_variables`'. This is used to communicate to DART whether the model will create and write state variables in Prior_Diag.nc and Posterior_Diag.nc. If `model_model_writes_state_variables = .false.` DART will define and write state variables to the new diagnostic files. If `model_model_writes_state_variables = .true.`, `nc_write_model_vars` is called as normal.

6.61.5 Perturbations

The option to perturb one ensemble member to produce an ensemble is in filter_nml:`perturb_from_single_instance`. The model_mod interface is now `pert_model_copies` not `pert_model_state`. Each task perturbs every ensemble member for its own subsection of state. This is more complicated than the Lanai routine `pert_model_state`, where a whole state vector is available. If a model_mod does not provide a perturb interface, filter will do the perturbing with an amplitude set in filter_nml:perturbation_amplitude. Note the perturb namelist options have been removed from ensemble_manager_nml

6.61.6 State_vector_io_nml

```
&state_vector_io_nml
   buffer_state_io         = .false.,
   single_precision_output = .false.,
/
```

When `buffer_state_io` is `.false.` the entire state is read into memory at once if .true. variables are read one at a time. If your model can not fit into memory at once this must be set to `.true.` .

`single_precision_output` allows you to run filter in double precision but write NetCDF files in single precision.

6.61.7 Quality_control_nml

These namelist options used to be in filter_nml, now they are in quality_control_nml.

```
&quality_control_nml
   input_qc_threshold          = 3,
   outlier_threshold           = 4,
   enable_special_outlier_code = .false.
/
```

6.61.8 Additions/changes to existing namelists

New namelist variables

filter_nml

```
&filter_nml
   single_file_in               = .false.,
   single_file_out              = .false.,

   input_state_file_list        = 'null',
   output_state_file_list       = 'null',
   input_state_files            = 'null',
   output_state_files           = 'null',

   stages_to_write              = 'output'
   write_all_stages_at_end      = .false.
   output_restarts              = .true.
   output_mean                  = .true.
   output_sd                    = .true.

   perturb_from_single_instance = .false.,
   perturbation_amplitude       = 0.2_r8,

   distributed_state            = .true.
/
```

Item	Type	Description
single_file_in	logical	True means that all of the restart and inflation information is read from a single NetCDF file. False means that you must specify an input_state_file_list and DART will be expecting input_{priorinf,postinf}_{mean,sd}.nc files for inflation.
single_file_out	logical	True means that all of the restart and inflation information is written to a single NetCDF file. False means that you must specify a output_state_file_list and DART will be output files specified in the list. Inflation files will be written in the form input_{priorinf,postinf}_{mean,sd}.nc.
input_restart_files	character array	This is used for single file input for low order models. For multiple domains you can specify a file for each domain. When specifying a list single_file_in, single_file_out must be set to .true.
output_restart_files	character array	This is used for single file input for low order models. For multiple domains you can specify a file for each domain. When specifying a list single_file_in, single_file_out must be set to .true.
input_state_file_list	character array	A list of files containing input model restarts. For multiple domains you can specify a file for each domain. When specifying a list single_file_in, single_file_out must be set to .false.
output_state_file_list	character array	A list of files containing output model restarts. For multiple domains you can specify a file for each domain. When specifying a list single_file_in, single_file_out must be set to .false.
stages_to_write	character array	Controls which stages to write. Currently there are four options: • `input` – writes input mean and sd only • `preassim` – before assimilation, before prior inflation is applied • `postassim` – after assimilation, before posterior inflation is applied • `output` – final output for filter which includes clamping and inflation
write_all_stages_at_end	logical	True means output all stages at the end of filter. This is more memory intensive but requires less

For NetCDF reads and writes

For **input** file names:

- give input_state_file_list a file for each domain, each of which contains a list of restart files.
- if no input_state_file_list is provided then default filenames will be used e.g. input_member_000*.nc, input_priorinf_mean.nc, input_priorinf_sd.nc

For **output** file names:

- give output_state_file_list a file for each domain, each of which contains a list of restart files.
- if no output_state_file_list is provided then default filenames will be used e.g. output_member_000*.nc, output_priorinf_mean.nc, output_priorinf_sd.nc

For small models you may want to use single_file_in, single_file_out which contains all copies needed to run filter.

Assim_tools_nml

```
&assim_tools_nml
   distribute_mean  = .true.
/
```

In previous DART releases, each processor gets a copy of the mean (in ens_mean_for_model). In RMA DART, the mean is distributed across all processors. However, a user can choose to have a copy of the mean on each processor by setting distribute_mean = .false. . Note that the mean state is accessed through get_state whether distribute_mean is .true. or .false.

Removed from existing namelists

```
&filter_nml
   input_qc_threshold          = 3,
   outlier_threshold           = 4,
   enable_special_outlier_code = .false.
   start_from_restart          = .false.
   output_inflation            = .true.
   output_restart              = .true.
   /
```

NOTE : output_restart has been renamed to output_restarts. **``output_inflation`` is no longer supported** and only writes inflation files if inf_flavor > 1

```
&ensemble_manager_nml
   single_restart_file_out = .true.
   perturbation_amplitude  = 0.2,
   /
```

```
&assim_manager_nml
   write_binary_restart_files = .true.,
   netCDF_large_file_support  = .false.
   /
```

6.62 DART Manhattan Differences from Lanai Release Notes

6.62.1 Overview

This document includes an overview of the changes in the DART system since the Lanai release. For further details on any of these items look at the HTML documentation for that specific part of the system.

The two most significant changes in the Manhattan version of DART are it can support running models with a state vector larger than the memory of a single task, removing a limit from the Lanai version of DART. It also reads and writes NetCDF files directly instead of requiring a conversion from one file to another. There are many other smaller changes, detailed below.

Manhattan supported models:

- 9var
- bgrid_solo
- cam-fv
- cice
- clm
- cm1
- forced_lorenz_96
- ikeda
- lorenz_63
- lorenz_84
- lorenz_96
- lorenz_96_2scale
- lorenz_04
- mpas_atm (NetCDF overwrite not supported for update_u_from_reconstruct = .true.)
- null_model
- POP
- ROMS
- simple_advection
- wrf

If your model of interest is not on the list consider checking out the ‘Classic’ release of DART, which is Lanai plus bug fixes and minor enhancements. All models previously supported by Lanai are still in DART ‘Classic’.

These are the major differences between the Lanai/Classic and Manhattan releases of DART:

- Read and write NetCDF restarts
- Calculation of forward operators
- Vertical conversion of observation locations
- Diagnostic file changes
- *State Stucture*

- model_mod interface changes
- Observation Quantity replaces Kind
- Perturbation of the state

6.62.2 NetCDF restart files

The programs filter and perfect_model_obs now read/write directly from NetCDF files rather than having to run converters (`model_to_dart` and `dart_to_model`). To facilitate this there is a new required call `add_domain` which must be called during `static_init_model`. It can be called multiple times in static_model_mod, e.g. once for each NetCDF file that contains state variables. There are three ways to add a domain:

- **From Blank** : This is for small models such as lorenz_96 and no NetCDF restarts
  - `dom_id = add_domain(model_size)`
- **From File** : This is for models which have NetCDF restart files
  - `dom_id = add_domain(template_file, num_vars, var_names, ... )`
- **From Spec** : Creates a skeleton structure for a domain (currently only used in bgrid_solo)
  - `dom_id = add_domain(num_vars, var_names, ... ) call add_dimension_to_variable(dom_id, var_id, dim_nam, dim_size) call finished_adding_domain`

For models without NetCDF restarts, use `add_domain(model_size)`. This is the minimum amount of information needed by DART to create a netdcf file. For models with NetCDF restarts use `add_domain(info_file, num_vars, var_names)` which lets DART read the NetCDF dimensions for a list of variables from a file (`info_file`). There are several routines that can be used together to create a domain from a description: `add_domain, add_dimension_to_variable, finished_adding_domain`. This can be used in models such as bgrid_solo where the model is spun up in perfect_model_obs, but the model itself has variable structure (3D variables with names). See Additions/Changes to existing namelists for how to use NetCDF IO.

Note when using NetCDF restarts, inflation files are NetCDF also. The inflation mean and inflation standard deviation are in separate files when you use NetCDF restarts. See *Netcdf Inflation Files* for details.

6.62.3 Calculation of forward operators

The forward operator code in model_mod now operates on an array of state values. See *Forward Operator* for more detail about distributed vs. non-distributed forward operators. In distributed mode the forward operators for all ensemble members are calculated in the same `model_interpolate` call. In non-distributed mode, the forward operators for all ensemble members a task owns (1-ens_size) are calculated at once.

6.62.4 Vertical conversion of observation and state locations

The vertical conversion of observation locations is done before the assimilation by default. This can be changed by namelist options.

In Lanai this calculation is done in the assimilation as part of `get_close_obs` if a model_mod does vertical conversion. See *Vertical Conversion of Observations* for details about this change. Note that not all models do vertical conversion or even have a concept of vertical location, but every model_mod must have the following routines:

```
call set_vertical_localization_coord(vert_localization_coord)

call convert_vertical_obs(ens_handle, num, locs, loc_qtys, loc_types, &
                          which_vert, status)

call convert_vertical_state(ens_handle, num, locs, loc_qtys, loc_indx, &
                            which_vert, istatus)
```

If there are NOT multiple choices for a vertical coordinate (e.g. cartesian, one dimensional), all these routines can be no-ops.

If there are multiple types of vertical coordinates, the convert routines must be able to convert between them. The 'set_vertical_localization_coord()' routine should be called from 'static_init_model()' to set what localization coordinate type is being requested.

The three routines related to vertical coordinates/localization choices are:

- `set_vert_localization_coord` - sets the vertical localization coordiate (not required if there is no vertical conversion)
- `convert_vertical_obs` - converts observation location to required vertical type (does nothing if there is no vertical conversion)
- `convert_vertical_state` - converts state vector location to required vertical type (does nothing if there is no vertical conversion)

6.62.5 DART diagnostic file changes

For large models DART format diagnostic files (`Prior_Diag.nc` and `Posterior_Diag.nc`) have been replaced with separate files for each copy that would have gone into Prior_Diag.nc and Posterior_Diag.nc.

For Prior_Diag.nc:

- **Mean and standard deviation**: preassim_mean.nc preassim_sd.nc
- **Inflation mean and standard deviation** (if state space inflation is used): preassim_priorinf_mean.nc preassim_priorinf_sd.nc
- **The number of ensemble members specifed** in filter_nml (num_output_state_members): preassim_member_####.nc

For Posterior_Diag.nc:

- **Mean and standard deviation**: postassim_mean.nc postassim_sd.nc
- **Inflation mean and standard deviation** (if state space inflation is used): postassim_priorinf_mean.nc postassim_priorinf_sd.nc
- **The number of ensemble members specifed** in filter_nml (num_output_state_members): postassim_member_####.nc

The `num_output_state_members` are not written separately from the restarts. Note that restarts will have been clamped if any clamping is applied (given as an arguement to add_domain). This is *different* to Posterior_Diag.nc which contains unclamped values. Note also that there are 2 more "stages" which might be output, in addition to the preassim and postassim discussed here.

For models with multiple domains the filenames above are appended with the domain number, e.g. preassim_mean.nc becomes preassim_mean_d01.nc, preassim_mean_d02.nc, etc.

Changes to nc_write_model_atts

`nc_write_model_atts` now has 2 arguments:

- ncid - open netcdf file identifier
- domain_id - domain number being written

The calling code will write the model state, so this routine should only add attributes and optionally, non-state information like grid arrays.

This routine will only be called if DART is creating an output NetCDF file from scratch. This may include any of the preassim, postassim, or output files.

Changes to nc_write_model_vars

`nc_write_model_vars` is currently unused (and in fact uncalled). It remains for possible future expansion.

6.62.6 Model_mod.f90 interface changes

The model_mod.f90 file contains all code that is specific to any particular model. The code in this file is highly constrained since these routines are *called by* other code in the DART system. All routine interfaces – the names, number of arguments, and the names of those arguments – must match the prescribed interfaces exactly. Since not all required interfaces are needed for every model there are default routines provided that can be referenced from a ‘use’ statement and then the routine name can be put in the module ‘public’ list without any code for that routine having to be written in the model_mod.f90 file.

The following 18 routines are required:

- static_init_model
- get_model_size
- get_state_meta_data
- shortest_time_between_assimilations
- model_interpolate
- end_model
- nc_write_model_atts
- nc_write_model_vars
- init_time
- init_conditions
- adv_1step
- pert_model_copies
- get_close_obs
- get_close_state
- convert_vertical_obs
- convert_vertical_state
- read_model_time
- write_model_time

Here is an example of code from the top of a model_mod file, including the modules where the default routines live and the required public list.

```
use     location_mod, only : location_type, get_close_type, &
                             get_close_obs, get_close_state, &
                             convert_vertical_obs, convert_vertical_state, &
                             set_location, set_location_missing, &
                             set_vertical_localization_coord
use    utilities_mod, only : register_module, error_handler, &
                             E_ERR, E_MSG
                             ! nmlfileunit, do_output, do_nml_file, do_nml_term,  &
                             ! find_namelist_in_file, check_namelist_read
use netcdf_utilities_mod, only : nc_add_global_attribute, nc_synchronize_file, &
                                 nc_add_global_creation_time, &
                                 nc_begin_define_mode, nc_end_define_mode
use state_structure_mod, only : add_domain
use ensemble_manager_mod, only : ensemble_type
use dart_time_io_mod, only  : read_model_time, write_model_time
use default_model_mod, only : pert_model_copies, nc_write_model_vars

implicit none
private

! required by DART code - will be called from filter and other
! DART executables.  interfaces to these routines are fixed and
! cannot be changed in any way.
public :: static_init_model,      &
          get_model_size,         &
          get_state_meta_data,    &
          shortest_time_between_assimilations, &
          model_interpolate,      &
          end_model,              &
          nc_write_model_atts,    &
          adv_1step,              &
          init_time,              &
          init_conditions

! public but in another module
public :: nc_write_model_vars,    &
          pert_model_copies,      &
          get_close_obs,          &
          get_close_state,        &
          convert_vertical_obs,   &
          convert_vertical_state, &
          read_model_time,        &
          write_model_time
```

6.62.7 Observation quantity replaces kinds

Historically there has been confusion about the terms for specific observation types (which often include the name of the instrument collecting the data) and the generic quantity that is being measured (e.g. temperature). The previous terms for these were ‘types’ and ‘kinds’, respectively.

Starting with the Manhattan release we have tried to clarify the terminology and make the interfaces consistent. The following table lists the original names from the Lanai/Classic release and the replacement routines in Manhattan.

All code that is part of the DART code repository has been updated to use the replacment routines, but if you have your own utilities written using this code, you will need to update your code. Contact us (dart@ucar.edu) for help if you have any questions.

```
public subroutines, existing name on left, replacement on right:

assimilate_this_obs_kind()     =>     assimilate_this_type_of_obs(type_index)
evaluate_this_obs_kind()       =>       evaluate_this_type_of_obs(type_index)
use_ext_prior_this_obs_kind()  =>  use_ext_prior_this_type_of_obs(type_index)

get_num_obs_kinds()      =>  get_num_types_of_obs()
get_num_raw_obs_kinds()  =>  get_num_quantities()

get_obs_kind_index()     => get_index_for_type_of_obs(type_name)
get_obs_kind_name()      => get_name_for_type_of_obs(type_index)

get_raw_obs_kind_index()  =>  get_index_for_quantity(quant_name)
get_raw_obs_kind_name()   =>  get_name_for_quantity(quant_index)

get_obs_kind_var_type()  =>  get_quantity_for_type_of_obs(type_index)

get_obs_kind()      =>  get_obs_def_type_of_obs(obs_def)
set_obs_def_kind()  =>  set_obs_def_type_of_obs(obs_def)

get_kind_from_menu()      =>  get_type_of_obs_from_menu()

read_obs_kind()     =>   read_type_of_obs_table(file_unit, file_format)
write_obs_kind()    =>  write_type_of_obs_table(file_unit, file_format)

maps obs_seq nums to specific type nums, only used in read_obs_seq:
map_def_index()  => map_type_of_obs_table()

removed.  apparently unused, and simply calls get_obs_kind_name():
get_obs_name()

apparently unused anywhere, removed:
add_wind_names()
do_obs_form_pair()

public integer parameter constants and subroutine formal argument names,
old on left, new on right:

KIND_ => QTY_
kind => quantity

TYPE_ => TYPE_
type => type_of_obs

integer parameters:
```

(continues on next page)

(continued from previous page)

```
max_obs_generic  =>  max_defined_quantities  (not currently public, leave private)
max_obs_kinds    =>  max_defined_types_of_obs
```

6.62.8 Additions/changes to existing namelists

Quality_control_nml

These namelist options used to be in filter_nml, now they are in quality_control_nml.

```
&quality_control_nml
   input_qc_threshold          = 3,
   outlier_threshold           = 4,
   enable_special_outlier_code = .false.
/
```

New namelist variables

filter_nml

```
&filter_nml
   single_file_in               = .false.,
   single_file_out              = .false.,

   input_state_file_list        = 'null',
   output_state_file_list       = 'null',
   input_state_files            = 'null',
   output_state_files           = 'null',

   stages_to_write              = 'output'
   write_all_stages_at_end      = .false.
   output_restarts              = .true.
   output_mean                  = .true.
   output_sd                    = .true.

   perturb_from_single_instance = .false.,
   perturbation_amplitude       = 0.2_r8,

   distributed_state            = .true.
/
```

Item	Type	Description
single_file_in	logical	True means that all of the restart and inflation information is read from a single NetCDF file. False means that you must specify an input_state_file_list and DART will be expecting input_{priorinf,postinf}_{mean,sd}.nc files for inflation.
single_file_out	logical	True means that all of the restart and inflation information is written to a single NetCDF file. False means that you must specify a output_state_files and DART will be output files specified in the list. Inflation files will be written in the form i nput_{priorinf,postinf}_{mean,sd}.nc.
input_state_files	character array	This is used for single file input for low order models. For multiple domains you can specify a file for each domain. When specifying a list single_file_in, single_file_out must be set to .true.
output_state_files	character array	This is used for single file input for low order models. For multiple domains you can specify a file for each domain. When specifying a list single_file_in, single_file_out must be set to .true.
input_state_file_list	character array	A list of files containing input model restarts. For multiple domains you can specify a file for each domain. When specifying a list single_file_in, single_file_out must be set to .false.
output_state_file_list	character array	A list of files containing output model restarts. For multiple domains you can specify a file for each domain. When specifying a list single_file_in, single_file_out must be set to .false.
stages_to_write	character array	Controls which stages to write. Currently there are four options: - `input` – writes input mean and sd only - `preassim` – before assimilation, before prior inflation is applied - `postassim` – after assimilation, before posterior inflation is applied - `output` – final output for filter which includes clamping and inflation
write_all_stages_at_end	logical	True means output all stages at the end of filter. This is more memory intensive but requires less time.

NetCDF reads and writes:

For **input** file names:

- give `input_state_file_list` a file for each domain, each of which contains a list of restart files. An example of an 'input_list.txt' might look something like :

```
advance_temp1/wrfinput_d01
advance_temp2/wrfinput_d01
advance_temp3/wrfinput_d01
advance_temp4/wrfinput_d01
advance_temp5/wrfinput_d01
....
```

- if no `input_state_file_list` is provided then default filenames will be used e.g. input_member_####.nc, input_priorinf_mean.nc, input_priorinf_sd.nc

For **output** file names:

- give `output_state_file_list` a file for each domain, each of which contains a list of restart files. An example of an 'input_list.txt' might for WRF might look something like :

```
wrf_out_d01.0001.nc
wrf_out_d01.0002.nc
wrf_out_d01.0003.nc
wrf_out_d01.0004.nc
wrf_out_d01.0005.nc
....
```

  if you would like to simply like to overwrite your previous data input_list.txt = output_list.txt

- if no `output_state_files` is provided then default filenames will be used e.g. output_member_####.nc, output_priorinf_mean.nc, output_priorinf_sd.nc

For small models you may want to use `single_file_in`, `single_file_out` which contains all copies needed to run filter.

State_vector_io_nml

```
&state_vector_io_nml
   buffer_state_io          = .false.,
   single_precision_output  = .false.,
/
```

When `buffer_state_io` is `.false.` the entire state is read into memory at once if .true. variables are read one at a time. If your model can not fit into memory at once this must be set to `.true.` .

`single_precision_output` allows you to run filter in double precision but write NetCDF files in single presision

Assim_tools_nml

```
&assim_tools_nml
   distribute_mean  = .true.
/
```

In previous DART releases, each processor gets a copy of the mean (in ens_mean_for_model). In RMA DART, the mean is distributed across all processors. However, a user can choose to have a copy of the mean on each processor by setting `distribute_mean = .false.` . Note that the mean state is accessed through `get_state` whether distribute_mean is `.true.` or `.false.`

Removed from existing namelists

```
&filter_nml
   input_qc_threshold          = 3,
   outlier_threshold           = 4,
   enable_special_outlier_code = .false.
   start_from_restart          = .false.
   output_inflation            = .true.
   output_restart              = .true.
   /
```

NOTE : `output_restart` has been renamed to `output_restarts`. **``output_inflation`` is no longer supported** and only writes inflation files if `inf_flavor > 1`

```
&ensemble_manager_nml
   single_restart_file_out = .true.
   perturbation_amplitude  = 0.2,
   /
```

```
&assim_manager_nml
   write_binary_restart_files = .true.,
   netCDF_large_file_support  = .false.
   /
```

6.62.9 Perturbations

The option to perturb one ensemble member to produce an ensemble is in filter_nml:`perturb_from_single_instance`. The model_mod interface is now `pert_model_copies` not `pert_model_state`. Each task perturbs every ensemble member for its own subsection of state. This is more complicated than the Lanai routine `pert_model_state`, where a whole state vector is available. If a model_mod does not provide a perturb interface, filter will do the perturbing with an amplitude set in filter_nml:perturbation_amplitude. Note the perturb namelist options have been removed from ensemble_manager_nml

6.63 Forward Operator

In Lanai the forward operator is performed by the first ens_size processors. This was because access to the whole state vector is required for the forward operator, and only the first ens_size processors had the whole state vector. The distributed state forward operator has a diffent loop structure to Lanai because all processors can do the foward operator for their observations.

The forward operator is performed in `get_obs_ens_distrb_state`. A limited call tree for `get_obs_ens_distrb_state` is shown below.

The QC_LOOP is in `get_obs_ens_distrb_state` because the qc across the ensemble is known. This removes the need for a transpose of the forward_op_ens_handle. Note this is different from Lanai. The window opening and closing in `get_obs_ens_distrb_state` is as follows:

1. State window created (processors can access other processor's memory)
2. Forward operator called
3. QC calculated
4. State window destroyed (processors can no longer access other processor's memory)

However, there may be occasions where having only the first ens_size processors perform the forward operator. For example, if the forward operator is being read from a file, or the forward operator uses a large portion of the state. Or when debugging it may be easier to have 1 task per ensemble member.

To transpose and do the forward operators like Lanai, you can use the filter_nml namelist option distribute_state = .false. The process is the same as above except the window creation and destruction are transposing the state.

1. State window created (state ensemble is transposed var complete)
2. Forward operator called
3. QC calculated
4. State window destroyed (state ensemble is tranaposed to copy complete)

Note, that if you have fewer tasks than ensemble members some tasks will still be doing vectorized forward operators (because they own more than one ensemble member).

6.63.1 State access

Model_mod routines no longer get an array containing the state. The state is accessed through the function `get_state`.

```
x = get_state(i, state_handle)
```

where x is the state at index i. `state_handle` is passed from above. During model_interpolate `get_state` returns an array. Durring `get_state` returns a single value (the mean state).

6.64 PROGRM OR MODULE name_of_thing

6.64.1 Overview

This is the place for the general description of the module or program or whatever.

overview issues, how/when/why to use this thing, etc.

more stuff about code, usage, etc.

6.64.2 Namelist

DART namelists are always read from file `input.nml`.

We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&NAMELIST_NML
   name=value,
   name=value,
   name=value
/
```

Item	Type	Description
name	type	(often multi-line) description

6.64.3 Other modules used

```
types_mod
model_mod
assim_tools_mod
time_manager_mod
fms_mod
```

6.64.4 Public interfaces

use xxxxxxx, only :	yyypubtype1
	yyyroutine1
	yyyroutine2
	yyyroutine3

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

```
type location_type
   private
   real(r8) :: x
end type location_type
```

The location type is essential to the fabric of the universe. If we don't know where we are, how do we know which exit to take?

Component	Description
x	is the nondimensional distance [0,1]

subroutine yyyroutine1(var1, var2, var3 [, global])

```
type(time_type),           intent(in)    ::  var1
real(r8), dimension(:),    intent(inout) ::  var2
real(r8), dimension(:,:),  intent(out)   ::  var3
real(r8), optional,        intent(in)    ::  global
```

Returns the resolution of compute domain for either the current processor or the global domain.

This is the best thing since sliced bread. All you have to do is throw some arguments in the call and the subroutine automatically slices, dices and makes julienne fries. *But wait!* There's more!

`var1`	is the number of spark plugs in a '67 Cuda.
`var2`	is used for both input and output. Upon input, this contains the furlongs per weekday for every vertical level. Upon exit, we now have the number of spotted gobies per square hectare at that level.
`var3`	The leftmost dimension pertains to the number of feet in an orange-footed brush fowl, indiginous to Australia. The next dimension is the number of feathers on said fowl, naturally.
global	is really, REALLY useful in certain situations.

notes would go here

function yyyroutine2(var1, var2, var3, [,bob])

```
type(time_type),          intent(in)  ::  var1
real(r8), dimension(:),   intent(in)  ::  var2
real(r8), dimension(:,:), intent(in)  ::  var3
real(r8), optional,       intent(in)  ::  bob
integer, dimension(size(var2))        ::  yyyroutine2
```

Returns the resolution of compute domain for either the current processor or the global domain. All input variables are not changed. Otherwise, this would be a subroutine.

This is the second-best thing since sliced bread. All you have to do is throw some arguments in the call and the function automatically deep fries.

`var1`	the first time you changed your oil.
`var2`	miles between every oil change you've ever done. Don't lie.
`var3`	the distances you've ridden. Each row corresponds to the hour-of-day, each column is a different day-of-the-week.
bob	mean time between failures. in msec.
`yyroutine2`	number of gray hairs as a function of time. in kilohairs.

notes would go here

6.64.5 Files

This is the place to discuss the files that are associated with this module. They could be input files, output files, data files, shell scripts … anything.

filename	purpose
inputfile1	to read some input
input.nml	to read namelists
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.64.6 References

- Anderson, J., T. Hoar, K. Raeder, H. Liu, N. Collins, R. Torn, and A. Arellano, 2009: The Data Assimilation Research Testbed: A Community Facility. Bull. Amer. Meteor. Soc., 90, 1283-1296. DOI: 10.1175/2009BAMS2618.1
- none

6.64.7 Private components

N/A

Any routines or 'local' variables of interest may be discussed here. There are generally lots of 'internal' functions that make life simpler, but you don't want to make them available outside the scope of the current module. This is the place to point them out, if you like.

```
type location_type
   private
   real(r8) :: x
end type location_type
```

6.65 PROGRM OR MODULE name_of_thing

6.65.1 Overview

Explain in general terms what this is.

6.65.2 Namelist

DART namelists are always read from file `input.nml`.

We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&NAMELIST_NML
   name=value,
   name=value,
   name=value
/
```

Any comments about the namelist as a whole.

Item	Type	Description
name	type	(often multi-line) description

6.65.3 Modules used

```
types_mod
utilities_mod
random_seq_mod
time_manager_mod
ensemble_manager_mod
```

6.65.4 Public interfaces

use this_module_name_mod, only :	subr/function name
	name2
	name3

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

subroutine subroutine1(arg1, [, arg2])

```
real(r8),           intent(in) :: arg1
real(r8), optional, intent(in) :: arg2
```

describe what this subroutine does.

`arg1`	Describe arg1.
arg2	Describe optional arg2.

function function1(arg1)

```
logical,              :: function1
integer, intent(in)   :: arg1
```

Describe function.

`function1`	describe what this function returns
`arg1`	describe function argument

```
type bob_type
   private
   integer :: bob1
   integer :: bob2
end type bob_type
```

describe bob

Component	Description
bob1	Describe bob1.
bob2	Describe bob2.

6.65.5 Files

describe files used by code

6.65.6 References

- author, title. publication, volume, pages. doi: nn.yyyy/rest_of_number

6.65.7 Private components

no discussion

6.66 Vertical Conversion of Observations

In Lanai vertical conversion of observations occurs in get_close_obs. The Lanai code in filter_assim is as follows:

If this algorithm was followed in RMA DART, all processors would have to communicate to calculate the location of observation i. This is a huge amount of contention since all processors are doing exactly the same calculation so need to access exactly the same state elements. This causes the code to run very slowly, for example 1 minute for 1000 observations, versus 5 seconds for 1000 observations for Lanai.

However, there is no need to calculate the vertical conversion inside the SEQUENTIAL_OBS do loop, since the mean state vector used is not updated during the loop. (In Lanai it is the array passed to the model_mod by ens_mean_for_model in filter_main). Also this calculation does not scale, because all processors do the same calculation.

In DART RMA the owner of an observation converts the vertical location of an observation and broacasts it to all other processors as part of the broadcast in the SEQUENTIAL_OBS do loop.

The DART RMA code calculates the vertical of all observations before the loop. This potentially scales better because processors only calculate their own observation conversions, but does require model_mod interfaces for vertical conversion.

The DART RMA code in filter_assim is as follows:

```
do i =, obs_ens_handle%my_num_vars
   call convert_vertical_location(my_obs_loc(i))
end do
SEQUENTIAL_OBS do i = 1, obs_ens_handle%num_vars
   ...
   broadcast increments and vertical location for observation i
   ...
enddo
```

6.66.1 Bitwise problem

Moving the `convert_vertical_location` changes the number of `get/set location` calls. There is a bitwise creep of the location when you do this. This is in the conversion from degrees to radians and back again. If you want to do the exact number of `get/set location` you can change the line lanai_bitwise = .false. to lanai_bitwise = .true. in assim_tools_mod.f90. Note this is not a namelist option because production code should not be run with lanai_bitwise = .true. For more detail on running bitwise with Lanai see *Bitwise Considerations*.

6.67 Bitwise Considerations

By bitwise we mean bit for bit identical results in the output obs_sequence file and the restarts (netcdf-to-netcdf or DART format-to-dart format) when comparing one version of the code to another. For testing the code to be bitwise with Lanai there are several things to change/set in Manhattan and your mkmf:

6.67.1 Important

The *CAM* and *bgrid_solo* model_mods have been altered so the state is in a different order inside filter. Thus DART format restarts will **not** be bitwise with Lanai DART format restarts, but netcdf files will be (after running dart_to_cam).

6.68 Netcdf Inflation Files

The filter_nml now read restart and inflation files directly from NetCDF files

Netcdf inflation files are no longer special files. DART format inflation files were always 2 copies in one file (mean and standard devation). Taking away this special status of inflation files has the advantage that all copies (restarts, ensemble mean, ensemble standard deviation, inflation mean, inflation sd, etc.) can all be treated the same for IO purposes. Since there are two inflation files when reading/writing netcdf the filenames are different to DART format restart files.

The names of the netcdf inflation files are now fixed.

Input inflation file names

The filter_nml option:

`inf_in_file_name = prior_inflation_ics, post_inflation_ics`

has been **deprecated** and for 1 domain filter is expecting to read:

input_{priorinf,postinf}_mean.nc
input_{priorinf,postinf}_sd.nc

For multiple domains filter is expecting to read:

input_{priorinf,postinf}_mean_d01.nc
input_{priorinf,postinf}_sd_d01.nc
input_{priorinf,postinf}_mean_d02.nc
input_{priorinf,postinf}_sd_d02.nc

where d0* is the domain number.

Output inflation file names

The filter_nml option:

`inf_out_file_name = prior_inflation_restart, post_inflation_restart`

has been **deprecated** and for 1 domain filter is expecting to read:

output_{priorinf,postinf}_mean.nc
output_{priorinf,postinf}_sd.nc

For multiple domains filter is expecting to write:

prior_inflation_restart_mean_d01
prior_inflation_restart_sd_d01
prior_inflation_restart_mean_d02
prior_inflation_restart_sd_d02

where d0* is the domain number.

6.69 State Stucture

State_structure_mod is a module that holds all the domain, variable, dimension info about the model_mods in the state. Note it stores only

It is the foundation for two parts of the code:

- Read/write state variables from/to netcdf files
- Calculate DART index from x,y,z variable indices and the inverse: x,y,z, variable from DART index.

Inside `static_init_model` a call is made to `add_domain`. This call is *required* as it communicates to the state structure that a new domain has been added to the state. The state structure keeps track of the number of domains in the state. These may be multiple domains in one model_mod, e.g. nested domains in WRF, or multiple model_mods, e.g. POP coupled with CAM. The minimum amount of information `add_domain` needs is model size which means vector of length model size has been added to the state. This equivalent to Lanai where the only information filter has is that the model is a vector of length model_size. For models with netcdf restart files you supply `add_domain` with:

- a netcdf file
- the number of variables to read from the file

- the name of the variables
- Optionally:
  - the DART KINDS of the variables
  - clamping upper and lower bounds
  - update/not update this variable

For models that are spun up in perfect_model_obs you can manually describe the variables so you can create netcdf files containing the varibles in the model state, e.g. Temperature, Surface Pressure, etc. There are 3 steps to this process:

1. Supply `add_domain` with almost the same arguments as you would for a netcdf file, but skip the first arguement (netcdf filename).
2. For each variable, loop around the required number of dimensions and call `add_dimension_to_variable`
3. Call `finished_adding_domain` to let the state structure know that you have finished adding dimensions to variables.

6.69.1 DART index

To get the dart index for an i,j,k,variable in a domain use:
`get_dart_vector_index(i, j, k, dom_id, var_id)`

To get the i,j,k, variable, domain from the dart index use:
`get_model_variable_indices(dart_index, i, j, k, var_id, dom_id)`

Note That (i,j,k) needs to be converted to (lon, lat, lev) or to whatever grid the variable is on. `get_dim_name` can be used to get the dimension name from i,j,k if needed.

6.69.2 Unlimited dimensions: io vs model_mod routines

Some model restart files have an unlimited dimension. For IO purposes, e.g. creating netcdf files, the unlimited dimension is used. For state structure accessor functions called be the model_mod the unlimited dimension is ignored. So if you have a variable TEMPARATURE in your netcdf file, with dimensions (lon, lat, level, time) the IO routines will see a 4D variable, but `get_num_dims` used in model_mod will return 3D.

6.70 Filter async modes

6.70.1 Options for parallelism both in DART and in the model advances

Simplest case, async=0:

This is a single MPI executable, with each call to the model being simply a subroutine call from each MPI task.
To the DART mpi intro document
Parallel advance, async=2:

The filter executable is one MPI program, and the model is a single, sequential executable. Each MPI task uses the unix "system()" call to invoke a shell script (advance_model.csh) which runs the models as independent programs.
To the DART mpi intro document
Other views of how the async=2 option is structured; these may be more or less helpful.
Parallel advance, async=2:

Parallel advance, async=2, second version:

Parallel model advance, async=2, showing how data is communicated between filter and the model thru intermediate files. IC are 'initial condition' files, UD are 'updated' files.

Parallel model advance, async=4:

The filter executable is one MPI program, and the model is also an MPI program. The filter executable communicates with the runme_filter shell script, which sequentially invokes mpirun to advance each of the model runs, one per ensemble member, still using advance_model.csh.
To the DART mpi intro document
Parallel model advance, async=4, showing how data is communicated between filter and the model thru intermediate files. IC are 'initial condition' files, UD are 'updated' files.

6.71 Distributed State

The key part of RMA DART is having a state that is physically distributed across processors. The location in memory of any part of the state vector (which processor and where in memory on that processor) is completely under the control of filter, not model_mod.

Implications of this:

- The model_mod never gets a whole state vector to use. So no whole vector for a forward operator, and no whole vector for the mean.
- The model_mod can not make any assumptions about the order of elements in the state. Currently, filter is ordering variables in the order they are listed in add_domain and with the dimenion order of the netcdf file. This is what is happening in most model_mod converters (model_to_dart, dart_to_model). However CAM and bgrid_solo rearrange the state in Lanai. These model_mods (and converters) have been changed to not rearrage the state.

So, how does the model_mod access the state without having the vector and not knowing the state order? - state accessor routines.

6.71.1 State accessor routines

Getting the dart index

```
function get_dart_vector_index(i, j, k, dom_id, var_id)
```

get_dart_vector_index returns the dart index for a given i,j,k of a variable. Note if the variable is 1D j and k are ignored. If a variable is 2D k is ignored. Note only variables upto 3D are supported, but this could be extended to support upto 7 dimensional variables (or whatever fortran and netcdf will support).

Getting the state at a given dart index

```
function x = get_state(index, state_handle)
```

get_state returns the state x at the given index. state_handle is a derived type which conatins the state information. state_handle is passed to the model_mod from above. get_state returns an array of values (the whole ensemble at index) during model_mod and a single value (the mean) during get_close_obs or vert_convert.

If you have an array of indices, for example a forward operator which is located in different levels on different ensemble members you can use get_state_array. An example of this is in CAM when an observation is in pressure, the level an observation is in depends on the state and so can vary across the ensemble.

```
subroutine get_state_array(x(:),  index(:),  state_handle)
```

The code inside get_state_array will do the minimum amount of communication to get you the indices you need. For example if

index = [3 4 3 3 4 3]

get_state_array will only do 2 mpi communications and return

x = [state(3), state(4), state(3), state(3), state(4), state(3)]

A limited module diagram is shown below. A -> B means A uses B:

Filter_mod and assim_tools_mod take care of making data available for use with get_state. Note get_state will only return data during *model_interpolate*, *get_close_obs*, or *vert_convert*. If you use get_state outside these routines you will get and error.

Compliation Notes

The Remote Memory Access programming model we are using uses mpi_windows. There are 2 ways to compile window mods for mpi and non-mpi filter. This is taken care of automatically when you run quickbuild.csh or an `mkmf_*` with -mpi or -nompi. However, if you use mpi, there is a choice of mpi_window mods:

- cray_win_mod.f90
- no_cray_win_mod.f90

We have these two modules that you can swap in your path_names files because the MPI 2 standard states:

Implementors may restrict the use of RMA communication that is synchronized by lock calls to windows in memory allocated by MPI_ALLOC_MEM.

MPI_ALLOC_MEM uses cray pointers, thus we have supplied a window module that uses cray pointers. However, no_cray_win_mod.f90 is the default since some versions of gfortran (4.9.0) do not support cray pointers. These different modules will go away when we swap to MPI 3.

6.72 DART Lanai Differences from Kodiak Release Notes

6.72.1 Overview

This document includes an overview of the changes in the DART system since the Kodiak release. For further details on any of these items look at the HTML documentation for that specific part of the system.

There is a longer companion document for this release, the *Lanai*, which include installation instructions, a walk-through of running one of the low-order models, the diagnostics, and a description of non-backward compatible changes. See the Notes for Current Users section for additional information on changes in this release.

6.72.2 Changes to core DART routines

This section describes changes in the basic DART library routines since the Kodiak release.

- Added a completely new random number generator based on the Mersenne Twister algorithm from the GNU scientific library. It seems to have better behavior if reseeded frequently, which is a possible usage pattern if perfect_model_obs is run for only single steps and the model is advanced in an external script. As part of this code update all random number code was moved into the random_seq_mod and random_nr_mod is deprecated.
- Perfect_model_obs calls a seed routine in the time manager now that generates a consistent seed based on the current time of the state. This makes subsequent runs give consistent results and yet separate runs don't get identical error values.
- Added random number generator seeds in several routines to try to get consistent results no matter how many MPI tasks the code was run with. This includes:
  - cam model_mod.f90, pert_model_state()
  - assim_tools_mod.f90, filter_assim(), filter kinds 2, 3, and 5
  - wrf model_mod.f90, pert_model_state()
  - adaptive_inflate_mod.f90, adaptive_inflate_init(), non-deterministic inf
- There is a new &filter_nml namelist item: enable_special_outlier_code. If .true. the DART quality control code will call a separate subroutine at the end of filter.f90 to evaluate the outlier threshold. The user can add code to that routine to change the threshold based on observation type or values as they wish. If .false. the default filter outlier threshold code will be called and the user routine ignored.
- If your `model_mod.f90` provides a customized `get_close_obs()` routine that makes use of the types/kinds arguments for either the base location or the close location list, there is an important change in this release. The fifth argument to the `get_close_obs()` call is now a list of generic kinds corresponding to the location list. The fourth argument to the `get_dist()` routine is now also a generic kind and not a specific type. In previous versions of the system the list of close locations was sometimes a list of specific types and other times a list of generic kinds. The system now always passes generic kinds for the close locations list for consistency. The base location and specific type remains the same as before. If you have a `get_close_obs()` routine in your `model_mod.f90` file and have questions about usage, contact the DART development team.
- Filter will call the end_model() subroutine in the model_mod for the first time. It should have been called all along, but was not.
- Added a time sort routine in the time_manager_mod.
- Avoid a pair of all-to-all transposes when setting the inflation mean and sd from the namelist. The new code finds the task which has the two copies and sets them directly without a transpose. The log messages were also moved to the end of the routine - if you read in the mean/sd values from a restart file the log messages that printed out the min/max values needed to be after the read from the file.
- Reordered the send/receive loops in the all-to-all transposes to scale better on yellowstone.
- Remove a state-vector size array from the stack in read_ensemble_restart(). The array is now allocated only if needed and then deallocated. The ensemble write routine was changed before the Kodiak release but the same code in read was apparently not changed simply as an oversight.
- If the ensemble mean is selected to be written out in dart restart file format, the date might not have been updated correctly. The code was fixed to ensure the ensemble mean date in the file was correct.
- filter writes the ensemble size into the log file.
- Reorganized the code in the section of obs_model_mod that prints out the time windows, with and without verbose details. Should be clearer if the next observation is in or out of the current assimilation window, and if the model needs to advance or not.

- Added a fill_inflation_restart utility which can write a file with a fixed mean and sd, so the first step of a long assimilation run can use the same 'start_from_restart_file' as subsequent steps.
- Added new location module options:
  - Channel coordinate system
  - [0-1] periodic 3D coordinate system
  - X,Y,Z 3D Cartesian coordinate system
  - 2D annulus coordinate system

6.72.3 New models or changes to existing models

Several new models have been incorporated into DART. This section details both changes to existing models and descriptions of new models that have been added since the Kodiak release.

- Support for components under the CESM framework:
  - Added support for the Community Land Model (CLM).
  - Added support to run the Community Atmospheric Model (CAM) under the CESM framework.
  - Added support for the CESM 1.1.1 release for CAM, POP, CLM: includes experiment setup scripts and assimilation scripts.
  - CAM, POP, and/or CLM can be assimilated either individually or in combination while running under the CESM framework. If assimilating into multiple components, they are assimilated sequentially with observations only affecting a single component directly. Other components are indirectly affected through interactions with the coupler.
  - Setup scripts are provided to configure a CESM experiment using the multi-instance feature of CESM to support ensembles for assimilation.
  - POP state vector contains potential temperature; observations from the World Ocean Database are in-situ or sensible temperature. The model_mod now corrects for this.
    - The state vector has all along contained potential temperature and not in-situ (sensible) temperature. The observations from the World Ocean Database are of sensible temperature. Changed the specific kind in the model_mod to be `QTY_POTENTIAL_TEMPERATURE` and added new code to convert from potential to in-situ temperature. Differences for even the deeper obs (4-5km) is still small (~ 0.2 degree). (in-situ or sensible temperature is what you measure with a regular thermometer.)
  - Support for the SE core (HOMME) of CAM has been developed but **is not** part of the current release. Contact the DART group if you have an interest in running this configuration of CAM.
- Changes to the WRF model_mod:
  - Allow advanced microphysics schemes (needed interpolation for 7 new kinds)
  - Interpolation in the vertical is done in log(p) instead of linear pressure space. log(p) is the default, but a compile-time variable can restore the linear interpolation.
  - Added support in the namelist to avoid writing updated fields back into the wrf netcdf files. The fields are still updated during the assimilation but the updated data is not written back to the wrfinput file during the dart_to_wrf step.

  - Fixed an obscure bug in the vertical convert routine of the wrf model_mod that would occasionally fail to convert an obs. This would make tiny differences in the output as the number of mpi tasks change. No quantitative differences in the results but they were not bitwise compatible before and they are again now.
- Added support for the MPAS_ATM and MPAS_OCN models.
  - Added interpolation routines for the voroni-tesselation grid (roughly hexagonal)
  - Includes vertical conversion routines for vertical localization.
  - Added code to the mpas_atm model to interpolate specific humidity and pressure, so we can assimilate GPS obs now.
- Added support for the 'SQG' uniform PV two-surface QC+1 spectral model.
- Added support for a flux-transport solar dynamo model.
- Added support for the GITM upper atmosphere model.
- Added support for the NOAH land model.
- Added support for the NAAPS model.
- Added model_mod interface code for the NOGAPS model to the SVN repository.
- Simple advection model:
  - Fix where the random number seed is set in the models/simple_advection model_mod - it needed to be sooner than it was being called.

6.72.4 New or changed forward operators

This section describes changes to the Foward Operators and new Generic Kinds or Specific Types that have been added since the Kodiak release.

- Many new kinds added to the DEFAULT_obs_kind_mod.f90:
  - QTY_CANOPY_WATER
  - QTY_CARBON
  - QTY_CLW_PATH
  - QTY_DIFFERENTIAL_REFLECTIVITY
  - QTY_DUST
  - QTY_EDGE_NORMAL_SPEED
  - QTY_FLASH_RATE_2D
  - QTY_GRAUPEL_VOLUME
  - QTY_GROUND_HEAT_FLUX QTY_HAIL_MIXING_RATIO
  - QTY_HAIL_NUMBER_CONCENTR
  - QTY_HAIL_VOLUME QTY_ICE QTY_INTEGRATED_AOD
  - QTY_INTEGRATED_DUST
  - QTY_INTEGRATED_SEASALT QTY_INTEGRATED_SMOKE
  - QTY_INTEGRATED_SULFATE
  - QTY_LATENT_HEAT_FLUX
  - QTY_LEAF_AREA_INDEX

  - QTY_LEAF_CARBON
  - QTY_LEAF_NITROGEN QTY_LIQUID_WATER
  - QTY_MICROWAVE_BRIGHT_TEMP
  - QTY_NET_CARBON_FLUX
  - QTY_NET_CARBON_PRODUCTION
  - QTY_NEUTRON_INTENSITY
  - QTY_NITROGEN QTY_RADIATION
  - QTY_ROOT_CARBON
  - QTY_ROOT_NITROGEN
  - QTY_SEASALT
  - QTY_SENSIBLE_HEAT_FLUX
  - QTY_SMOKE
  - QTY_SNOWCOVER_FRAC
  - QTY_SNOW_THICKNESS
  - QTY_SNOW_WATER
  - QTY_SO2
  - QTY_SOIL_CARBON
  - QTY_SOIL_NITROGEN
  - QTY_SPECIFIC_DIFFERENTIAL_PHASE
  - QTY_STEM_CARBON
  - QTY_STEM_NITROGEN
  - QTY_SULFATE
  - QTY_VORTEX_WMAX
  - QTY_WATER_TABLE_DEPTH
  - QTY_WIND_TURBINE_POWER
  - plus slots 151-250 reserved for Chemistry (specifically WRF-Chem) kinds

- Added a forward operator for total precipitable water. It loops over model levels so it can be used as an example of how to handle this without having to hardcode the number of levels into the operator.
- Added a forward operator (and obs_seq file converter) for COSMOS ground moisture observations.
- Added a forward operator (and obs_seq file converter) for MIDAS observations of Total Electron Count.
- Added a 'set_1d_integral()' routine to the obs_def_1d_state_mod.f90 forward operator for the low order models. This subroutine isn't used by filter but it would be needed if someone wanted to write a standalone program to generate obs of this type. We use this file as an example of how to write an obs type that has metadata, but we need to give an example of how to set the metadata if you aren't using create_obs_sequence interactively (e.g. your data is in netcdf and you have a separate converter program.)

6.72.5 Observation converters

This section describes support for new observation types or sources that have been added since the Kodiak release.

- Added an obs_sequence converter for wind profiler data from MADIS.
- Added an obs_sequence converter for Ameriflux land observations(latent heat flux, sensible heat flux, net ecosystem production).
- Added an obs_sequence converter for MODIS snow coverage measurements.
- Added an obs_sequence converter for COSMOS ground moisture observations.
- Added an obs_sequence converter for MIDAS observations of Total Electron Count.
- Updated scripts for the GPS converter; added options to convert data from multiple satellites.
- More scripting support in the MADIS obs converters; more error checks added to the rawin converter.
- Added processing for wind profiler observation to the wrf_dart_obs_preprocess program.
- Fix BUG in airs converter - the humidity obs are accumulated across the layers and so the best location for them is the layer midpoint and not on the edges (levels) as the temperature obs are. Also fixed off-by-one error where the converter would make one more obs above the requested top level.
- Made gts_to_dart converter create separate obs types for surface dewpoint vs obs aloft because they have different vertical coordinates.
- Converted mss commands to hpss commands for a couple observation converter shell scripts (inc AIRS).
- New matlab code to generate evenly spaced observations on the surface of a sphere (e.g. the globe).
- Added obs_loop.f90 example file in obs_sequence directory; example template for how to construct special purpose obs_sequence tools.
- Change the default in the script for the prepbufr converter so it will swap bytes, since all machines except ibms will need this now.
- The ‘wrf_dart_obs_preprocess’ program now refuses to superob observations that include the pole, since the simple averaging of latitude and longitude that works everyplace else won’t work there. Also treats observations near the prime meridian more correctly.

6.72.6 New or updated DART diagnostics

This section describes new or updated diagnostic routines that have been added since the Kodiak release.

- Handle empty epochs in the obs_seq_to_netcdf converter.
- Added a matlab utility to show the output of a ‘hop’ test (running a model for a continuous period vs. stopping and restarting a run).
- Improved the routine that computes axes tick values in plots with multiple values plotted on the same plot.
- The obs_common_subset program can select common observations from up to 4 observation sequence files at a time.
- Add code in obs_seq_verify to ensure that the ensemble members are in the same order in all netcdf files.
- Added support for the unstructured grids of mpas to our matlab diagnostics.
- Fix to writing of ReportTime in obs_seq_coverage.
- Fixed logic in obs_seq_verify when determining the forecast lat.

- Fixed loops inside obs_seq_coverage which were using the wrong limits on the loops. Fixed writing of 'ntimes' in output netcdf variable.
- The obs_common_subset tool supports comparing more than 2 obs_seq.final files at a time, and will loop over sets of files.
- Rewrote the algorithm in the obs_selection tool so it had better scaling with large numbers of obs.
- Several improvements to the 'obs_diag' program:
  - Added preliminary support for a list of 'trusted obs' in the obs_diag program.
  - Can disable the rank histogram generation with a namelist item.
  - Can define height_edges or heights in the namelist, but not both.
  - The 'rat_cri' namelist item (critical ratio) has been deprecated.
- Extend obs_seq_verify so it can be used for forecasts from a single member. minor changes to obs_selection, obs_seq_coverage and obs_seq_verify to support a single member.
- Added Matlab script to read/print timestamps from binary dart restart/ic files.
- Default for obs_seq_to_netcdf in all the namelists is now 'one big time bin' so you don't have to know the exact timespan of an obs_seq.final file before converting to netCDF.

6.72.7 Tutorial, scripting, setup, builds

This section describes updates and changes to the tutorial materials, scripting, setup, and build information since the Kodiak release.

- The mkmf-generated Makefiles now take care of calling 'fixsystem' if needed so the mpi utilities code compiles without further user intervention.
- Make the default input.nml for the Lorenz 96 and Lorenz 63 model gives good assimilation results. Rename the original input.nml to input.workshop.nml. The workshop_setup script renames it back before doing anything else so this won't break the workshop instructions. Simplify all the workshop_setup.csh scripts to do the minimal work needed by the DART tutorial.
- Updates to the models/template directory with the start of a full 3d geophysical model template. Still under construction.
- Move the pdf files in the tutorial directory up a level. Removed framemaker source files because we no longer have access to a working version of the Framemaker software. Moved routines that generate figures and diagrams to a non-distributed directory of the subversion repository.
- Enable netCDF large file support in the work/input.nml for models which are likely to have large state vectors.
- Minor updates to the doc.css file, make pages look identical in the safari and firefox browsers.
- Added a utility that sorts and reformats namelists, culls all comments to the bottom of the file. Useful for doing diffs and finding duplicated namelists in a file.
- Cleaned up mkmf files - removed files for obsolete platforms and compilers, updated suggested default flags for intel.
- Update the mkmf template for gfortran to allow fortran source lines longer than 132 characters.

6.73 DART "Jamaica release" difference document

This document describes the changes from the "Iceland" or "Post-Iceland" DART releases to the new **jamaica** release.
A word about MPI: It is very important to remember that all the MPI routines are in one module – mpi_utilities_mod.f90. If you are NOT using MPI - there is a simple equivalent module named null_mpi_utilities_mod.f90 which is the default.

6.73.1 These files have changed (or are added)

1. adaptive_inflate_mod.f90: This module was almost completely rewritten although fundamental operations remain unchanged. The only change to have direct impact on users is that there is no longer a namelist for this module (inflation is now controlled through the filter_nml) and inflation restarts are always in a single file, no matter what the choice of multiple restart files for state restarts is.

There have been a number of structural and algorithmic changes. First, the module is now somewhat more object-oriented with an **adaptive_inflate_type** introduced that contains all the information about a particular type of inflation. The **adaptive_inflate_type** includes: the inflation_flavor (observation, spatially fixed state space, or spatially-varying state space) the unit open for inflation diagnostic output, logicals that control whether an inflation restart is read or written, and a logical that controls whether the inflation is done with a deterministic or stochastic algorithm. The file names for the input and output restarts and the diagnostic file are also included. The mean value and standard deviation of the inflation for the spatially fixed state space are in the type. Also a lower bound on the standard deviation and lower and upper bounds on the mean inflation value are included. All calls to apply or adjust inflation values now require an inflate_handle of type adaptive_inflate_type.

Several algorithmic improvements were made to attempt to increase the numerical stability of the inflation algorithms. First, an improved quadratic solver was added. The rest of the inflation code was restructured to reduce the loss of precision in computing the mean of an updated inflation. A minor code change removed the possibility of inflation values of exactly 1.0 introducing inflation through round-off on some compilers. Numerical stability for extreme outliers in ensembles was greatly improved.

There is evidence that inflation computations can continue to be sensitive to numerical round-off when run with (r4) precision instead of (r8). There may be a need to switch to enforced higher precision for challenging applications that require the use of (r4) as the default precision (like some WRF applications).

2. assim_model_mod.f90: There were several changes to the public interfaces. First, get_close_states and get_num_close_states no longer exist. This is part of moving the default ability to find close points out of model_mod and into the location mod. Second, an additional optional argument, override_write_format, was added to function open_restart_write. This allows a caller to force a write in either binary or ascii independent of the namelist setting.

A number of non-algorithmic changes were made internal to the code. These include using the utilities_mod routines to check arguments for netcdf calls and addition of output control for the mpi implementation. More precision was added in diagnostic output of days and seconds to avoid a problem with WRF.

3. assim_tools_mod.f90: Large portions of this module were entirely rewritten. The parallel assimilation algorithm is implemented in the subroutine filter_assim which is now the ONLY public interface into assim_tools. The new parallel assimilation algorithm and its associated inflation computations are documented in the manuscript

by Anderson and Collins that is to appear in the Journal of Atmospheric and Oceanic Technology and in section 22 of the tutorial. The manual definition of ‘regions’ no longer exists. assim_region.csh no longer exists.

The namelist has also seen major changes. Namelist entries do_parallel, num_domains and parallel_command no longer exist. Namelist entry num_close_threshold has been renamed as adaptive_localization_threshold and has a default value of -1 which means no adaptive localization is in use. Namelist entry print_every_nth_obs is new and requests that a message be printed after every nth observation is assimilated in order to monitor progress of large test jobs.

Algorithmically, the `obs_increment_????` routines are unchanged but the ensemble mean and variance are passed in as arguments now to avoid a redundant computation. The update_from_obs_increment routine also has the mean and variance for the observed variable input to avoid extra computation while the correl argument has been made optional and the correlation is only computed if it is requested in the call.

An additional observation space increment option has been added along with the supporting subroutine obs_increment_hybrid subroutine. This interface is not documented at present and should not be used without consulting the DART development team. Additional support for a class of particle filters is provided by subroutine update_ens_from_weights. This routine is also undocumented and should not be used without consulting the DART development team.

4. `types_mod.f90`: Types i4, r4, c4 and c8 were added to the public list and additional comment documentation added.

5. `cov_cutoff_mod.f90`: The interface to comp_cov_factor was modified to add four new arguments, obs_loc, obs_kind, target_loc, and target_kind. These arguments are not used in the default implementation but are made available for applications in which users want to implement more complicated cutoff functions that can be a function of the location and type of both the observation and the state variable.

6. `diagnostics/oned/obs_diag.f90`: The default value of rat_cri in the namelist was changed from 3.0 to 4.0 to avoid throwing out too many observations as outside the expected range. Observation-space diagnostics have been extended to handle the fact that inflation can happen in the prior or posterior (by keying on the new dart_qc variable). The new dart_qc values were used to determine which observations were assimilated for posterior and prior estimates and a histogram of the qc values is output. The namelist variable qc_threshold was renamed input_qc_threshold to further distinguish the fact there are now two types of QC values: dart_qc and whatever comes with the observation to begin with.

7. `diagnostics/threed_sphere/obs_diag`: Observation-space diagnostics have been extended to handle which observations were assimilated for posterior and/or prior estimates (by keying on the new dart_qc variable). The namelist variable qc_threshold was renamed input_qc_threshold to further distinguish the fact there are now two types of QC values: dart_qc and whatever comes with the observation to begin with.

8. `ensemble_manager_mod.f90`: This module has been almost entirely rewritten for the mpi implementation. Its continues to manage ensemble storage but adds a general transpose capability that is central the the parallel implementation.

The namelist retains the single_restart_file_in and single_restart_file_out but drops the in_core entry and adds perturbation_amplitude which controls the standard deviation of perturbations added to the state when starting from a single state estimate and generating a default ensemble. See the html documentation for details on the new interfaces.

9. `filter.f90`: There are major changes throughout filter for the mpi implementation. Foremost is the fact that the async=3 option no longer exists, which removes the need for filter_server.csh; nor do manually-defined 'regions' exist, which removes the need for assim_region.csh.

Changes to namelist: for full details see the html documentation for filter and the tutorial documentation for adaptive inflation. The namelist can be divided into two parts, a new part that controls inflation (previously done in the now defunct adaptive_inflate.nml) and an old part that controls other aspects of the assimilation. In the old part, the namelist entries output_state_ens_mean, output_state_ens_spread, output_obs_ens_mean and output_obs_ens_spread have been removed. Mean and spread are now always output. New entries first_obs_days, first_obs_seconds, last_obs_days and last_obs_seconds have been added. These specify the time of the first and last observation in the obs_sequence file that are to be used. The default values of -1 indicate that these are to be ignored (see html documentation). The entry input_qc_threshold has been added: observations with an associated qc field in the input obs_sequence that is larger than the threshold are not assimilated. The entry output_forward_op_errors (default false) has been added: it outputs a detailed file containing a list of all failed forward observation operators in the assimilation when true. The entry output_timestamps has been added: when true it generates diagnostic ouput bounding each model advance call in filter.

The inflation portion of the namelist is divided into two columns, the first controlling prior inflation and the second controlling posterior inflation. Details of these controls can be found in the html documentation. They are related to the defunct namelist values that were found in adaptive_inflate_mod.nml in the post-I release which supported only prior inflation.

A number of other internal details were changed. The main program is now a single statement that calls a subroutine, filter_main. This allows for better memory management and avoids lots of shared 'module' storage from the main program. In order to support a fully modular smoother capability, the subroutine filter_state_space_diagnostics has been moved to smoother_mod.f90 and smoother_mod is used by filter. The arguments for all previously existing internal subroutines have been modified and are described in the html documentation.

10. `integrate_model.f90`: The namelist integrate_model.nml no longer exists. The old namelist had entries for the input and output files, but they are now hard-coded to 'temp_ic' and 'temp_ud'. The target time for the advance is now taken directly from the input file and the old namelist entries for target_time_days and target_time_seconds are no longer needed. It is essential that integrate_model be compiled WITHOUT the parallel mpi_utilities.

11. `oned/location_mod.f90`: location modules are now responsible for providing interfaces to (efficiently)

find a set of locations that are close to a given location. The new and modified public interfaces are: get_close_obs, get_close_obs_destroy, get_close_maxdist_init, and get_close_obs_init. In addition, a new type, get_close_type, is defined to store information that helps to do this type of search efficiently given a fixed set of locations to check. The oned location_mod doesn't have a need to do this efficiently, so these new interfaces are generally just stubs. The old get_close_obs still works as before to find locations that are within a certain distance of a given location.

The get_dist interface has been changed to include two new arguments, kind1 and kind2. These are the kinds associated with the two locations. These arguments are not used in the default implementation of get_dist, but are made available for users who want to define distances using not only the location but also the kinds.

12. `threed_sphere/location_mod.f90`: The location module is now primarily responsible for the efficient search for close states / obs. A series of new interfaces have been added, along with namelist modifications, to support finding a subset of locations that are close to a single given location. This can be used both for get_close_obs and get_close_state computations and replaces the get_close_state interfaces that were in the model_mod.

A new type, the get_close_type, is defined in a partially object-oriented fashion. For the threed_sphere, the algorithm works by partitioning the surface of the sphere using a longitude/latitude equally-spaced grid. This grid divides the sphere's surface into a set of nlon by nlat boxes. The first step in the efficient search computes the minimum distance between points in boxes that are separated by a given number of boxes in latitude and in longitude. This is accomplished by the new interface get_close_maxdist_init. This routine also accepts a cutoff radius and keeps a list of all box offsets from a box at a given latitude that are possibly within the radius. The second step takes a list of locations and places them into the appropriate boxes. This is performed by new interface get_close_obs_init. Finally, interface get_close_obs finds all observations that are less than the cutoff distance from a single input location and returns their indices in the original list of locations, along with the distance between them and the single base location if requested. An interface, get_close_obs_destroy, is also provided to destroy an instance of the get_close_type.

Three new namelist entries control the number of boxes used in the search, nlon and nlat, and allow for detailed diagnostic output of the performance of the close search, output_box_info. The public interface print_get_close_type is also provided for debug and diagnostic use.

The get_close_obs algorithm only partitions a subset of the sphere's surface into boxes if the input set of locations is confined to a small region of the surface. The algorithm works most efficiently when the average number of locations in a box is small compared to the total number of locations being searched but large compared to 1. Additional guidance in tuning the nlon and nlat control over the number of boxes is available from the DART development team.

The get_dist interface has been changed to include two new arguments, kind1 and kind2. These are the kinds associated with the two locations. These arguments are not used in the default implementation of get_dist, but are made available for users who want to define distances using not only the location but also the kinds.

13. `model_mod.f90`: Unfortunately, there are minor changes to the model_mod public interfaces required to work with the switch to using the location_mod to find close locations. The public interface model_get_close_states is no longer required. Three new interfaces: get_close_maxdist_init, get_close_obs_init, and get_close_obs are required to use the new location module. In a minimal implementation, these three interfaces can just be satisfied by using the interfaces of the same name in the location module. The models/template/model_mod.f90 demonstrates this minimal implementation. Large models can implement their own modified algorithms for the get_close interfaces if desired for efficiency or correctness purposes. Examples of this can be seen in the model_mod.f90 for cam or wrf.

An additional new interface, ens_mean_for_model has also been added. This routine is used to pass the ensemble mean state vector into model_mod by the filter before each assimilation step. This allows the model_mod to save this ensemble mean state if it is needed for computing forward operators as in some large atmospheric models (see cam). For low-order models, this interface can be a stub as shown in the template/model_mod.f90.

14. `PBL_1d/create_real_network.f90` originated from create_fixed_network. It uses module_wrf to get obs from smos file, with file, date, and interval controlled via the wrf1d namelist. Note that an obs_def is still required to control which obs are actually written out. Normally, this would be created with create_obs_sequence. This would be run in place of both create_fixed_network and perfect_model_obs.

15. `cam/model_mod.f90`: model_mod can now automatically handle the eulerian and finite volume CAMs (and should handle the Semi-Lagrangian core too), both single threaded and MPI-enabled. The latter enables efficient use of more processors than there are ensemble members. This situation is becoming more common, since DART can now assimilate using smaller ensembles, and massively parallel machines are becoming more common. This new mode of running jobs (async=4) replaces the async=3 and requires only 2 scripts instead of 4.

The multi-core capability required reorganizing the state vector, so new filter_ic files will be necessary. These can be created, from the CAM initial files for the relevant dynamical core, using program trans_sv_pv_time0.

The namelist has changed: state_names_pert has been replaced by pert_names, pert_sd and pert_base_vals as described in the cam/model_mod.html page. highest_obs_pressure_mb, which prevents obs above this height to be assimilated, has been joined by 2 other parameters. Observations on heights (or model levels) will have their vertical location converted to pressure, and be restricted by highest_obs_pressure_mb. highest_state_pressure_mb damps the influence of all obs on state variables above this height. max_obs_lat_degree restricts obs to latitudes less than this parameter, which is needed by some GPS observation sets.

If more fields from the CAM initial files are to be added to the state vector, it may be necessary to add more 'TYPE's in model_mod, and more 'KIND's in DART/obs_kind/DEFAULT_obs_kind_mod.F90, and possibly a new obs_def_ZZZ_mod.f90.

There is a new program, trans_pv_sv_pert0.f90, which can be useful in parameterization studies. It takes a model parameter, which has been added to the CAM initial files, and gives it a spread of values among the filter_ic files that it creates.

16. `wrf/model_mod.f90`: several researchers had their own subtly-different versions of WRF model_mod.f90. These versions have been integrated (assimilated? ;) into one version. The new version performs vertical localization, support for soil parameters, and a host of other features. Hui Liu (DAReS), Altug Aksoy, Yongsheng Chen, and David Dowell of NCAR's MMM group are extensively using this model.

17. `DEFAULT_obs_def_mod.F90`: A new public interface, get_obs_def_key, was added to return the integer key given an obs_def_type. The interface to read_obs_def has an additional intent(inout) argument, obs_val. This is NOT used in the default implementation, but is required for the implementation of certain special observations like radar reflectivity.

17a. `obs_def_radar_mod.f90`: added nyquist velocity metadata field to radial velocity.

17b. `obs_def_QuikSCAT_mod.f90`: New module for the SeaWinds instrument (on the QuikSCAT satellite) data as available in the NCEP BUFR files.

17c. obs_def_reanalysis_bufr_mod.f90: Added land surface and ATOVS temperature and moisture values.

17d. obs_def_GWD_mod.f90: module to define 'observations' of gravity wave drag that are needed to perform parameter estimation studies.

18. DEFAULT_obs_kind_mod.F90: Added in several new raw variable types including KIND_CLOUD_LIQUID_WATER, KIND_CLOUD_ICE, KIND_CONDENSATION_HEATING, KIND_VAPOR_MIXING_RATIO, KIND_ICE_NUMBER_CONCENTRATION, KIND_GEOPOTENTIAL_HEIGHT, KIND_SOIL_MOISTURE, KIND_GRAV_WAVE_DRAG_EFFIC, and KIND_GRAV_WAVE_STRESS_FRACTION.

19. obs_model_mod.f90: There were major internal changes to this routine which implements the shell interface advance of models that is used in the new async = 2 and async = 4 advance options. The public interfaces also changed with the old get_close_states being removed and the advance_state interface being made public. The advance_state interface appears the same except that the intent(in) argument model_size no longer exists. The advance_state interface allows the model to be advanced to a particular target_time without going through the move_ahead interface that uses the observation sequence to drive the advance.

20. merge_obs_seq.f90: This routine is now MUCH faster for both insertions and simple 'appends' and can now handle multiple input files. Conversion between ASCII and binary formats (even for a single file) is now supported. Sorting by time and the removal of unused blocks of observations is also possible.

21. obs_sequence_mod.f90: The obs_sequence_mod presents abstractions for both the obs_sequence and the obs_type.

For the observation sequence, public interfaces delete_seq_head, delete_seq_tail, get_next_obs_from_key and get_prev_obs_from_key were added. Given a time and a sequence, the delete_seq_tail deletes all observations later than the time from the sequence and delete_seq_head deletes all observations earlier than the time. Given a sequence and an integer key, get_next_obs_from_key returns the next observation after the one with 'key' and get_prev_obs_from_key returns the previous observation. A bug in get_obs_time_range that could occur when the entire time range was after the end of the sequence was corrected. A bug that occurred when the only observation in a sequence was deleted with delete_obs_from_seq was corrected.

For the obs_type section, public interfaces replace_obs_values and replace_qc were added. These replace the values of either the observations or the qc fields given a sequence and a key to an observation in that sequence. The interface read_obs had an optional argument, max_obs, added. It allows error checking to make sure that the maximum storage space in the sequence is not exceeded during a read. The value of the observation is now passed as an argument to read_obs_def where it can be used to make observed value dependent modifications to the definition. This is only used at present by the doppler velocity obs_def_mod when it does unfolding of aliased doppler velocities.

22. perfect_model_obs.f90: There were major internal changes to be consistent with the new ensemble_manager_mod and to use a one-line main program that calls a subroutine to avoid lots of shared storage. The namelist has 4 additional arguments, first_obs_days, first_obs_seconds, last_obs_days and last_obs_seconds. These specify times before which and after which observations in the input obs_sequence should be ignored. The default value is -1 which implies that all observations are to be used.

23. random_nr_mod.f90: Converted to use digits12 for real computations to avoid possible change in sequences when reduced precision is used for the r8 kind defined in types_mod.f90.

24. random_seq_mod.f90: Interface init_random_seq was modified to accept an additional optional argument, seed, which is the seed for the sequence if present.

25. utilities_mod.f90: Several modules had duplicate netCDF error checking routines; these have been consolidated into an nc_check() routine in the utilities module. A new set_output() routine can control which tasks in a multi-task MPI job write output messages (warnings and errors are written from any task). The default is for task 0 to write and all others not to. A routine do_output() returns .true. if this task should write messages. This is true by default in a single process job, so user code can always safely write: if (do_output()) write(,) 'informative message' In an MPI job only task 0 will return true and only one copy of the message will appear in the log file or on standard output.

In an MPI job messages written via the error_handler() will prefix the message with the task number. The initialize_utilities() routine now takes an alternative log filename which overrides the default in the input.nml namelist; this allows utility programs to select their own separate log files and avoid conflicts with other DART programs. The MPI initialization and finalize routines call the utility init and finalize routines internally, so programs which use the MPI utilities no longer need to initialize the utilities separately.

26. mpi_utilities_mod.f90: A new module which isolates all calls to the MPI libraries to this one module. Includes interfaces for sending and receiving arrays of data, broadcasts, barriers for synchronization, data reduction (e.g. global sum), and routines for identifying the local task number and total number of tasks. Also contains a block and restart routine for use with the async=4 mode of interacting with a parallel MPI model advance. Programs using this module must generally be compiled with either an MPI wrapper script (usually called mpif90) or with the proper command line flags. Some MPI installations use an include file to define the MPI parameters, others use an F90 module. If the mpi_utilities_mod does not compile as distributed search the source code of this module for the string 'BUILD TIP' for more detailed suggestions on getting it to compile.

When using MPI the call to initialize_mpi_utilities() must be made as close to the start of the execution of the program as possible, and the call to finalize_mpi_utilities() as close to the end of execution as possible. Some implementations of the MPICH library (which is common on Linux clusters) require that MPI be initialized before any I/O is done, and other implementations (SGI in particular) will not allow I/O after MPI is finalized. These routines call the normal utilities init and finalize routines internally, so at the user level only the mpi versions need to be called.

27. `null_mpi_utilities_mod.f90`: A module which has all the same entry points as the mpi_utilities_mod but does not require the MPI library. A program which compiles with this module instead of the real MPI utilities module can only be run with a single task since it cannot do real parallel communication, but does not require the MPI libraries to compile or link. This is the default module – you cannot simultaneously use both the mpi_utilities_mod and the null_mpi_utilities_mod.

28. `mkmf/mkmf`: The mkmf program takes a new -w argument. If specified, the resulting makefile will call 'wrappers' for the fortran compiler and loader. The default compiler and loader are $(FC) and $(LD); with the -w flag they will become $(MPIFC) and $(MPILD). In the mkmf.template file you can then define both the MPI wrappers (generally 'mpif90') and the regular F90 compiler.

29. `mkmf.template.*`: The mkmf.template files have been consolidated where possible and heavily commented to reflect nuances of implementations on different comiler/OS combinations. The habit of appending individual platform names (which led to file creep) is hopefully broken.

30. `input.nml`: All the default input.nml namelists are now easily 'diff'ed against the corresponding input.nml.*_template files. This enhances the ability to determine what values are different than the default values.

31. `DART/shell_scripts/DiffCVS_SVN`: is a new script that identifies differences in two parallel instantiations of source files. I used it so many times during the migration from CVS to SVN that I decided to add it to the DART project. This script compares all the source files in two directories … after removing the trivial differences like the form of the copyright tags (its a new year, you know) and the fact that SVN has different revision numbers than CVS. If you set an environment variable XDIFF to a graphical comparator like 'xdiff' or 'xxdiff' it is used to display the differences between the files. With no arguments, a usage note is printed.

The hope is that you can use it see on your 'old' sandboxes in the directories where you have modified code and compare that to the new code to see if there are any conflicts. The directories do not have to be under CVS or SVN control, by the way.

32. models/PBL_1d/src/*: These source files are 'directly' from other developers and had file extensions that required the use of special compilation flags. There is now a script PBL_1d/shell_scripts/ChangeExtensions.csh that not only changes the file extensions (to something the compilers understand i.e. F90) it also modifies the path_names_* files appropriately. The original files had an extension of .F even though they used F90-like formatting. .F is generally understood to mean the contents of the file abide by the F77 fixed-format syntax … columns 2-5 are for line numbers, column 7 is a line-continuation flag … etc. Now if we can only get them to not rely on the 64bit real autopromotion …

33. DART/matlab: The matlab scripts have experienced no major overhauls, but do handle a few more models than previously. The next release is scheduled to have full matlab support for CAM, and WRF. The ReadASCIIObsSeq.m function has a couple (backwards compatible) tweaks to accomodate some changes in R2006a, R2006b.

The biggest change is the addition of plot_observation_locations.m, which facilitates exploring observation locations (by type) for any 3D (i.e. real-world) observation sequence. This ability is derived by running obs_diag with a namelist variable set such that a matlab-readable dataset is written out.

34. models/ikeda: There is a whole new model - courtesy of Greg Lawson of CalTech. A nice 2-variable system that does not require any fancy time-stepping routines. Thanks Greg!

35. obs_def_dew_point_mod.f90: implements a more robust method (based on Bolton's Approximation) for computing dew point.

6.73.2 These files are obsolete - and have been deleted

1. assim_tools/assim_region.f90: new algorithms and methodologies fundamentally replace the need to manually define and then assimilate regions. Think of all those intermediate files that are not needed!

2. filter_server* are no longer needed because async == 3 is no longer supported (again - replaced by MPI).

3. scripts that advance 'ensembles' are all gone - again because of the MPI implementation. The only script now needed is 'advance_model.csh'.

4. smoother/smoother.f90: The standalone smoother program has become a module and the functionality is now part of the filter program.

6.74 DART "pre_j release" Documentation

6.75 Overview of DART

The Data Assimilation Research Testbed (DART) is designed to facilitate the combination of assimilation algorithms, models, and observation sets to allow increased understanding of all three. The DART programs have been compiled with several Fortran 90 compilers and run on a linux compute-server and linux clusters. You should definitely read the Customizations section.

DART employs a modular programming approach to apply an Ensemble Kalman Filter which nudges models toward a state that is more consistent with information from a set of observations. Models may be swapped in and out, as can different algorithms in the Ensemble Kalman Filter. The method requires running multiple instances of a model to generate an ensemble of states. A forward operator appropriate for the type of observation being used is applied to each of the states to generate the model's estimate of the observation. Comparing these estimates and their uncertainty to the observation and its uncertainty ultimately results in the adjustments to the model states. Sort of. There's more to it, described in detail in the tutorial directory of the package.

DART ultimately creates a few netCDF files containing the model states just before the adjustment `Prior_Diag.nc` and just after the adjustment `Posterior_Diag.nc` as well as a file `obs_seq.final` with the model estimates of the observations. There is a suite of Matlab® functions that facilitate exploration of the results.

The "pre_j" distribution.

The **pre_j** release provides several new models and has a greatly expanded capability for **real** observations which required a fundamentally different implementation of the low-level routines. It is now required to run a preprocessor on several of the program units to construct the source code files which will be compiled by the remaining units. Due to the potentially large number of observations types possible and for portability reasons, the preprocessor is actually a F90 program that uses the namelist mechanism for specifying the observation types to be included. This also prevents having a gory set of compile flags that is different for every compiler. One very clever colleague also 'built a better mousetrap' and figured out how to effectively and robustly read namelists, detect errors, and generate meaningful error messages. HURRAY!

The pre_j release has also been tested with more compilers in an attempt to determine non-portable code elements. It is my experience that the largest impediment to portable code is the reliance on the compiler to autopromote `real` variables to one flavor or another. Using the F90 "kind" allows for much more flexible code, in that the use of interface procedures is possible only when two routines do not have identical sets of input arguments – something that happens when the compiler autopromotes 32bit reals to 64bit reals, for example.

DART programs can require three different types of input. First, some of the DART programs, those for creating synthetic observational datasets, require interactive input from the keyboard. For simple cases, this interactive input can be made directly from the keyboard. In more complicated cases, a file containing the appropriate keyboard input can be created and this file can be directed to the standard input of the DART program. Second, many DART programs expect one or more input files in DART specific formats to be available. For instance, `perfect_model_obs`, which creates a synthetic observation set given a particular model and a description of a sequence of observations, requires an input file that describes this observation sequence. At present, the observation files for DART are in a custom format in either human-readable ascii or more compact machine-specific binary. Third, many DART modules (including main programs) make use of the Fortan90 namelist facility to obtain values of certain parameters at run-time. All programs look for a namelist input file called `input.nml` in the directory in which the program is executed. The `input.nml` file can contain a sequence of individual Fortran90 namelists which specify values of particular parameters for modules that compose the executable program. DART provides a mechanism that automatically generates namelists with the default values for each program to be run.

DART uses the netCDF self-describing data format with a particular metadata convention to describe output that is used to analyze the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. A set of Matlab scripts, designed to produce graphical diagnostics from DART netCDF output files are available. DART users have also used ncview to create rudimentary graphical displays of output data fields. The NCO tools, produced by UCAR's Unidata group, are available to do operations like concatenating, slicing, and dicing of netCDF files.

6.75.1 Requirements: an F90 compiler

The DART software has been successfully built on several Linux/x86 platforms with several versions of the Intel Fortran Compiler for Linux, which (at one point) is/was free for individual scientific use. It has also been built and successfully run with several versions of each of the following: Portland Group Fortran Compiler, Lahey Fortran Compiler, Pathscale Fortran Compiler, Absoft Fortran 90/95 Compiler (Mac OSX). Since recompiling the code is a necessity to experiment with different models, there are no binaries to distribute.

DART uses the netCDF self-describing data format for the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. In particular, DART also makes use of the F90 interface to the library which is available through the `netcdf.mod` and `typesizes.mod` modules. *IMPORTANT*: different compilers create these modules with different "case" filenames, and sometimes they are not **both** installed into the expected directory. It is required that both modules be present. The normal place would be in the `netcdf/include` directory, as opposed to the `netcdf/lib` directory.

If the netCDF library does not exist on your system, you must build it (as well as the F90 interface modules). The library and instructions for building the library or installing from an RPM may be found at the netCDF home page: http://www.unidata.ucar.edu/packages/netcdf/ Pay particular attention to the compiler-specific patches that must be applied for the Intel Fortran Compiler. (Or the PG compiler, for that matter.)

The location of the netCDF library, `libnetcdf.a`, and the locations of both `netcdf.mod` and `typesizes.mod` will be needed by the makefile template, as described in the compiling section.

6.75.2 Unpacking the distribution

The DART source code is distributed as a compressed tar file from our download site. When gunzip'ed and untarred, the source tree will begin with a directory named `DART` and will be approximately 189 Mb. Compiling the code in this tree (as is usually the case) will necessitate much more space.

gunzip `DART_pre_j.tar.gz` tar -xvf `DART_pre_j.tar`

The code tree is very "bushy"; there are many directories of support routines, etc. but only a few directories involved with the customization and installation of the DART software. If you can compile and run ONE of the low-order models, you should be able to compile and run ANY of the low-order models. For this reason, we can focus on the Lorenz `63 model. Subsequently, the only directories with files to be modified to check the installation are: `DART/mkmf`, `DART/models/lorenz_63/work`, and `DART/matlab` (but only for analysis).

6.75.3 Customizing the build scripts – overview

DART executable programs are constructed using two tools: `make` and `mkmf`. The `make` utility is a relatively common piece of software that requires a user-defined input file that records dependencies between different source files. `make` then performs a hierarchy of actions when one or more of the source files is modified. The `mkmf` utility is a custom preprocessor that generates a `make` input file (named `Makefile`) and an example namelist *input.nml.program_default* with the default values. The `Makefile` is designed specifically to work with object-oriented Fortran90 (and other languages) for systems like DART.

`mkmf` requires two separate input files. The first is a `template' file which specifies details of the commands required for a specific Fortran90 compiler and may also contain pointers to directories containing pre-compiled utilities required by the DART system. **This template file will need to be modified to reflect your system**. The second input file is a `path_names' file which includes a complete list of the locations (either relative or absolute) of all Fortran90 source files that are required to produce a particular DART program. Each 'path_names' file must contain a path for exactly one Fortran90 file containing a main program, but may contain any number of additional paths pointing to files containing Fortran90 modules. An `mkmf` command is executed which uses the 'path_names' file and the mkmf template file to produce a `Makefile` which is subsequently used by the standard `make` utility.

Shell scripts that execute the mkmf command for all standard DART executables are provided as part of the standard DART software. For more information on `mkmf` see the FMS mkmf description.

One of the benefits of using `mkmf` is that it also creates an example namelist file for each program. The example namelist is called *input.nml.program_default*, so as not to clash with any exising `input.nml` that may exist in that directory.

Building and customizing the 'mkmf.template' file

A series of templates for different compilers/architectures exists in the `DART/mkmf/` directory and have names with extensions that identify either the compiler, the architecture, or both. This is how you inform the build process of the specifics of your system. Our intent is that you copy one that is similar to your system into `mkmf.template` and customize it. For the discussion that follows, knowledge of the contents of one of these templates (i.e. `mkmf.template.pgf90.ghotiol`) is needed: (note that only the LAST lines are shown here, the head of the file is just a big comment)

```
# Makefile template for PGI f90
FC = pgf90
LD = pgf90
CPPFLAGS =
LIST = -Mlist
NETCDF = /contrib/netcdf-3.5.1-cc-c++-pgif90.5.2-4
FFLAGS = -O0 -Ktrap=fp -pc 64 -I$(NETCDF)/include
LIBS = -L$(NETCDF)/lib -lnetcdf
LDFLAGS = $(LIBS)
...
```

Essentially, each of the lines defines some part of the resulting `Makefile`. Since `make` is particularly good at sorting out dependencies, the order of these lines really doesn't make any difference. The `FC = pgf90` line ultimately defines the Fortran90 compiler to use, etc. The lines which are most likely to need site-specific changes start with `FFLAGS` and `NETCDF`, which indicate where to look for the netCDF F90 modules and the location of the netCDF library and modules.

Netcdf

Modifying the NETCDF value should be relatively straightforward.
Change the string to reflect the location of your netCDF installation containing netcdf.mod and typesizes.mod. The value of the NETCDF variable will be used by the FFLAGS, LIBS, and LDFLAGS variables.

FFLAGS

Each compiler has different compile flags, so there is really no way to exhaustively cover this other than to say the templates as we supply them should work – depending on the location of your netCDF. The low-order models can be compiled without a -r8 switch, but the bgrid_solo model cannot.

Customizing the 'path_names_*' file

Several path_names_* files are provided in the work directory for each specific model, in this case: DART/models/lorenz_63/work.

1. path_names_preprocess
2. path_names_create_obs_sequence
3. path_names_create_fixed_network_seq
4. path_names_perfect_model_obs
5. path_names_filter
6. path_names_obs_diag

Since each model comes with its own set of files, no further customization is needed.

6.75.4 Building the Lorenz_63 DART project

Currently, DART executables are constructed in a work subdirectory under the directory containing code for the given model. In the top-level DART directory, change to the L63 work directory and list the contents:

cd DART/models/lorenz_63/work ls -1

With the result:

```
filter_ics
filter_restart
input.nml
mkmf_create_fixed_network_seq
mkmf_create_obs_sequence
mkmf_filter
mkmf_obs_diag
mkmf_perfect_model_obs
mkmf_preprocess
obs_seq.final
obs_seq.in
obs_seq.out
obs_seq.out.average
obs_seq.out.x
obs_seq.out.xy
```

(continues on next page)

(continued from previous page)

```
obs_seq.out.xyz
obs_seq.out.z
path_names_create_fixed_network_seq
path_names_create_obs_sequence
path_names_filter
path_names_obs_diag
path_names_perfect_model_obs
path_names_preprocess
perfect_ics
perfect_restart
Posterior_Diag.nc
Prior_Diag.nc
set_def.out
True_State.nc
workshop_setup.csh
```

There are six `mkmf_`*xxxxxx* files for the programs `preprocess`, `create_obs_sequence`, `create_fixed_network_seq`, `perfect_model_obs`, `filter`, and `obs_diag` along with the corresponding `path_names_`*xxxxxx* files. You can examine the contents of one of the `path_names_`*xxxxxx* files, for instance `path_names_filter`, to see a list of the relative paths of all files that contain Fortran90 modules required for the program `filter` for the L63 model. All of these paths are relative to your `DART` directory. The first path is the main program (`filter.f90`) and is followed by all the Fortran90 modules used by this program (after preprocessing).

The `mkmf_`*xxxxxx* scripts are cryptic but should not need to be modified – as long as you do not restructure the code tree (by moving directories, for example). The only function of the `mkmf_`*xxxxxx* script is to generate a `Makefile` and an *input.nml.program_default* file. It is not supposed to compile anything:

csh mkmf_preprocess make

The first command generates an appropriate `Makefile` and the `input.nml.preprocess_default` file. The second command results in the compilation of a series of Fortran90 modules which ultimately produces an executable file: `preprocess`. Should you need to make any changes to the `DART/mkmf/mkmf.template`, you will need to regenerate the `Makefile`.

The `preprocess` program actually builds source code to be used by all the remaining modules. It is **imperative** to actually **run** `preprocess` before building the remaining executables. This is how the same code can assimilate state vector 'observations' for the Lorenz_63 model and real radar reflectivities for WRF without needing to specify a set of radar operators for the Lorenz_63 model!

`preprocess` reads the `&preprocess_nml` namelist to determine what observations and operators to incorporate. For this exercise, we will use the values in `input.nml`. `preprocess` is designed to abort if the files it is supposed to build already exist. For this reason, it is necessary to remove a couple files (if they exist) before you run the preprocessor. It is just a good habit to develop.

```
\rm -f ../../../obs_def/obs_def_mod.f90
\rm -f ../../../obs_kind/obs_kind_mod.f90
./preprocess
ls -l ../../../obs_def/obs_def_mod.f90
ls -l ../../../obs_kind/obs_kind_mod.f90
```

This created `../../../obs_def/obs_def_mod.f90` from `../../../obs_kind/DEFAULT_obs_kind_mod.F90` and several other modules.

../../../obs_kind/obs_kind_mod.f90 was created similarly. Now we can build the rest of the project.

A series of object files for each module compiled will also be left in the work directory, as some of these are undoubtedly needed by the build of the other DART components. You can proceed to create the other five programs needed to work with L63 in DART as follows:

```
csh mkmf_create_obs_sequence
make
csh mkmf_create_fixed_network_seq
make
csh mkmf_perfect_model_obs
make
csh mkmf_filter
make
csh mkmf_obs_diag
make
```

The result (hopefully) is that six executables now reside in your work directory. The most common problem is that the netCDF libraries and include files (particularly typesizes.mod) are not found. Edit the DART/mkmf/mkmf.template, recreate the Makefile, and try again.

program	purpose
preprocess	creates custom source code for just the observations of interest
create_obs_sequence	specify a (set) of observation characteristics taken by a particular (set of) instruments
create_fixed_network_seq	specify the temporal attributes of the observation sets
perfect_model_obs	spinup, generate “true state” for synthetic observation experiments, . . .
filter	perform experiments
obs_diag	creates observation-space diagnostic files to be explored by the Matlab® scripts.

6.75.5 Running Lorenz_63

This initial sequence of exercises includes detailed instructions on how to work with the DART code and allows investigation of the basic features of one of the most famous dynamical systems, the 3-variable Lorenz-63 model. The remarkable complexity of this simple model will also be used as a case study to introduce a number of features of a simple ensemble filter data assimilation system. To perform a synthetic observation assimilation experiment for the L63 model, the following steps must be performed (an overview of the process is given first, followed by detailed procedures for each step):

6.75.6 Experiment overview

1. Integrate the L63 model for a long time starting from arbitrary initial conditions to generate a model state that lies on the attractor. The ergodic nature of the L63 system means a ‘lengthy’ integration always converges to some point on the computer’s finite precision representation of the model’s attractor.
2. Generate a set of ensemble initial conditions from which to start an assimilation. Since L63 is ergodic, the ensemble members can be designed to look like random samples from the model’s ‘climatological distribution’. To generate an ensemble member, very small perturbations can be introduced to the state on the attractor generated by step 1. This perturbed state can then be integrated for a very long time until all memory of its initial condition can be viewed as forgotten. Any number of ensemble initial conditions can be generated by repeating this procedure.

3. Simulate a particular observing system by first creating an ‘observation set definition’ and then creating an ‘observation sequence’. The ‘observation set definition’ describes the instrumental characteristics of the observations and the ‘observation sequence’ defines the temporal sequence of the observations.
4. Populate the ‘observation sequence’ with ‘perfect’ observations by integrating the model and using the information in the ‘observation sequence’ file to create simulated observations. This entails operating on the model state at the time of the observation with an appropriate forward operator (a function that operates on the model state vector to produce the expected value of the particular observation) and then adding a random sample from the observation error distribution specified in the observation set definition. At the same time, diagnostic output about the ‘true’ state trajectory can be created.
5. Assimilate the synthetic observations by running the filter; diagnostic output is generated.

1. Integrate the L63 model for a ‘long’ time

`perfect_model_obs` integrates the model for all the times specified in the ‘observation sequence definition’ file. To this end, begin by creating an ‘observation sequence definition’ file that spans a long time. Creating an ‘observation sequence definition’ file is a two-step procedure involving `create_obs_sequence` followed by `create_fixed_network_seq`. After they are both run, it is necessary to integrate the model with `perfect_model_obs`.

1.1 Create an observation set definition

`create_obs_sequence` creates an observation set definition, the time-independent part of an observation sequence. An observation set definition file only contains the `location, type,` and `observational error characteristics` (normally just the diagonal observational error variance) for a related set of observations. There are no actual observations, nor are there any times associated with the definition. For spin-up, we are only interested in integrating the L63 model, not in generating any particular synthetic observations. Begin by creating a minimal observation set definition.

In general, for the low-order models, only a single observation set need be defined. Next, the number of individual scalar observations (like a single surface pressure observation) in the set is needed. To spin-up an initial condition for the L63 model, only a single observation is needed. Next, the error variance for this observation must be entered. Since we do not need (nor want) this observation to have any impact on an assimilation (it will only be used for spinning up the model and the ensemble), enter a very large value for the error variance. An observation with a very large error variance has essentially no impact on deterministic filter assimilations like the default variety implemented in DART. Finally, the location and type of the observation need to be defined. For all types of models, the most elementary form of synthetic observations are called ‘identity’ observations. These observations are generated simply by adding a random sample from a specified observational error distribution directly to the value of one of the state variables. This defines the observation as being an identity observation of the first state variable in the L63 model. The program will respond by terminating after generating a file (generally named `set_def.out`) that defines the single identity observation of the first state variable of the L63 model. The following is a screenshot (much of the verbose logging has been left off for clarity), the user input looks *like this*.

```
[unixprompt]$ ./create_obs_sequence
Initializing the utilities module.
Trying to read from unit          10
Trying to open file dart_log.out

Registering module :
$source: /home/dart/CVS.REPOS/DART/utilities/utilities_mod.f90,v $
$revision: 1.18 $
$date: 2004/06/29 15:16:40 $
```

(continues on next page)

(continued from previous page)

```
Registration complete.

&UTILITIES_NML
TERMLEVEL= 2,LOGFILENAME=dart_log.out

/

Registering module :
$source: /home/dart/CVS.REPOS/DART/obs_sequence/create_obs_sequence.f90,v $
$revision: 1.18 $
$date: 2004/05/24 15:41:46 $
Registration complete.

{ ... }

Input upper bound on number of observations in sequence
10

Input number of copies of data (0 for just a definition)
0

Input number of quality control values per field (0 or greater)
0

Input a -1 if there are no more obs
0

Registering module :
$source$
$revision: 3169 $
$date: 2007-12-07 16:40:53 -0700 (Fri, 07 Dec 2007) $
Registration complete.


Registering module :
$source$
$revision: 3169 $
$date: 2007-12-07 16:40:53 -0700 (Fri, 07 Dec 2007) $
Registration complete.

initialize_module obs_kind_nml values are

-------------- ASSIMILATE_THESE_OBS_TYPES --------------
RAW_STATE_VARIABLE
-------------- EVALUATE_THESE_OBS_TYPES --------------
------------------------------------------------------


   Input -1 * state variable index for identity observations
   OR input the name of the observation kind from table below:
   OR input the integer index, BUT see documentation...
               1 RAW_STATE_VARIABLE

-1

Input time in days and seconds
1 0
```

(continues on next page)

(continued from previous page)

```
Input error variance for this observation definition
1000000

Input a -1 if there are no more obs
-1

Input filename for sequence (  set_def.out   usually works well)
set_def.out
write_obs_seq  opening formatted file set_def.out
write_obs_seq  closed file set_def.out
```

1.2 Create an observation sequence definition

`create_fixed_network_seq` creates an 'observation sequence definition' by extending the 'observation set definition' with the temporal attributes of the observations.

The first input is the name of the file created in the previous step, i.e. the name of the observation set definition that you've just created. It is possible to create sequences in which the observation sets are observed at regular intervals or irregularly in time. Here, all we need is a sequence that takes observations over a long period of time - indicated by entering a 1. Although the L63 system normally is defined as having a non-dimensional time step, the DART system arbitrarily defines the model timestep as being 3600 seconds. If we declare that we have one observation per day for 1000 days, we create an observation sequence definition spanning 24000 'model' timesteps; sufficient to spin-up the model onto the attractor. Finally, enter a name for the 'observation sequence definition' file. Note again: there are no observation values present in this file. Just an observation type, location, time and the error characteristics. We are going to populate the observation sequence with the `perfect_model_obs` program.

```
[unixprompt]$ ./create_fixed_network_seq

...

Registering module :
$source: /home/dart/CVS.REPOS/DART/obs_sequence/obs_sequence_mod.f90,v $
$revision: 1.31 $
$date: 2004/06/29 15:04:37 $
Registration complete.

Input filename for network definition sequence (usually  set_def.out  )
set_def.out

...

To input a regularly repeating time sequence enter 1
To enter an irregular list of times enter 2
1
Input number of observations in sequence
1000
Input time of initial ob in sequence in days and seconds
1, 0
Input period of obs in days and seconds
1, 0
           1
           2
           3
```

(continues on next page)

(continued from previous page)

```
...
         997
         998
         999
        1000
What is output file name for sequence (  obs_seq.in   is recommended )
obs_seq.in
write_obs_seq  opening formatted file obs_seq.in
write_obs_seq closed file [blah blah blah]/work/obs_seq.in
```

1.3 Initialize the model onto the attractor

perfect_model_obs can now advance the arbitrary initial state for 24,000 timesteps to move it onto the attractor.

perfect_model_obs uses the Fortran90 namelist input mechanism instead of (admittedly gory, but temporary) interactive input. All of the DART software expects the namelists to found in a file called input.nml. When you built the executable, an example namelist was created input.nml.perfect_model_obs_default that contains all of the namelist input for the executable. If you followed the example, each namelist was saved to a unique name. We must now rename and edit the namelist file for perfect_model_obs. Copy input.nml.perfect_model_obs_default to input.nml and edit it to look like the following: (just worry about the highlighted stuff)

```
&perfect_model_obs_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .false.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1 /

&ensemble_manager_nml
   in_core = .true.,
   single_restart_file_in = .true.,
   single_restart_file_out = .true. /

&assim_tools_nml
   filter_kind = 1,
   cutoff = 0.2,
   sort_obs_inc = .false.,
   cov_inflate = -1.0,
   cov_inflate_sd = 0.05,
   sd_lower_bound = 0.05,
   deterministic_cov_inflate = .true.,
   start_from_assim_restart = .false.,
   assim_restart_in_file_name =
'assim_tools_ics',
   assim_restart_out_file_name =
'assim_tools_restart',
```

(continues on next page)

(continued from previous page)

```
   do_parallel = 0,
   num_domains = 1
   parallel_command = "./assim_filter.csh",
   spread_restoration = .false.,
   cov_inflate_upper_bound = 10000000.0,
   internal_outlier_threshold = -1.0 /

&cov_cutoff_nml
   select_localization = 1 /

&reg_factor_nml
   select_regression = 1,
   input_reg_file = "time_mean_reg"
   save_reg_diagnostics = .false.,
   reg_diagnostics_file = 'reg_diagnostics' /

&obs_sequence_nml
   write_binary_obs_sequence = .false. /

&obs_kind_nml
   assimilate_these_obs_types = 'RAW_STATE_VARIABLE' /

&assim_model_nml
   write_binary_restart_files = .true. /

&model_nml
   sigma = 10.0,
   r = 28.0,
   b = 2.6666666666667,
   deltat = 0.01,
   time_step_days = 0,
   time_step_seconds = 3600 /

&utilities_nml
   TERMLEVEL = 1
   logfilename = 'dart_log.out' /
```

For the moment, only two namelists warrant explanation. Each namelists is covered in detail in the html files accompanying the source code for the module.

perfect_model_obs_nml

namelist variable	description
`async`	For the lorenz_63, simply ignore this. Leave it set to '0'
`advance_ens_command`	specifies the shell commands or script to execute when async /= 0
`obs_seq_in_file_name`	specifies the file name that results from running `create_fixed_network_seq`, i.e. the 'observation sequence definition' file.
`obs_seq_out_file_name`	specifies the output file name containing the 'observation sequence', finally populated with (perfect?) 'observations'.
`start_from_restart`	When set to 'false', `perfect_model_obs` generates an arbitrary initial condition (which cannot be guaranteed to be on the L63 attractor).
`output_restart`	When set to 'true', `perfect_model_obs` will record the model state at the end of this integration in the file named by `restart_out_file_name`.
`restart_in_file_name`	is ignored when 'start_from_restart' is 'false'.
`restart_out_file_name`	if `output_restart` is 'true', this specifies the name of the file containing the model state at the end of the integration.
`init_time_`*xxxx*	the start time of the integration.
`output_interval`	interval at which to save the model state.

utilities_nml

namelist variable	description
`TERMLEVEL`	When set to '1' the programs terminate when a 'warning' is generated. When set to '2' the programs terminate only with 'fatal' errors.
`logfilename`	Run-time diagnostics are saved to this file. This namelist is used by all programs, so the file is opened in APPEND mode. Subsequent executions cause this file to grow.

Executing `perfect_model_obs` will integrate the model 24,000 steps and output the resulting state in the file `perfect_restart`. Interested parties can check the spinup in the `True_State.nc` file.

perfect_model_obs

2. Generate a set of ensemble initial conditions

The set of initial conditions for a 'perfect model' experiment is created in several steps. 1) Starting from the spun-up state of the model (available in `perfect_restart`), run `perfect_model_obs` to generate the 'true state' of the experiment and a corresponding set of observations. 2) Feed the same initial spun-up state and resulting observations into `filter`.

The first step is achieved by changing a perfect_model_obs namelist parameter, copying `perfect_restart` to `perfect_ics`, and rerunning `perfect_model_obs`. This execution of `perfect_model_obs` will advance the model state from the end of the first 24,000 steps to the end of an additional 24,000 steps and place the final state in `perfect_restart`. The rest of the namelists in `input.nml` should remain unchanged.

```
&perfect_model_obs_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
```

(continues on next page)

(continued from previous page)

```
   start_from_restart = .true.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1
/
```

cp perfect_restart perfect_ics perfect_model_obs

A `True_State.nc` file is also created. It contains the 'true' state of the integration.

Generating the ensemble

This step (#2 from above) is done with the program `filter`, which also uses the Fortran90 namelist mechanism for input. It is now necessary to copy the `input.nml.filter_default` namelist to `input.nml` or you may simply insert the `filter_nml` namelist block into the existing `input.nml`. Having the `perfect_model_obs` namelist in the input.nml does not hurt anything. In fact, I generally create a single `input.nml` that has all the namelist blocks in it. I simply copied the filter namelist block from `input.nml.filter_default` and inserted it into our `input.nml` for the following example.

```
&perfect_model_obs_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .true.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1 /

&filter_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   ens_size = 100,
   cov_inflate = 1.0,
   start_from_restart = .false.,
   output_restart = .true.,
   obs_sequence_in_name = "obs_seq.out",
   obs_sequence_out_name = "obs_seq.final",
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "filter_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_state_ens_mean = .true.,
   output_state_ens_spread = .true.,
   output_obs_ens_mean = .true.,
   output_obs_ens_spread = .true.,
   num_output_state_members = 20,
```

(continues on next page)

(continued from previous page)

```
   num_output_obs_members = 20,
   output_interval = 1,
   num_groups = 1,
   outlier_threshold = -1.0 /


&ensemble_manager_nml
   in_core = .true.,
   single_restart_file_in = .true.,
   single_restart_file_out = .true. /

&assim_tools_nml
   filter_kind = 1,
   cutoff = 0.2,
   sort_obs_inc = .false.,
   cov_inflate = -1.0,
   cov_inflate_sd = 0.05,
   sd_lower_bound = 0.05,
   deterministic_cov_inflate = .true.,
   start_from_assim_restart = .false.,
   assim_restart_in_file_name =
'assim_tools_ics',
   assim_restart_out_file_name =
'assim_tools_restart',
   do_parallel = 0,
   num_domains = 1
   parallel_command = "./assim_filter.csh",
   spread_restoration = .false.,
   cov_inflate_upper_bound = 10000000.0,
   internal_outlier_threshold = -1.0 /

&cov_cutoff_nml
   select_localization = 1 /

&reg_factor_nml
   select_regression = 1,
   input_reg_file = "time_mean_reg"
   save_reg_diagnostics = .false.,
   reg_diagnostics_file = 'reg_diagnostics' /

&obs_sequence_nml
   write_binary_obs_sequence = .false. /

&obs_kind_nml
   assimilate_these_obs_types = 'RAW_STATE_VARIABLE'
/

&assim_model_nml
   write_binary_restart_files = .true. /

&model_nml
   sigma = 10.0,
   r = 28.0,
   b = 2.6666666666667,
   deltat = 0.01,
   time_step_days = 0,
   time_step_seconds = 3600 /
```

(continues on next page)

(continued from previous page)

```

&utilities_nml
   TERMLEVEL = 1
   logfilename = 'dart_log.out' /
```

Only the non-obvious(?) entries for `filter_nml` will be discussed.

namelist variable	description
`ens_size`	Number of ensemble members. 100 is sufficient for most of the L63 exercises.
`cov_inflate`	A value of 1.0 results in no inflation.(spin-up)
`start_from_restart`	when '.false.', `filter` will generate its own ensemble of initial conditions. It is important to note that the filter still makes use of `perfect_ics` by randomly perturbing these state variables.
`output_state_ens_mean`	when '.true.' the mean of all ensemble members is output.
`output_state_ens_spread`	when '.true.' the spread of all ensemble members is output.
`num_output_state_members`	may be a value from 0 to `ens_size`
`output_obs_ens_mean`	when '.true.' Output ensemble mean in observation output file.
`output_obs_ens_spread`	when '.true.' Output ensemble spread in observation output file.
`num_output_obs_members`	may be a value from 0 to `ens_size`
`output_interval`	The frequency with which output state diagnostics are written. Units are in assimilation times. Default value is 1 meaning output is written at every observation time

The filter is told to generate its own ensemble initial conditions since `start_from_restart` is '.false.'. However, it is important to note that the filter still makes use of `perfect_ics` which is set to be the `restart_in_file_name`. This is the model state generated from the first 24,000 step model integration by `perfect_model_obs`. `Filter` generates its ensemble initial conditions by randomly perturbing the state variables of this state.

The arguments `output_state_ens_mean` and `output_state_ens_spread` are '.true.' so that these quantities are output at every time for which there are observations (once a day here) and `num_output_ens_members` means that the same diagnostic files, `Posterior_Diag.nc` and `Prior_Diag.nc` also contain values for 20 ensemble members once a day. Once the namelist is set, execute `filter` to integrate the ensemble forward for 24,000 steps with the final ensemble state written to the `filter_restart`. Copy the `perfect_model_obs` restart file `perfect_restart` (the `true state') to `perfect_ics`, and the `filter` restart file `filter_restart` to `filter_ics` so that future assimilation experiments can be initialized from these spun-up states.

filter cp perfect_restart perfect_ics cp filter_restart filter_ics

The spin-up of the ensemble can be viewed by examining the output in the netCDF files `True_State.nc` generated by `perfect_model_obs` and `Posterior_Diag.nc` and `Prior_Diag.nc` generated by `filter`. To do this, see the detailed discussion of matlab diagnostics in Appendix I.

3. Simulate a particular observing system

Begin by using `create_obs_sequence` to generate an observation set in which each of the 3 state variables of L63 is observed with an observational error variance of 1.0 for each observation. To do this, use the following input sequence (the text including and after # is a comment and does not need to be entered):

4	# upper bound on num of observations in sequence
0	# number of copies of data (0 for just a definition)
0	# number of quality control values per field (0 or greater)
0	# -1 to exit/end observation definitions
-1	# observe state variable 1
0 0	# time – days, seconds
1.0	# observational variance
0	# -1 to exit/end observation definitions
-2	# observe state variable 2
0 0	# time – days, seconds
1.0	# observational variance
0	# -1 to exit/end observation definitions
-3	# observe state variable 3
0 0	# time – days, seconds
1.0	# observational variance
-1	# -1 to exit/end observation definitions
set_def.out	# Output file name

Now, generate an observation sequence definition by running `create_fixed_network_seq` with the following input sequence:

set_def.out	# Input observation set definition file
1	# Regular spaced observation interval in time
1000	# 1000 observation times
0, 43200	# First observation after 12 hours (0 days, 12 * 3600 seconds)
0, 43200	# Observations every 12 hours
obs_seq.in	# Output file for observation sequence definition

4. Generate a particular observing system and true state

An observation sequence file is now generated by running `perfect_model_obs` with the namelist values (unchanged from step 2):

```
&perfect_model_obs_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   obs_seq_in_file_name = "obs_seq.in",
   obs_seq_out_file_name = "obs_seq.out",
   start_from_restart = .true.,
   output_restart = .true.,
   restart_in_file_name = "perfect_ics",
   restart_out_file_name = "perfect_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_interval = 1 /
```

This integrates the model starting from the state in perfect_ics for 1000 12-hour intervals outputting synthetic observations of the three state variables every 12 hours and producing a netCDF diagnostic file, True_State.nc.

5. Filtering

Finally, filter can be run with its namelist set to:

```
&filter_nml
   async = 0,
   adv_ens_command = "./advance_ens.csh",
   ens_size = 100,
   cov_inflate = 1.0,
   start_from_restart = .true.,
   output_restart = .true.,
   obs_sequence_in_name = "obs_seq.out",
   obs_sequence_out_name = "obs_seq.final",
   restart_in_file_name = "filter_ics",
   restart_out_file_name = "filter_restart",
   init_time_days = 0,
   init_time_seconds = 0,
   output_state_ens_mean = .true.,
   output_state_ens_spread = .true.,
   output_obs_ens_mean = .true.,
   output_obs_ens_spread = .true.,
   num_output_state_members = 20,
   num_output_obs_members = 20,
   output_interval = 1,
   num_groups = 1,
   outlier_threshold = -1.0 /
```

filter produces two output diagnostic files, Prior_Diag.nc which contains values of the ensemble mean, ensemble spread, and ensemble members for 12- hour lead forecasts before assimilation is applied and Posterior_Diag.nc which contains similar data for after the assimilation is applied (sometimes referred to as analysis values).

Now try applying all of the matlab diagnostic functions described in the Matlab Diagnostics section.

6.75.7 Matlab® diagnostics

The output files are netCDF files, and may be examined with many different software packages. We happen to use Matlab®, and provide our diagnostic scripts in the hopes that they are useful.

The diagnostic scripts and underlying functions reside in two places: DART/diagnostics/matlab and DART/matlab. They are reliant on the public-domain netcdf toolbox from http://woodshole.er.usgs.gov/staffpages/cdenham/public_html/MexCDF/nc4ml5.html as well as the public-domain CSIRO matlab/netCDF interface from http://www.marine.csiro.au/sw/matlab-netcdf.html. If you do not have them installed on your system and want to use Matlab to peruse netCDF, you must follow their installation instructions. The 'interested reader' may want to look at the DART/matlab/startup.m file I use on my system. If you put it in your $HOME/matlab directory, it is invoked every time you start up Matlab.

Once you can access the getnc function from within Matlab, you can use our diagnostic scripts. It is necessary to prepend the location of the DART/matlab scripts to the matlabpath. Keep in mind the location of the netcdf operators on your system WILL be different from ours . . . and that's OK.

```
0[269]0 ghotiol:/<5>models/lorenz_63/work]$ matlab -nojvm

                                             < M A T L A B >
                                 Copyright 1984-2002 The MathWorks, Inc.
                                     Version 6.5.0.180913a Release 13
                                               Jun 18 2002

  Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.

  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

>> which getnc
/contrib/matlab/matlab_netcdf_5_0/getnc.m
>>ls *.nc

ans =

Posterior_Diag.nc  Prior_Diag.nc  True_State.nc


>>path('../../../matlab',path)
>>path('../../../diagnostics/matlab',path)
>>which plot_ens_err_spread
../../../matlab/plot_ens_err_spread.m
>>help plot_ens_err_spread

  DART : Plots summary plots of the ensemble error and ensemble spread.
                         Interactively queries for the needed information.
                         Since different models potentially need different
                         pieces of information ... the model types are
                         determined and additional user input may be queried.

  Ultimately, plot_ens_err_spread will be replaced by a GUI.
  All the heavy lifting is done by PlotEnsErrSpread.

  Example 1 (for low-order models)

  truth_file = 'True_State.nc';
  diagn_file = 'Prior_Diag.nc';
  plot_ens_err_spread

>>plot_ens_err_spread
```

And the matlab graphics window will display the spread of the ensemble error for each state variable. The scripts are designed to do the "obvious" thing for the low-order models and will prompt for additional information if needed. The philosophy of these is that anything that starts with a lower-case *plot_some_specific_task* is intended to be user-callable and should handle any of the models. All the other routines in `DART/matlab` are called BY the high-level routines.

Matlab script	description
`plot_bins`	plots ensemble rank histograms
`plot_correl`	Plots space-time series of correlation between a given variable at a given time and other variables at all times in a n ensemble time sequence.
`plot_ens_err_spread`	Plots summary plots of the ensemble error and ensemble spread. Interactively queries for the needed information. Since different models potentially need different pieces of information … the model types are determined and additional user input may be queried.
`plot_ens_mean_time_series`	Queries for the state variables to plot.
`plot_ens_time_series`	Queries for the state variables to plot.
`plot_phase_space`	Plots a 3D trajectory of (3 state variables of) a single ensemble member. Additional trajectories may be superimposed.
`plot_total_err`	Summary plots of global error and spread.
`plot_var_var_correl`	Plots time series of correlation between a given variable at a given time and another variable at all times in an ensemble time sequence.

6.75.8 Bias, filter divergence and covariance inflation (with the l63 model)

One of the common problems with ensemble filters is filter divergence, which can also be an issue with a variety of other flavors of filters including the classical Kalman filter. In filter divergence, the prior estimate of the model state becomes too confident, either by chance or because of errors in the forecast model, the observational error characteristics, or approximations in the filter itself. If the filter is inappropriately confident that its prior estimate is correct, it will then tend to give less weight to observations than they should be given. The result can be enhanced overconfidence in the model's state estimate. In severe cases, this can spiral out of control and the ensemble can wander entirely away from the truth, confident that it is correct in its estimate. In less severe cases, the ensemble estimates may not diverge entirely from the truth but may still be too confident in their estimate. The result is that the truth ends up being farther away from the filter estimates than the spread of the filter ensemble would estimate. This type of behavior is commonly detected using rank histograms (also known as Talagrand diagrams). You can see the rank histograms for the L63 initial assimilation by using the matlab script `plot_bins`.

A simple, but surprisingly effective way of dealing with filter divergence is known as covariance inflation. In this method, the prior ensemble estimate of the state is expanded around its mean by a constant factor, effectively increasing the prior estimate of uncertainty while leaving the prior mean estimate unchanged. The program `filter` has a namelist parameter that controls the application of covariance inflation, `cov_inflate`. Up to this point, `cov_inflate` has been set to 1.0 indicating that the prior ensemble is left unchanged. Increasing `cov_inflate` to values greater than 1.0 inflates the ensemble before assimilating observations at each time they are available. Values smaller than 1.0 contract (reduce the spread) of prior ensembles before assimilating.

You can do this by modifying the value of `cov_inflate` in the namelist, (try 1.05 and 1.10 and other values at your discretion) and run the filter as above. In each case, use the diagnostic matlab tools to examine the resulting changes to the error, the ensemble spread (via rank histogram bins, too), etc. What kind of relation between spread and error is seen in this model?

6.75.9 Synthetic observations

Synthetic observations are generated from a `perfect' model integration, which is often referred to as the `truth' or a `nature run'. A model is integrated forward from some set of initial conditions and observations are generated as $y = H(x) + e$ where H is an operator on the model state vector, x, that gives the expected value of a set of observations, y, and e is a random variable with a distribution describing the error characteristics of the observing instrument(s) being simulated. Using synthetic observations in this way allows students to learn about assimilation algorithms while being isolated from the additional (extreme) complexity associated with model error and unknown observational error characteristics. In other words, for the real-world assimilation problem, the model has (often substantial) differences from what happens in the real system and the observational error distribution may be very complicated and is certainly not well known. Be careful to keep these issues in mind while exploring the capabilities of the ensemble filters with synthetic observations.

6.76 DART POST-Iceland revisions

6.77 DART "POST Iceland release" summary of changes.

The DART POST Iceland release (16 June 2006) is an intermediate release designed to provide error fixes required for a number of users and to provide enhanced support for the CAM, WRF, and PBL_1D models. It also provides several new assimilation algorithm options that are already widely used by CAM implementations. The most important are additional methods for doing adaptive inflation. The previously existing observation space inflation is retained and two new options for adaptive state space inflation are introduced. One option has a value of inflation that is constant across all state variables but varies in time. The second has an inflation factor for EACH state variable and these can each vary in time. An initial version of an algorithm to do adaptive localization is also introduced. The local density of observations is measured by computing the number of observations from a set at a given time that is close to the observation being assimilated. If the density is such that more than a given threshold number of observations is close to the current observation, the localization radius is adjusted so that the expected number of observations in the adjusted radius is equal to the threshold value. This will decrease the localization radius in areas with dense observations.

Another important new capability is provided by a `merge_obs_sequence` program that can combine most existing observation sequences into a single sequence.

The *shell scripts* used to coordinate the execution of the model advances or regional assimilation have been modified to be more robust and easier to read. Furthermore, each script has been standardized to be much more machine/queueing system independent. Each script should now be able to be used interactively (in a debug scenario), with the LSF queueing system (as a batch job), or with the PBS queueing system. The scripts have a block of logic that converts the system-specific variables to ones commonly used throughout the DART documentation. This has now made it possible to prepare a script that will conditionally execute a series of batch jobs wherein each batch job will process exactly one observation sequence file. Only upon successful completion of the previous batch job with the subsequent batch job be started. Look for information on `Qfiller.csh` on the DART www page.

`merge_obs_sequence` program that can combine most existing observation sequences into a single sequence.

An initial version of an ensemble smoother was added to DART and implemented with the low-order models and the bgrid model. The smoother continues to use the OLD inflation options and an old version of the `assim_tools_mod` module.

6.77.1 Changes

A summary list of changes occurring in each DART directory/file follows:

test_dart.csh	This testing script in the main DART directory has been updated to test the new inflation options with the new scripts. An attempt has been made to preserve the input environment such that if you wanted to run it twice in a row, you could. It now stores all the run-time output of the lorenz_96 tests in the `lorenz_96/work/xxxx` directory, where xxx is now an argument to `test_dart.csh`. Simply typing the file name will now echo usage notes.
adaptive_inflate_mod.f90	This new module now handles all the computations for the adaptive inflation computations. It has the code that was previously in `assim_tools` to do the Bayesian update of observation space inflation. It also provides the additional algorithms required to do state space inflation. Documentation is provided in the module and in several papers on the DART home page.
assim_tools_mod.f90	Uses `adaptive_inflation` module. Namelist modified (see below). Storage for observation space inflation for regions no longer needed. No longer an `assim_restart` file; all restart info is now stored in `adaptive_inflate`. Routine `update_inflation` replaces `bayes_cov_inflate` which has been moved to `adaptive_inflate_mod`. Correlation is now passed out of `update_from_obs_inc` for use with adaptive inflation updates. Ability to do sampling_error_correction from a file using `correl_error.f90` in `system_simulation`. This is turned on by namelist parameter `sampling_error_correction` and requires appropriate error correction files for a given ensemble size (this is still in preliminary testing phase). The inflation values are now passed into `filter_assim_region`. The mean and variance of the observation space priors are computed up front in `filter_assim_region` for use with adaptive inflation algorithm. Observational density thinning controlled by namelist parameter `num_close_threshold` implemented. If number of observations close to a given observation is larger than the cutoff, the localization cutoff is adjusted to try to make the number close the same as the cutoff. This is a fundamentally two-d algorithm in this naive implementation. Update computations for spatial inflation are included in `filter_assim_region`. Routine `filter_assim` also gets the inflation values as arguments. Copying of these inflation values for the shell driven files is added. Previously commented `print_regional_results` is deleted (produced errors with absoft compilers). Routine `comp_correl` added to compute correlations. Routine `get_correction_from_file` added to support `sampling_error_correction`. No longer need routine `assim_tools_end` since there is no longer a restart requirement. An error in the correlation computation for assimilation was removed. If all prior values of an observation were identical, a `NaN` could result from the old correlation computation. The `assim_tools_nml` was modified as follows: Removed `cov_inflate, cov_inflate_sd, sd_lower_bound, deterministic_cov_inflate, start_from_assim_restart, assim_restart_in_file_name, assim_restart_out_file_name, cov_inflate_upper_bound`. Added `sampling_error_correction` and `num_close_threshold`.
assim_tools_smoother.f90	The smoother is still using the previous version of inflation and the corresponding `assim_tools module` which has been renamed as `assim_tools_smoother`. Additional routines have been added to support the updates required to do smoothing (see documentation in smoother directory).
diagnostics	Observation space diagnostics `obs_diag` were modified to support observations on model levels. This is particularly useful for "perfect model" experiments. Direct observations of model variables (identity observations) are *not* supported, as this is an exploration of state-space, not observation-space. However, since the advent of `merge_obs_sequence` (see below) an observation sequence file *may* contain synthetic and real observations. Any identity observations are skipped. All other observations may be considered.
ensemble_manager_mod.f90	Modified to be able to handle multiple ensembles at one time to support the smoother which has an ensemble for each lag-time in the smoothing.
filter.f90	The `adaptive_inflate` module is now used and the namelist entry `cov_inflate` has been removed from `filter_nml`. Inflation is now done with `filter_ensemble_inflate` only if constant or varying spatial inflation is selected in the `adaptive_inflate` namelist. Information about state space inflation is passed to `filter_assim` as arguments. The call to `assim_tools_end` has been replace by `adaptive_inflate_end` which creates restarts for adaptive inflation. For spatially varying state inflation, two extra fields are tacked onto the state space diagnostic netcdf files to record the inflation mean and standard deviation. At present, inflation is done for the whole state at once; this may be very inefficient and should be examined. The entry `cov_inflate` was removed from the namelist.
merge_obs_seq.f90	This is a fundamentally new program to DART. This routine can combine any two observation sequence files that are compatible. The files are deemed compatible if the 'copies' of the observations and the QC fields are *identical* between the two sequences. If one observation sequence file has only an ensemble mean and spread, the other observation sequence file can have *only* an ensemble mean and spread – it cannot additionally have the N ensemble member estimates of the observation. Most of the time, this routine is envisioned to be used to combine `obs_seq.out` files (as opposed to `obs_seq.final` files). If the two sequences temporally overlap, it is faster to put the shorter sequence as `filenam_seq2`, the insertion sort

6.78 DART "Iceland release" Documentation

6.79 DART "post-Iceland release" Documentation

6.80 Overview of DART

The Data Assimilation Research Testbed (DART) is designed to facilitate the combination of assimilation algorithms, models, and observation sets to allow increased understanding of all three. The DART programs have been compiled with several Fortran 90 compilers and run on a linux compute-server and linux clusters. You should definitely read the Customizations section.

DART employs a modular programming approach to apply an Ensemble Kalman Filter which nudges models toward a state that is more consistent with information from a set of observations. Models may be swapped in and out, as can different algorithms in the Ensemble Kalman Filter. The method requires running multiple instances of a model to generate an ensemble of states. A forward operator appropriate for the type of observation being used is applied to each of the states to generate the model's estimate of the observation. Comparing these estimates and their uncertainty to the observation and its uncertainty ultimately results in the adjustments to the model states. Sort of. There's more to it, described in detail in the tutorial directory of the package.

DART ultimately creates a few netCDF files containing the model states just before the adjustment `Prior_Diag.nc` and just after the adjustment `Posterior_Diag.nc` as well as a file `obs_seq.final` with the model estimates of the observations. There is a suite of Matlab® functions that facilitate exploration of the results.

The latest software release – "post_iceland".

The **post_iceland** release provides several new models and has a greatly expanded capability for **real** observations which required a fundamentally different implementation of the low-level routines. It is now required to run a preprocessor on several of the program units to construct the source code files which will be compiled by the remaining units. Due to the potentially large number of observations types possible and for portability reasons, the preprocessor is actually a F90 program that uses the namelist mechanism for specifying the observation types to be included. This also prevents having a gory set of compile flags that is different for every compiler. One very clever colleague also 'built a better mousetrap' and figured out how to effectively and robustly read namelists, detect errors, and generate meaningful error messages. HURRAY!

The post_iceland release has also been tested with more compilers in an attempt to determine non-portable code elements. It is my experience that the largest impediment to portable code is the reliance on the compiler to autopromote `real` variables to one flavor or another. Using the F90 "kind" allows for much more flexible code, in that the use of interface procedures is possible only when two routines do not have identical sets of input arguments – something that happens when the compiler autopromotes 32bit reals to 64bit reals, for example.

DART programs can require three different types of input. First, some of the DART programs, those for creating synthetic observational datasets, require interactive input from the keyboard. For simple cases, this interactive input can be made directly from the keyboard. In more complicated cases, a file containing the appropriate keyboard input can be created and this file can be directed to the standard input of the DART program. Second, many DART programs expect one or more input files in DART specific formats to be available. For instance, `perfect_model_obs`, which creates a synthetic observation set given a particular model and a description of a sequence of observations, requires an input file that describes this observation sequence. At present, the observation files for DART are in a custom format in either human-readable ascii or more compact machine-specific binary. Third, many DART modules (including main programs) make use of the Fortan90 namelist facility to obtain values of certain parameters at run-time. All programs look for a namelist input file called `input.nml` in the directory in which the program is executed. The `input.nml` file can contain a sequence of individual Fortran90 namelists which specify values of particular parameters for modules that compose the executable program. DART provides a mechanism that automatically generates namelists with the default values for each program to be run.

DART uses the netCDF self-describing data format with a particular metadata convention to describe output that is used to analyze the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. A set of Matlab scripts, designed to produce graphical diagnostics from DART netCDF output files are available. DART users have also used ncview to create rudimentary graphical displays of output data fields. The NCO tools, produced by UCAR's Unidata group, are available to do operations like concatenating, slicing, and dicing of netCDF files.

6.80.1 Requirements: an F90 compiler

The DART software has been successfully built on several Linux/x86 platforms with several versions of the Intel Fortran Compiler for Linux, which (at one point) is/was free for individual scientific use. It has also been built and successfully run with several versions of each of the following: Portland Group Fortran Compiler, Lahey Fortran Compiler, Pathscale Fortran Compiler, Absoft Fortran 90/95 Compiler (Mac OSX). Since recompiling the code is a necessity to experiment with different models, there are no binaries to distribute.

DART uses the netCDF self-describing data format for the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. In particular, DART also makes use of the F90 interface to the library which is available through the `netcdf.mod` and `typesizes.mod` modules. *IMPORTANT*: different compilers create these modules with different "case" filenames, and sometimes they are not **both** installed into the expected directory. It is required that both modules be present. The normal place would be in the `netcdf/include` directory, as opposed to the `netcdf/lib` directory.

If the netCDF library does not exist on your system, you must build it (as well as the F90 interface modules). The library and instructions for building the library or installing from an RPM may be found at the netCDF home page: http://www.unidata.ucar.edu/packages/netcdf/ Pay particular attention to the compiler-specific patches that must be applied for the Intel Fortran Compiler. (Or the PG compiler, for that matter.)

The location of the netCDF library, `libnetcdf.a`, and the locations of both `netcdf.mod` and `typesizes.mod` will be needed by the makefile template, as described in the compiling section.

6.80.2 Unpacking the distribution

The DART source code is distributed as a compressed tar file from our download site. When gunzip'ed and untarred, the source tree will begin with a directory named `DART` and will be approximately 189 Mb. Compiling the code in this tree (as is usually the case) will necessitate much more space.

gunzip `DART_post_iceland.tar.gz` tar -xvf `DART_post_iceland.tar`

The code tree is very "bushy"; there are many directories of support routines, etc. but only a few directories involved with the customization and installation of the DART software. If you can compile and run ONE of the low-order models, you should be able to compile and run ANY of the low-order models. For this reason, we can focus on the Lorenz `63 model. Subsequently, the only directories with files to be modified to check the installation are: `DART/mkmf`, `DART/models/lorenz_63/work`, and `DART/matlab` (but only for analysis).

6.80.3 Customizing the build scripts – overview

DART executable programs are constructed using two tools: `make` and `mkmf`. The `make` utility is a relatively common piece of software that requires a user-defined input file that records dependencies between different source files. `make` then performs a hierarchy of actions when one or more of the source files is modified. The `mkmf` utility is a custom preprocessor that generates a `make` input file (named `Makefile`) and an example namelist *input.nml.program_default* with the default values. The `Makefile` is designed specifically to work with object-oriented Fortran90 (and other languages) for systems like DART.

mkmf requires two separate input files. The first is a `template' file which specifies details of the commands required for a specific Fortran90 compiler and may also contain pointers to directories containing pre-compiled utilities required by the DART system. **This template file will need to be modified to reflect your system**. The second input file is a `path_names' file which includes a complete list of the locations (either relative or absolute) of all Fortran90 source files that are required to produce a particular DART program. Each 'path_names' file must contain a path for exactly one Fortran90 file containing a main program, but may contain any number of additional paths pointing to files containing Fortran90 modules. An mkmf command is executed which uses the 'path_names' file and the mkmf template file to produce a Makefile which is subsequently used by the standard make utility.

Shell scripts that execute the mkmf command for all standard DART executables are provided as part of the standard DART software. For more information on mkmf see the FMS mkmf description.

One of the benefits of using mkmf is that it also creates an example namelist file for each program. The example namelist is called *input.nml.program_default*, so as not to clash with any exising input.nml that may exist in that directory.

Building and customizing the 'mkmf.template' file

A series of templates for different compilers/architectures exists in the DART/mkmf/ directory and have names with extensions that identify either the compiler, the architecture, or both. This is how you inform the build process of the specifics of your system. Our intent is that you copy one that is similar to your system into mkmf.template and customize it. For the discussion that follows, knowledge of the contents of one of these templates (i.e. mkmf.template.pgf90.ghotiol) is needed: (note that only the LAST lines are shown here, the head of the file is just a big comment)

# Makefile template for PGI f90 FC = pgf90 LD = pgf90 CPPFLAGS = LIST = -Mlist NETCDF = /contrib/netcdf-3.5.1-cc-c++-pgif90.5.2-4 FFLAGS = -O0 -Ktrap=fp -pc 64 -I$(NETCDF)/include LIBS = -L$(NETCDF)/lib -lnetcdf LDFLAGS = $(LIBS) . . .

Essentially, each of the lines defines some part of the resulting Makefile. Since make is particularly good at sorting out dependencies, the order of these lines really doesn't make any difference. The FC = pgf90 line ultimately defines the Fortran90 compiler to use, etc. The lines which are most likely to need site-specific changes start with FFLAGS and NETCDF, which indicate where to look for the netCDF F90 modules and the location of the netCDF library and modules.

` <netCDF>`__

Netcdf

Modifying the NETCDF value should be relatively straightforward.

Change the string to reflect the location of your netCDF installation containing netcdf.mod and typesizes.mod. The value of the NETCDF variable will be used by the FFLAGS, LIBS, and LDFLAGS variables.

` <fflags>`__

FFLAGS

Each compiler has different compile flags, so there is really no way to exhaustively cover this other than to say the templates as we supply them should work – depending on the location of your netCDF. The low-order models can be compiled without a `-r8` switch, but the `bgrid_solo` model cannot.

Customizing the 'path_names_*' file

Several `path_names_*` files are provided in the `work` directory for each specific model, in this case: `DART/models/lorenz_63/work`.

1. `path_names_preprocess`
2. `path_names_create_obs_sequence`
3. `path_names_create_fixed_network_seq`
4. `path_names_perfect_model_obs`
5. `path_names_filter`
6. `path_names_obs_diag`

Since each model comes with its own set of files, no further customization is needed.

6.80.4 Building the Lorenz_63 DART project

Currently, DART executables are constructed in a `work` subdirectory under the directory containing code for the given model. In the top-level DART directory, change to the L63 work directory and list the contents:

cd DART/models/lorenz_63/work ls -1

With the result:

```
filter_ics
filter_restart
input.nml
mkmf_create_fixed_network_seq
mkmf_create_obs_sequence
mkmf_filter
mkmf_obs_diag
mkmf_perfect_model_obs
mkmf_preprocess
obs_seq.final
obs_seq.in
obs_seq.out
obs_seq.out.average
obs_seq.out.x
obs_seq.out.xy
obs_seq.out.xyz
obs_seq.out.z
path_names_create_fixed_network_seq
path_names_create_obs_sequence
path_names_filter
path_names_obs_diag
path_names_perfect_model_obs
path_names_preprocess
perfect_ics
```

(continues on next page)

(continued from previous page)

```
perfect_restart
Posterior_Diag.nc
Prior_Diag.nc
set_def.out
True_State.nc
workshop_setup.csh
```

There are six `mkmf_`*xxxxxx* files for the programs `preprocess`, `create_obs_sequence`, `create_fixed_network_seq`, `perfect_model_obs`, `filter`, and `obs_diag` along with the corresponding `path_names_`*xxxxxx* files. You can examine the contents of one of the `path_names_`*xxxxxx* files, for instance `path_names_filter`, to see a list of the relative paths of all files that contain Fortran90 modules required for the program `filter` for the L63 model. All of these paths are relative to your `DART` directory. The first path is the main program (`filter.f90`) and is followed by all the Fortran90 modules used by this program (after preprocessing).

The `mkmf_`*xxxxxx* scripts are cryptic but should not need to be modified – as long as you do not restructure the code tree (by moving directories, for example). The only function of the `mkmf_`*xxxxxx* script is to generate a `Makefile` and an *input.nml.program_default* file. It is not supposed to compile anything:

csh mkmf_preprocess make

The first command generates an appropriate `Makefile` and the `input.nml.preprocess_default` file. The second command results in the compilation of a series of Fortran90 modules which ultimately produces an executable file: `preprocess`. Should you need to make any changes to the `DART/mkmf/mkmf.template`, you will need to regenerate the `Makefile`.

The `preprocess` program actually builds source code to be used by all the remaining modules. It is **imperative** to actually **run** `preprocess` before building the remaining executables. This is how the same code can assimilate state vector 'observations' for the Lorenz_63 model and real radar reflectivities for WRF without needing to specify a set of radar operators for the Lorenz_63 model!

`preprocess` reads the `&preprocess_nml` namelist to determine what observations and operators to incorporate. For this exercise, we will use the values in `input.nml`. `preprocess` is designed to abort if the files it is supposed to build already exist. For this reason, it is necessary to remove a couple files (if they exist) before you run the preprocessor. It is just a good habit to develop.

\rm -f ../../../obs_def/obs_def_mod.f90 \rm -f ../../../obs_kind/obs_kind_mod.f90 ./preprocess ls -l ../../../obs_def/obs_def_mod.f90 ls -l ../../../obs_kind/obs_kind_mod.f90

This created `../../../obs_def/obs_def_mod.f90` from `../../../obs_kind/DEFAULT_obs_kind_mod.F90` and several other modules. `../../../obs_kind/obs_kind_mod.f90` was created similarly. Now we can build the rest of the project.

A series of object files for each module compiled will also be left in the work directory, as some of these are undoubtedly needed by the build of the other DART components. You can proceed to create the other five programs needed to work with L63 in DART as follows:

csh mkmf_create_obs_sequence make csh mkmf_create_fixed_network_seq make csh mkmf_perfect_model_obs make csh mkmf_filter make csh mkmf_obs_diag make

The result (hopefully) is that six executables now reside in your work directory. The most common problem is that the netCDF libraries and include files (particularly `typesizes.mod`) are not found. Edit the `DART/mkmf/mkmf.template`, recreate the `Makefile`, and try again.

program	purpose
`preprocess`	creates custom source code for just the observations of interest
`create_obs_sequence`	specify a (set) of observation characteristics taken by a particular (set of) instruments
`create_fixed_network_seq`	specify the temporal attributes of the observation sets
`perfect_model_obs`	spinup, generate "true state" for synthetic observation experiments, . . .
`filter`	perform experiments
`obs_diag`	creates observation-space diagnostic files to be explored by the Matlab® scripts.

6.80.5 Running Lorenz_63

This initial sequence of exercises includes detailed instructions on how to work with the DART code and allows investigation of the basic features of one of the most famous dynamical systems, the 3-variable Lorenz-63 model. The remarkable complexity of this simple model will also be used as a case study to introduce a number of features of a simple ensemble filter data assimilation system. To perform a synthetic observation assimilation experiment for the L63 model, the following steps must be performed (an overview of the process is given first, followed by detailed procedures for each step):

6.80.6 Experiment overview

1. Integrate the L63 model for a long time starting from arbitrary initial conditions to generate a model state that lies on the attractor. The ergodic nature of the L63 system means a 'lengthy' integration always converges to some point on the computer's finite precision representation of the model's attractor.
2. Generate a set of ensemble initial conditions from which to start an assimilation. Since L63 is ergodic, the ensemble members can be designed to look like random samples from the model's 'climatological distribution'. To generate an ensemble member, very small perturbations can be introduced to the state on the attractor generated by step 1. This perturbed state can then be integrated for a very long time until all memory of its initial condition can be viewed as forgotten. Any number of ensemble initial conditions can be generated by repeating this procedure.
3. Simulate a particular observing system by first creating an 'observation set definition' and then creating an 'observation sequence'. The 'observation set definition' describes the instrumental characteristics of the observations and the 'observation sequence' defines the temporal sequence of the observations.
4. Populate the 'observation sequence' with 'perfect' observations by integrating the model and using the information in the 'observation sequence' file to create simulated observations. This entails operating on the model state at the time of the observation with an appropriate forward operator (a function that operates on the model state vector to produce the expected value of the particular observation) and then adding a random sample from the observation error distribution specified in the observation set definition. At the same time, diagnostic output about the 'true' state trajectory can be created.
5. Assimilate the synthetic observations by running the filter; diagnostic output is generated.

1. Integrate the L63 model for a 'long' time

perfect_model_obs integrates the model for all the times specified in the 'observation sequence definition' file. To this end, begin by creating an 'observation sequence definition' file that spans a long time. Creating an 'observation sequence definition' file is a two-step procedure involving create_obs_sequence followed by create_fixed_network_seq. After they are both run, it is necessary to integrate the model with perfect_model_obs.

1.1 Create an observation set definition

create_obs_sequence creates an observation set definition, the time-independent part of an observation sequence. An observation set definition file only contains the location, type, and observational error characteristics (normally just the diagonal observational error variance) for a related set of observations. There are no actual observations, nor are there any times associated with the definition. For spin-up, we are only interested in integrating the L63 model, not in generating any particular synthetic observations. Begin by creating a minimal observation set definition.

In general, for the low-order models, only a single observation set need be defined. Next, the number of individual scalar observations (like a single surface pressure observation) in the set is needed. To spin-up an initial condition for the L63 model, only a single observation is needed. Next, the error variance for this observation must be entered. Since we do not need (nor want) this observation to have any impact on an assimilation (it will only be used for spinning up the model and the ensemble), enter a very large value for the error variance. An observation with a very large error variance has essentially no impact on deterministic filter assimilations like the default variety implemented in DART. Finally, the location and type of the observation need to be defined. For all types of models, the most elementary form of synthetic observations are called 'identity' observations. These observations are generated simply by adding a random sample from a specified observational error distribution directly to the value of one of the state variables. This defines the observation as being an identity observation of the first state variable in the L63 model. The program will respond by terminating after generating a file (generally named set_def.out) that defines the single identity observation of the first state variable of the L63 model. The following is a screenshot (much of the verbose logging has been left off for clarity), the user input looks *like this*.

```
[unixprompt]$ ./create_obs_sequence
 Initializing the utilities module.
 Trying to read from unit           10
 Trying to open file dart_log.out

 Registering module :
 $source: /home/dart/CVS.REPOS/DART/utilities/utilities_mod.f90,v $
 $revision: 1.18 $
 $date: 2004/06/29 15:16:40 $
 Registration complete.

 &UTILITIES_NML
 TERMLEVEL= 2,LOGFILENAME=dart_log.out

 /

 Registering module :
 $source: /home/dart/CVS.REPOS/DART/obs_sequence/create_obs_sequence.f90,v $
 $revision: 1.18 $
 $date: 2004/05/24 15:41:46 $
 Registration complete.

 { ... }
```

(continues on next page)

(continued from previous page)

```
 Input upper bound on number of observations in sequence
10

 Input number of copies of data (0 for just a definition)
0

 Input number of quality control values per field (0 or greater)
0

 input a -1 if there are no more obs
0

 Registering module :
 $Source$
 $Revision$
 $Date$
 Registration complete.


 Registering module :
 $Source$
 $Revision$
 $Date$
 Registration complete.

 initialize_module obs_kind_nml values are

 -------------- ASSIMILATE_THESE_OBS_TYPES --------------
 RAW_STATE_VARIABLE
 -------------- EVALUATE_THESE_OBS_TYPES --------------
 ------------------------------------------------------

      Input -1 * state variable index for identity observations
      OR input the name of the observation kind from table below:
      OR input the integer index, BUT see documentation...
                 1 RAW_STATE_VARIABLE

-1

 input time in days and seconds
1 0

 Input error variance for this observation definition
1000000

 input a -1 if there are no more obs
-1

 Input filename for sequence (  set_def.out   usually works well)
 set_def.out
 write_obs_seq  opening formatted file set_def.out
 write_obs_seq  closed file set_def.out
```

1.2 Create an observation sequence definition

`create_fixed_network_seq` creates an 'observation sequence definition' by extending the 'observation set definition' with the temporal attributes of the observations.

The first input is the name of the file created in the previous step, i.e. the name of the observation set definition that you've just created. It is possible to create sequences in which the observation sets are observed at regular intervals or irregularly in time. Here, all we need is a sequence that takes observations over a long period of time - indicated by entering a 1. Although the L63 system normally is defined as having a non-dimensional time step, the DART system arbitrarily defines the model timestep as being 3600 seconds. If we declare that we have one observation per day for 1000 days, we create an observation sequence definition spanning 24000 'model' timesteps; sufficient to spin-up the model onto the attractor. Finally, enter a name for the 'observation sequence definition' file. Note again: there are no observation values present in this file. Just an observation type, location, time and the error characteristics. We are going to populate the observation sequence with the `perfect_model_obs` program.

```
[unixprompt]$ ./create_fixed_network_seq

 ...

 Registering module :
 $source: /home/dart/CVS.REPOS/DART/obs_sequence/obs_sequence_mod.f90,v $
 $revision: 1.31 $
 $date: 2004/06/29 15:04:37 $
 Registration complete.

 Input filename for network definition sequence (usually  set_def.out  )
set_def.out

 ...

 To input a regularly repeating time sequence enter 1
 To enter an irregular list of times enter 2
1
 Input number of observations in sequence
1000
 Input time of initial ob in sequence in days and seconds
1, 0
 Input period of obs in days and seconds
1, 0
           1
           2
           3
...
         997
         998
         999
        1000
What is output file name for sequence (  obs_seq.in   is recommended )
obs_seq.in
 write_obs_seq  opening formatted file obs_seq.in
 write_obs_seq closed file [blah blah blah]/work/obs_seq.in
```

1.3 Initialize the model onto the attractor

`perfect_model_obs` can now advance the arbitrary initial state for 24,000 timesteps to move it onto the attractor. `perfect_model_obs` uses the Fortran90 namelist input mechanism instead of (admittedly gory, but temporary) interactive input. All of the DART software expects the namelists to found in a file called `input.nml`. When you built the executable, an example namelist was created `input.nml.perfect_model_obs_default` that contains all of the namelist input for the executable. If you followed the example, each namelist was saved to a unique name. We must now rename and edit the namelist file for `perfect_model_obs`. Copy `input.nml.perfect_model_obs_default` to `input.nml` and edit it to look like the following: (just worry about the highlighted stuff)

&perfect_model_obs_nml async = 0, adv_ens_command = "./advance_ens.csh", obs_seq_in_file_name = "obs_seq.in", obs_seq_out_file_name = "obs_seq.out", start_from_restart = .false., output_restart = .*true.*, restart_in_file_name = "perfect_ics", restart_out_file_name = "perfect_restart", init_time_days = 0, init_time_seconds = 0, output_interval = 1 / &ensemble_manager_nml in_core = .true., single_restart_file_in = .true., single_restart_file_out = .true. / &assim_tools_nml filter_kind = 1, cutoff = 0.2, sort_obs_inc = .false., cov_inflate = -1.0, cov_inflate_sd = 0.05, sd_lower_bound = 0.05, deterministic_cov_inflate = .true., start_from_assim_restart = .false., assim_restart_in_file_name = 'assim_tools_ics', assim_restart_out_file_name = 'assim_tools_restart', do_parallel = 0, num_domains = 1 parallel_command = "./assim_filter.csh", spread_restoration = .false., cov_inflate_upper_bound = 10000000.0, internal_outlier_threshold = -1.0 / &cov_cutoff_nml select_localization = 1 / &reg_factor_nml select_regression = 1, input_reg_file = "time_mean_reg" save_reg_diagnostics = .false., reg_diagnostics_file = 'reg_diagnostics' / &obs_sequence_nml write_binary_obs_sequence = .false. / &obs_kind_nml assimilate_these_obs_types = *'RAW_STATE_VARIABLE'* / &assim_model_nml write_binary_restart_files = .true. / &model_nml sigma = 10.0, r = 28.0, b = 2.6666666666667, deltat = 0.01, time_step_days = 0, time_step_seconds = 3600 / &utilities_nml TERMLEVEL = 1 logfilename = 'dart_log.out' /

For the moment, only two namelists warrant explanation. Each namelists is covered in detail in the html files accompanying the source code for the module.

perfect_model_obs_nml

namelist variable	description
`async`	For the lorenz_63, simply ignore this. Leave it set to '0'
`advance_ens_command`	specifies the shell commands or script to execute when async /= 0
`obs_seq_in_file_name`	specifies the file name that results from running `create_fixed_network_seq`, i.e. the 'observation sequence definition' file.
`obs_seq_out_file_name`	specifies the output file name containing the 'observation sequence', finally populated with (perfect?) 'observations'.
`start_from_restart`	When set to 'false', `perfect_model_obs` generates an arbitrary initial condition (which cannot be guaranteed to be on the L63 attractor).
`output_restart`	When set to 'true', `perfect_model_obs` will record the model state at the end of this integration in the file named by `restart_out_file_name`.
`restart_in_file_name`	is ignored when 'start_from_restart' is 'false'.
`restart_out_file_name`	if `output_restart` is 'true', this specifies the name of the file containing the model state at the end of the integration.
`init_time_`*xxxx*	the start time of the integration.
`output_interval`	interval at which to save the model state.

utilities_nml

namelist variable	description
`TERMLEVEL`	When set to '1' the programs terminate when a 'warning' is generated. When set to '2' the programs terminate only with 'fatal' errors.
`logfilename`	Run-time diagnostics are saved to this file. This namelist is used by all programs, so the file is opened in APPEND mode. Subsequent executions cause this file to grow.

Executing `perfect_model_obs` will integrate the model 24,000 steps and output the resulting state in the file `perfect_restart`. Interested parties can check the spinup in the `True_State.nc` file.

perfect_model_obs

2. Generate a set of ensemble initial conditions

The set of initial conditions for a 'perfect model' experiment is created in several steps. 1) Starting from the spun-up state of the model (available in `perfect_restart`), run `perfect_model_obs` to generate the 'true state' of the experiment and a corresponding set of observations. 2) Feed the same initial spun-up state and resulting observations into `filter`.

The first step is achieved by changing a perfect_model_obs namelist parameter, copying `perfect_restart` to `perfect_ics`, and rerunning `perfect_model_obs`. This execution of `perfect_model_obs` will advance the model state from the end of the first 24,000 steps to the end of an additional 24,000 steps and place the final state in `perfect_restart`. The rest of the namelists in `input.nml` should remain unchanged.

&perfect_model_obs_nml async = 0, adv_ens_command = "./advance_ens.csh", obs_seq_in_file_name = "obs_seq.in", obs_seq_out_file_name = "obs_seq.out", start_from_restart = *.true.*, output_restart = .true., restart_in_file_name = "perfect_ics", restart_out_file_name = "perfect_restart", init_time_days = 0, init_time_seconds = 0, output_interval = 1 /

cp perfect_restart perfect_ics perfect_model_obs

A `True_State.nc` file is also created. It contains the 'true' state of the integration.

Generating the ensemble

This step (#2 from above) is done with the program `filter`, which also uses the Fortran90 namelist mechanism for input. It is now necessary to copy the `input.nml.filter_default` namelist to `input.nml` or you may simply insert the `filter_nml` namelist block into the existing `input.nml`. Having the `perfect_model_obs` namelist in the input.nml does not hurt anything. In fact, I generally create a single `input.nml` that has all the namelist blocks in it. I simply copied the filter namelist block from `input.nml.filter_default` and inserted it into our `input.nml` for the following example.

&perfect_model_obs_nml async = 0, adv_ens_command = "./advance_ens.csh", obs_seq_in_file_name = "obs_seq.in", obs_seq_out_file_name = "obs_seq.out", start_from_restart = .true., output_restart = .true., restart_in_file_name = "perfect_ics", restart_out_file_name = "perfect_restart", init_time_days = 0, init_time_seconds = 0, output_interval = 1 / &filter_nml async = 0, adv_ens_command = "./advance_ens.csh", ens_size = *100*, cov_inflate = 1.0, start_from_restart = .false., output_restart = *.true.*, obs_sequence_in_name = "obs_seq.out", obs_sequence_out_name = "obs_seq.final", restart_in_file_name = *"perfect_ics"*, restart_out_file_name = "filter_restart", init_time_days = 0, init_time_seconds = 0, output_state_ens_mean = .true., output_state_ens_spread = .true., output_obs_ens_mean = .true., output_obs_ens_spread = .true., num_output_state_members = *20*, num_output_obs_members = *20*, output_interval = 1, num_groups = 1, outlier_threshold = -1.0 / &ensemble_manager_nml in_core = .true., single_restart_file_in = .true., single_restart_file_out = .true. / &assim_tools_nml filter_kind = 1, cutoff = 0.2, sort_obs_inc = .false., cov_inflate = -1.0, cov_inflate_sd = 0.05, sd_lower_bound = 0.05, deterministic_cov_inflate = .true., start_from_assim_restart = .false., assim_restart_in_file_name = 'assim_tools_ics', assim_restart_out_file_name = 'assim_tools_restart', do_parallel = 0, num_domains = 1 parallel_command = "./assim_filter.csh", spread_restoration = .false., cov_inflate_upper_bound = 10000000.0, internal_outlier_threshold = -1.0 / &cov_cutoff_nml select_localization = 1 / &reg_factor_nml select_regression = 1, input_reg_file = "time_mean_reg" save_reg_diagnostics = .false., reg_diagnostics_file = 'reg_diagnostics' / &obs_sequence_nml write_binary_obs_sequence = .false. / &obs_kind_nml assimilate_these_obs_types = 'RAW_STATE_VARIABLE' / &assim_model_nml write_binary_restart_files = .true. / &model_nml sigma = 10.0, r = 28.0, b = 2.6666666666667, deltat = 0.01, time_step_days = 0, time_step_seconds = 3600 / &utilities_nml TERMLEVEL = 1 logfilename = 'dart_log.out' /

Only the non-obvious(?) entries for `filter_nml` will be discussed.

namelist variable	description
`ens_size`	Number of ensemble members. 100 is sufficient for most of the L63 exercises.
`cov_inflate`	A value of 1.0 results in no inflation.(spin-up)
`start_from_restart`	when '.false.', `filter` will generate its own ensemble of initial conditions. It is important to note that the filter still makes use of `perfect_ics` by randomly perturbing these state variables.
`output_state_ens_mean`	when '.true.' the mean of all ensemble members is output.
`output_state_ens_spread`	when '.true.' the spread of all ensemble members is output.
`num_output_state_members`	may be a value from 0 to `ens_size`
`output_obs_ens_mean`	when '.true.' Output ensemble mean in observation output file.
`output_obs_ens_spread`	when '.true.' Output ensemble spread in observation output file.
`num_output_obs_members`	may be a value from 0 to `ens_size`
`output_interval`	The frequency with which output state diagnostics are written. Units are in assimilation times. Default value is 1 meaning output is written at every observation time

The filter is told to generate its own ensemble initial conditions since `start_from_restart` is '.false.'. However, it is important to note that the filter still makes use of `perfect_ics` which is set to be the `restart_in_file_name`. This is the model state generated from the first 24,000 step model integration by `perfect_model_obs`. `Filter` generates its ensemble initial conditions by randomly perturbing the state variables of this state.

The arguments `output_state_ens_mean` and `output_state_ens_spread` are '.true.' so that these quantities are output at every time for which there are observations (once a day here) and `num_output_ens_members` means that the same diagnostic files, `Posterior_Diag.nc` and `Prior_Diag.nc` also contain values for 20 ensemble members once a day. Once the namelist is set, execute `filter` to integrate the ensemble forward for 24,000 steps with the final ensemble state written to the `filter_restart`. Copy the `perfect_model_obs` restart file `perfect_restart` (the `true state') to `perfect_ics`, and the `filter` restart file `filter_restart` to `filter_ics` so that future assimilation experiments can be initialized from these spun-up states.

filter cp perfect_restart perfect_ics cp filter_restart filter_ics

The spin-up of the ensemble can be viewed by examining the output in the netCDF files `True_State.nc` generated by `perfect_model_obs` and `Posterior_Diag.nc` and `Prior_Diag.nc` generated by `filter`. To do this, see the detailed discussion of matlab diagnostics in Appendix I.

3. Simulate a particular observing system

Begin by using `create_obs_sequence` to generate an observation set in which each of the 3 state variables of L63 is observed with an observational error variance of 1.0 for each observation. To do this, use the following input sequence (the text including and after # is a comment and does not need to be entered):

4	# upper bound on num of observations in sequence
0	# number of copies of data (0 for just a definition)
0	# number of quality control values per field (0 or greater)
0	# -1 to exit/end observation definitions
-1	# observe state variable 1
0 0	# time – days, seconds
1.0	# observational variance
0	# -1 to exit/end observation definitions
-2	# observe state variable 2
0 0	# time – days, seconds
1.0	# observational variance
0	# -1 to exit/end observation definitions
-3	# observe state variable 3
0 0	# time – days, seconds
1.0	# observational variance
-1	# -1 to exit/end observation definitions
set_def.out	# Output file name

Now, generate an observation sequence definition by running `create_fixed_network_seq` with the following input sequence:

set_def.out	# Input observation set definition file
1	# Regular spaced observation interval in time
1000	# 1000 observation times
0, 43200	# First observation after 12 hours (0 days, 12 * 3600 seconds)
0, 43200	# Observations every 12 hours
obs_seq.in	# Output file for observation sequence definition

4. Generate a particular observing system and true state

An observation sequence file is now generated by running `perfect_model_obs` with the namelist values (unchanged from step 2):

&perfect_model_obs_nml async = 0, adv_ens_command = "./advance_ens.csh", obs_seq_in_file_name = "obs_seq.in", obs_seq_out_file_name = "obs_seq.out", start_from_restart = .true., output_restart = .true., restart_in_file_name = "perfect_ics", restart_out_file_name = "perfect_restart", init_time_days = 0, init_time_seconds = 0, output_interval = 1 /

This integrates the model starting from the state in `perfect_ics` for 1000 12-hour intervals outputting synthetic observations of the three state variables every 12 hours and producing a netCDF diagnostic file, `True_State.nc`.

5. Filtering

Finally, `filter` can be run with its namelist set to:

&filter_nml async = 0, adv_ens_command = “./advance_ens.csh”, ens_size = 100, cov_inflate = 1.0, start_from_restart = *.true.*, output_restart = .true., obs_sequence_in_name = “obs_seq.out”, obs_sequence_out_name = “obs_seq.final”, restart_in_file_name = *“filter_ics”*, restart_out_file_name = “filter_restart”, init_time_days = 0, init_time_seconds = 0, output_state_ens_mean = .true., output_state_ens_spread = .true., output_obs_ens_mean = .true., output_obs_ens_spread = .true., num_output_state_members = 20, num_output_obs_members = 20, output_interval = 1, num_groups = 1, outlier_threshold = -1.0 /

`filter` produces two output diagnostic files, `Prior_Diag.nc` which contains values of the ensemble mean, ensemble spread, and ensemble members for 12- hour lead forecasts before assimilation is applied and `Posterior_Diag.nc` which contains similar data for after the assimilation is applied (sometimes referred to as analysis values).

Now try applying all of the matlab diagnostic functions described in the Matlab Diagnostics section.

6.80.7 Matlab® diagnostics

The output files are netCDF files, and may be examined with many different software packages. We happen to use Matlab®, and provide our diagnostic scripts in the hopes that they are useful.

The diagnostic scripts and underlying functions reside in two places: `DART/diagnostics/matlab` and `DART/matlab`. They are reliant on the public-domain netcdf toolbox from `http://woodshole.er.usgs.gov/staffpages/cdenham/public_html/MexCDF/nc4ml5.html` as well as the public-domain CSIRO matlab/netCDF interface from `http://www.marine.csiro.au/sw/matlab-netcdf.html`. If you do not have them installed on your system and want to use Matlab to peruse netCDF, you must follow their installation instructions. The ‘interested reader’ may want to look at the `DART/matlab/startup.m` file I use on my system. If you put it in your `$HOME/matlab` directory, it is invoked every time you start up Matlab.

Once you can access the `getnc` function from within Matlab, you can use our diagnostic scripts. It is necessary to prepend the location of the `DART/matlab` scripts to the `matlabpath`. Keep in mind the location of the netcdf operators on your system WILL be different from ours … and that’s OK.

```
0[269]0 ghotiol:/<5>models/lorenz_63/work]$ matlab -nojvm

                                               < M A T L A B >
                                   Copyright 1984-2002 The MathWorks, Inc.
                                       Version 6.5.0.180913a Release 13
                                                 Jun 18 2002

  Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.

  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

>> which getnc
/contrib/matlab/matlab_netcdf_5_0/getnc.m
>>ls *.nc

ans =
```

(continues on next page)

(continued from previous page)

```
Posterior_Diag.nc  Prior_Diag.nc  True_State.nc


>>path('../../../matlab',path)
>>path('../../../diagnostics/matlab',path)
>>which plot_ens_err_spread
../../../matlab/plot_ens_err_spread.m
>>help plot_ens_err_spread

  DART : Plots summary plots of the ensemble error and ensemble spread.
                       Interactively queries for the needed information.
                       Since different models potentially need different
                       pieces of information ... the model types are
                       determined and additional user input may be queried.

  Ultimately, plot_ens_err_spread will be replaced by a GUI.
  All the heavy lifting is done by PlotEnsErrSpread.

  Example 1 (for low-order models)

  truth_file = 'True_State.nc';
  diagn_file = 'Prior_Diag.nc';
  plot_ens_err_spread

>>plot_ens_err_spread
```

And the matlab graphics window will display the spread of the ensemble error for each state variable. The scripts are designed to do the “obvious” thing for the low-order models and will prompt for additional information if needed. The philosophy of these is that anything that starts with a lower-case *plot_some_specific_task* is intended to be user-callable and should handle any of the models. All the other routines in `DART/matlab` are called BY the high-level routines.

Matlab script	description
`plot_bins`	plots ensemble rank histograms
`plot_correl`	Plots space-time series of correlation between a given variable at a given time and other variables at all times in a n ensemble time sequence.
`plot_ens_err_spread`	Plots summary plots of the ensemble error and ensemble spread. Interactively queries for the needed information. Since different models potentially need different pieces of information . . . the model types are determined and additional user input may be queried.
`plot_ens_mean_time_series`	Queries for the state variables to plot.
`plot_ens_time_series`	Queries for the state variables to plot.
`plot_phase_space`	Plots a 3D trajectory of (3 state variables of) a single ensemble member. Additional trajectories may be superimposed.
`plot_total_err`	Summary plots of global error and spread.
`plot_var_var_correl`	Plots time series of correlation between a given variable at a given time and another variable at all times in an ensemble time sequence.

6.80.8 Bias, filter divergence and covariance inflation (with the l63 model)

One of the common problems with ensemble filters is filter divergence, which can also be an issue with a variety of other flavors of filters including the classical Kalman filter. In filter divergence, the prior estimate of the model state becomes too confident, either by chance or because of errors in the forecast model, the observational error characteristics, or approximations in the filter itself. If the filter is inappropriately confident that its prior estimate is correct, it will then tend to give less weight to observations than they should be given. The result can be enhanced overconfidence in the model's state estimate. In severe cases, this can spiral out of control and the ensemble can wander entirely away from the truth, confident that it is correct in its estimate. In less severe cases, the ensemble estimates may not diverge entirely from the truth but may still be too confident in their estimate. The result is that the truth ends up being farther away from the filter estimates than the spread of the filter ensemble would estimate. This type of behavior is commonly detected using rank histograms (also known as Talagrand diagrams). You can see the rank histograms for the L63 initial assimilation by using the matlab script `plot_bins`.

A simple, but surprisingly effective way of dealing with filter divergence is known as covariance inflation. In this method, the prior ensemble estimate of the state is expanded around its mean by a constant factor, effectively increasing the prior estimate of uncertainty while leaving the prior mean estimate unchanged. The program `filter` has a namelist parameter that controls the application of covariance inflation, `cov_inflate`. Up to this point, `cov_inflate` has been set to 1.0 indicating that the prior ensemble is left unchanged. Increasing `cov_inflate` to values greater than 1.0 inflates the ensemble before assimilating observations at each time they are available. Values smaller than 1.0 contract (reduce the spread) of prior ensembles before assimilating.

You can do this by modifying the value of `cov_inflate` in the namelist, (try 1.05 and 1.10 and other values at your discretion) and run the filter as above. In each case, use the diagnostic matlab tools to examine the resulting changes to the error, the ensemble spread (via rank histogram bins, too), etc. What kind of relation between spread and error is seen in this model?

6.80.9 Synthetic observations

Synthetic observations are generated from a \`perfect' model integration, which is often referred to as the \`truth' or a \`nature run'. A model is integrated forward from some set of initial conditions and observations are generated as $y = H(x) + e$ where H is an operator on the model state vector, x, that gives the expected value of a set of observations, y, and e is a random variable with a distribution describing the error characteristics of the observing instrument(s) being simulated. Using synthetic observations in this way allows students to learn about assimilation algorithms while being isolated from the additional (extreme) complexity associated with model error and unknown observational error characteristics. In other words, for the real-world assimilation problem, the model has (often substantial) differences from what happens in the real system and the observational error distribution may be very complicated and is certainly not well known. Be careful to keep these issues in mind while exploring the capabilities of the ensemble filters with synthetic observations.

6.81 DART Iceland revisions

The DART Iceland release (23 Nov 2005) is expected to be the last update that makes major modifications to the user interfaces to DART and the DART interfaces to models or observations. The primary purpose of this release is to improve the way that DART handles observation sequences. The goal is to provide a more modular fashion for adding new types of observations with potentially complex forward operators and arbitrary amounts and types of metadata that are needed to define a particular observation. Limited backward compatibility has been implemented for existing observation sequence files, however, it is strongly recommended that users switch to the new format as quickly as possible.

6.81.1 Improvements

The changes to the observation sequences impact large portions of the DART code. In addition, Iceland also implements a number of other improvements and additions to DART and fixes several known bugs from earlier releases. Highlights of the changes from the workshop and Hawaii releases are:

1. Namelist error detection. Enhanced error checking capabilities for DART namelist reads are included. The input.nml must now contain an entry for each namelist that is read by a program, even if none of the values in that namelist entry are set. For instance, if a program that uses the assim_tools_mod is run, input.nml MUST contain an entry &assim_tools_nml. Any namelist values to be set must follow this and the list is terminated by a /. If no entries are to be set in a given namelist, it must still be terminated by a line that contains only a slash (/). If a variable that is NOT in a given namelist is found in input.nml, a fatal error occurs. Failing to terminate a namelist with slash is also fatal. Tools to support this improved namelist error detection are implemented in utilities_mod and can be used to check for errors in other applications.

2. Automatic detection of input file format. Restart and observation sequence files can be written in binary or ascii as in previous releases. However, file formats for reads are now detected automatically. The namelist entries 'read_binary_restart_files' in assim_model_nml and 'read_binary_obs_sequence' in obs_sequence_nml have been removed.

3. Error corrected in threed_sphere/location_mod. An error in the Hawaii and Workshop releases led to the possibility of erroneous computations of distance in the threed_sphere/location_mod. The error only occurred if the namelist entry 'approximate_distance' was set to .true. (the default was .false.) and errors were generally small and confined to observations located in polar regions. The result could be that localization was not applied properly to polar observations.

4. Support for reduced precision real computation. For some large model applications, using reduced precision reals can enhance performance. The definition of 'real(r8)' in types_mod can now be modified to support different real precision. Some computations in the time_manager can fail with reduced precision, so all time computations now use an independent fixed precision.

5. Quality control definition change. values written to quality control (qc) fields by perfect_model_obs and filter have been modified. If a prior (posterior) forward operator computation fails, qc for that observation is incremented by 1000 (1000000). If the prior (posterior) observation lies outside the outlier_threshold, the qc for that observation is incremented by 100 (400).

6. Added obs_sequence file header. Observation sequence files (obs_sequence files) now include a metadata header that includes a list of the names of each observation type that might be used in the file along with a mapping to an integer for each name in the file. Old format files without a header are automatically detected and a default mapping of integer indices for observation types to observation type names is used. This default mapping works for most low-order model, bgrid and CAM obs_sequence files. It is strongly recommended that legacy obs_sequence files be converted to the new format by inserting a header block with the appropriate names and integer mappings.

7. Interactive creation input for obs_sequence files. A standard method for creating obs_sequence files is to redirect a text file to standard input and use the interactive creation facility in create_obs_sequence. Old input files for this procedure may no longer work correctly and may need to be updated. The new interactive creation interface allows the name of observations to be used, or an integer index. However, the integer index is now defined by a table in the obs_kind_mod and may change dynamically. Users should understand this procedure.

8. obs_def_nml moved to obs_kind_nml. The obs_def_nml in previous releases had entries to choose what observation types to assimilate or evaluate. These entries have been moved to obs_kind_nml and obs_def_nml no longer exists.

9. Preprocessor functionality has changed. While the preprocess program still needs to be run when setting up DART experiments, it now works differently. Previously, the preprocessor read obs_def_mod.F90 and created obs_def_mod.f90 in a fashion very similar to the standard CPP preprocessor. The new preprocessor does not look like CPP. It reads input from DEFAULT_obs_kind_mod.F90, DEFAULT_obs_def_mod.F90, and from any

requested special obs_def modules and creates an obs_kind_mod.f90 and an obs_def_mod.f90. Documentation of this new functionality is available in tutorial section 21 and in the html files for preprocess, obs_kind_mod, and obs_def_mod.

10. Improved observation space diagnostics interfaces. The observation space diagnostics program and associated diagnostic tools are now all located in the diagnostics directory. The interfaces and controls have been modified and are described in html documentation and in the tutorial section 18. Consistent interfaces for low-order models and large three-dimensional atmospheric models have been provided.

6.81.2 Changes

A summary list of changes occurring in each DART directory follows:

assim_model	Uses digits12 for netcdf time computations allowing reduced precision models; automatic detect
assim_tools	Namelist error checking added. Order in which computation of ensemble prior and observation t
common: types_mod.f90	Added digits12 real precision kind for doing times with required precision when rest of assimilat
converters	These WRF specific routines have been significantly modified to use the updated release of WRF
cov_cutoff	added optional argument localization_override to comp_cov_factor to override the namelist selec
diagnostics	Observation space diagnostics are now included in this directory. There are directories for oned r
ensemble_manager	Includes commented block needed to write out ensemble mean for WRF boundary forcing comp
filter	Incorporated new namelist error checking, modified calls to read_obs_seq_header to support auto
integrate_model	Namelist error checking.
location/threed_sphere	Added 5 VERTIS**** variables for describing vertical location kinds. Corrected error in table lo
matlab	Minor modifications to several scripts.
mkmf	Templates cleaned up and templates for additional platforms added.
models	All with namelists have namelist error detection.
models/bgrid_solo	Use new generic kind definitions to decide how to interpolate observations.
models/lorenz_04	Added nc_read_model_vars to read in netcdf file format.
ncep_obs	The code from NCEP to read bufr files has been added to the directory. This is not technically pa
DEFAULT_obs_def_mod.F90	Replaces obs_def_mod.f90, preprocessed to create obs_def_mod.f90. No longer has a namelist (
obs_def_dew_point_mod.f90	New module for doing dew point forward operators.
obs_def_metar_mod.f90	New module for doing surface observation forward operators.
obs_def_radar_mod.f90	Revised version of radar forward operator module that works with DEFAULT_obs_def_mod.F90
obs_def_1d_state_mod.f90	Computes forward operators for interpolation and integrals of low-order models with a single sta
obs_def_reanalysis_bufr_mod.f90	Computes forward operators for all types of observations available in the reanalysis BUFR files.
DEFAULT_obs_kind_mod.F90	Replaces obs_kind_mod.f90, preprocessed to create obs_kind_mod.f90. Includes new 'generic' k
obs_sequence_mod.f90	obs_sequence files now have a header that maps from obs_kind indices to a string that uniquely i
perfect_model_obs.f90	Uses revised calls to read_obs_seq_header and read_obs_seq to use automatic file format detecti
preprocess.f90	Now preprocesses the DEFAULT_obs_kind_mod.F90 and DEFAULT_obs_def_mod.F90 and inp
reg_factor_mod.f90	Automatic namelist error detection.
shell_scripts	Several new scripts for managing files and cleaning up DART directories have been added. Signi
time_manager_mod.f90	Use of digits12 precision for real computations allows reduced precision to be used for rest of da
tutorial	The workshop tutorial scripts have been updated to correct several errors and to be consistent wit
utilities_mod.f90	Namelist error detection added.

6.81.3 Future enhancements / work

- Extend PBL_1d support for all matlab scripts. currently only supported by the observation-space diagnostics and a crude implementation for 'plot_total_err'.
- Unify the machine-specific scripts to handle PBS, LSF and interactive submission in one script.
- Incorporate support for 'null_model'. A useful exercise to test many facets of the DART code without a chaotic model. Should provide capability to perform regression testing of DART infrasturcture.
- Improve netcdf error messages. Will incorporate an additional argument to the 'check' routine to append a string to the netCDF error library string.

6.82 DART "pre_hawaii release" Documentation

6.83 Overview of DART

The Data Assimilation Research Testbed (DART) is designed to facilitate the combination of assimilation algorithms, models, and observation sets to allow increased understanding of all three. The DART programs have been compiled with the Intel 7.1 Fortran compiler and run on a linux compute-server. If your system is different, you will definitely need to read the Customizations section.

The latest software release – "pre_hawaii"[43 Mb – .tar.gz]

DART programs can require three different types of input. First, some of the DART programs, those for creating synthetic observational datasets, require interactive input from the keyboard. For simple cases, this interactive input can be made directly from the keyboard. In more complicated cases, a file containing the appropriate keyboard input can be created and this file can be directed to the standard input of the DART program. Second, many DART programs expect one or more input files in DART specific formats to be available. For instance, `perfect_model_obs` creates a synthetic observation set given a particular model and a description of a sequence of observations requires an input file that describes this observation sequence. At present, the observation files for DART are inefficient but human-readable ascii files in a custom format. Third, many DART modules (including main programs) make use of the Fortan90 namelist facility to obtain values of certain parameters at run-time. All programs look for a namelist input file called `input.nml` in the directory in which the program is executed. The `input.nml` file can contain a sequence of individual Fortran90 namelists which specify values of particular parameters for modules that compose the executable program. Unfortunately, the Fortran90 namelist interface is poorly defined in the language standard, leaving considerable leeway to compiler developers in implementing the facility. The Intel 7.1 compiler has some particularly unpleasant behavior when a namelist file contains an entry that is NOT defined in the program reading the namelist. Error behavior is unpredictable, but often results in read errors for other input files opened by DART programs. If you encounter run-time read errors, the first course of action should be to ensure the components of the namelist are actual components. Changing the names of the namelist components **will** create unpleasantries. DART provides a mechanism that automatically generates namelists with the default values for each program to be run.

DART uses the netCDF self-describing data format with a particular metadata convention to describe output that is used to analyze the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. A set of Matlab scripts, designed to produce graphical diagnostics from DART netCDF output files are available. DART users have also used ncview to create rudimentary graphical displays of output data fields. The NCO tools, produced by UCAR's Unidata group, are available to do operations like concatenating, slicing, and dicing of netCDF files.

6.83.1 Requirements: an F90 compiler

The DART software has been successfully built on several Linux/x86 platforms with the Intel Fortran Compiler 7.1 for Linux, which is free for individual scientific use. It has also been built and successfully run with the Portland Group Fortran Compiler (5.02), and again with the Intel 8.0.034 compiler. Since recompiling the code is a necessity to experiment with different models, there are no binaries to distribute.

DART uses the netCDF self-describing data format for the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. In particular, DART also makes use of the F90 interface to the library which is available through the `netcdf.mod` and `typesizes.mod` modules. *IMPORTANT*: different compilers create these modules with different "case" filenames, and sometimes they are not **both** installed into the expected directory. It is required that both modules be present. The normal place would be in the `netcdf/include` directory, as opposed to the `netcdf/lib` directory.

If the netCDF library does not exist on your system, you must build it (as well as the F90 interface modules). The library and instructions for building the library or installing from an RPM may be found at the netCDF home page: http://www.unidata.ucar.edu/packages/netcdf/ Pay particular attention to the compiler-specific patches that must be applied for the Intel Fortran Compiler. (Or the PG compiler, for that matter.)

The location of the netCDF library, `libnetcdf.a`, and the locations of both `netcdf.mod` and `typesizes.mod` will be needed by the makefile template, as described in the compiling section.

DART also uses the **very** common udunits library for manipulating units of physical quantities. If, somehow, it is not installed on your system, you will need to install it (instructions are available from Unidata's Downloads page).

The location of the udunits library, `libudunits.a`, will be needed by the makefile template, as described in the compiling section.

6.83.2 Unpacking the distribution

The DART source code is distributed as a compressed tar file DART_hawaii.tar.gz [22347692 bytes]. When untarred, the source tree will begin with a directory named `DART` and will be approximately 105 Mb. Compiling the code in this tree (as is usually the case) will necessitate much more space.

gunzip `DART_pre_hawaii.tar.gz` tar -xvf `DART_pre_hawaii.tar`

The code tree is very "bushy"; there are many directories of support routines, etc. but only a few directories involved with the customization and installation of the DART software. If you can compile and run ONE of the low-order models, you should be able to compile and run ANY of the low-order models. For this reason, we can focus on the Lorenz `63 model. Subsequently, the only directories with files to be modified to check the installation are: `DART/mkmf`, `DART/models/lorenz_63/work`, and `DART/matlab` (but only for analysis).

6.83.3 Customizing the build scripts – overview

DART executable programs are constructed using two tools: `make` and `mkmf`. The `make` utility is a relatively common piece of software that requires a user-defined input file that records dependencies between different source files. `make` then performs a hierarchy of actions when one or more of the source files is modified. The `mkmf` utility is a custom preprocessor that generates a `make` input file (named `Makefile`) and an example namelist `input.nml.mkmf` with the default values. The `Makefile` is designed specifically to work with object-oriented Fortran90 (and other languages) for systems like DART.

`mkmf` requires two separate input files. The first is a `template' file which specifies details of the commands required for a specific Fortran90 compiler and may also contain pointers to directories containing pre-compiled utilities required by the DART system. **This template file will need to be modified to reflect your system**. The second input file is a `path_names' file which includes a complete list of the locations (either relative or absolute) of all Fortran90 source files that are required to produce a particular DART program. Each 'path_names' file must contain a path for exactly one Fortran90 file containing a main program, but may contain any number of additional paths pointing to files containing Fortran90 modules. An mkmf command is executed which uses the 'path_names' file and the mkmf template file to produce a Makefile which is subsequently used by the standard make utility.

Shell scripts that execute the mkmf command for all standard DART executables are provided as part of the standard DART software. For more information on mkmf see the FMS mkmf description.
One of the benefits of using mkmf is that it also creates an example namelist file for each program. The example namelist is called input.nml.mkmf, so as not to clash with any exising input.nml that may exist in that directory.

Building and customizing the 'mkmf.template' file

A series of templates for different compilers/architectures exists in the DART/mkmf/ directory and have names with extensions that identify either the compiler, the architecture, or both. This is how you inform the build process of the specifics of your system. Our intent is that you copy one that is similar to your system into mkmf.template and customize it. For the discussion that follows, knowledge of the contents of one of these templates (i.e. mkmf.template.pgi) is needed: (note that only the first few lines are shown here)

# Makefile template for PGI f90 FC = pgf90 CPPFLAGS = FFLAGS = -r8 -Ktrap=fp -pc 64 -I/usr/local/netcdf/include LD = pgf90 LDFLAGS = $(LIBS) LIBS = -L/usr/local/netcdf/lib -lnetcdf -L/usr/local/udunits-1.11.7/lib -ludunits LIST = -Mlist # you should never need to change any lines below. . . .

Essentially, each of the lines defines some part of the resulting Makefile. Since make is particularly good at sorting out dependencies, the order of these lines really doesn't make any difference. The FC = pgf90 line ultimately defines the Fortran90 compiler to use, etc. The lines which are most likely to need site-specific changes start with FFLAGS and LIBS, which indicate where to look for the netCDF F90 modules and the location of the netCDF and udunits libraries.

` <fflags>`__

FFLAGS

Each compiler has different compile flags, so there is really no way to exhaustively cover this other than to say the templates as we supply them should work – depending on the location of the netCDF modules netcdf.mod and typesizes.mod. Change the /usr/local/netcdf/include string to reflect the location of your modules. The low-order models can be compiled without the -r8 switch, but the bgrid_solo model cannot.

` <libs>`__

Libs

Modifying the LIBS value should be relatively straightforward.
Change the /usr/local/netcdf/lib string to reflect the location of your libnetcdf.a.
Change the /usr/local/udunits-1.11.7/lib string to reflect the location of your libudunits.a.

Customizing the 'path_names_*' file

Several `path_names_*` files are provided in the `work` directory for each specific model, in this case: `DART/models/lorenz_63/work`.

1. `path_names_create_obs_sequence`
2. `path_names_create_fixed_network_seq`
3. `path_names_perfect_model_obs`
4. `path_names_filter`

Since each model comes with its own set of files, no further customization is needed.

6.83.4 Building the Lorenz_63 DART project

Currently, DART executables are constructed in a `work` subdirectory under the directory containing code for the given model. In the top-level DART directory, change to the L63 work directory and list the contents:

cd DART/models/lorenz_63/work ls -1

With the result:

```
filter_ics
mkmf_create_fixed_network_seq
mkmf_create_obs_sequence
mkmf_filter
mkmf_perfect_model_obs
path_names_create_fixed_network_seq
path_names_create_obs_sequence
path_names_filter
path_names_perfect_model_obs
perfect_ics
```

There are four `mkmf_`*xxxxxx* files for the programs `create_obs_sequence`, `create_fixed_network_seq`, `perfect_model_obs`, and `filter` along with the corresponding `path_names_`*xxxxxx* files. You can examine the contents of one of the `path_names_`*xxxxxx* files, for instance `path_names_filter`, to see a list of the relative paths of all files that contain Fortran90 modules required for the program `filter` for the L63 model. All of these paths are relative to your `DART` directory. The first path is the main program (`filter.f90`) and is followed by all the Fortran90 modules used by this program.

The `mkmf_`*xxxxxx* scripts are cryptic but should not need to be modified – as long as you do not restructure the code tree (by moving directories, for example). The only function of the `mkmf_`*xxxxxx* script is to generate a `Makefile` and an instance of the default namelist file: `input.nml.`*xxxxxx*`_default`. It is not supposed to compile anything.

csh mkmf_create_obs_sequence make

The first command generates an appropriate `Makefile` and the `input.nml.create_obs_sequence_default` file. The `make` command results in the compilation of a series of Fortran90 modules which ultimately produces an executable file: `create_obs_sequence`. Should you need to make any changes to the `DART/mkmf/mkmf.template`, (*i.e.* change compile options) you will need to regenerate the `Makefile`. A series of object files for each module compiled will also be left in the work directory, as some of these are undoubtedly needed by the build of the other DART components. You can proceed to create the other three programs needed to work with L63 in DART as follows:

csh mkmf_create_fixed_network_seq make csh mkmf_perfect_model_obs make csh mkmf_filter make

The result (hopefully) is that four executables now reside in your work directory. The most common problem is that the netCDF libraries and include files (particularly `typesizes.mod`) are not found. If this is the case; edit the `DART/mkmf/mkmf.template`, recreate the `Makefile`, and try again.

program	purpose
`create_obs_sequence`	specify a (set) of observation characteristics taken by a particular (set of) instruments
`create_fixed_network_seq`	specify the temporal attributes of the observation sets
`perfect_model_obs`	spinup, generate "true state" for synthetic observation experiments, …
`filter`	perform experiments

6.83.5 Running Lorenz_63

This initial sequence of exercises includes detailed instructions on how to work with the DART code and allows investigation of the basic features of one of the most famous dynamical systems, the 3-variable Lorenz-63 model. The remarkable complexity of this simple model will also be used as a case study to introduce a number of features of a simple ensemble filter data assimilation system. To perform a synthetic observation assimilation experiment for the L63 model, the following steps must be performed (an overview of the process is given first, followed by detailed procedures for each step):

6.83.6 Experiment overview

1. Integrate the L63 model for a long time starting from arbitrary initial conditions to generate a model state that lies on the attractor. The ergodic nature of the L63 system means a 'lengthy' integration always converges to some point on the computer's finite precision representation of the model's attractor.
2. Generate a set of ensemble initial conditions from which to start an assimilation. Since L63 is ergodic, the ensemble members can be designed to look like random samples from the model's 'climatological distribution'. To generate an ensemble member, very small perturbations can be introduced to the state on the attractor generated by step 1. This perturbed state can then be integrated for a very long time until all memory of its initial condition can be viewed as forgotten. Any number of ensemble initial conditions can be generated by repeating this procedure.
3. Simulate a particular observing system by first creating an 'observation set definition' and then creating an 'observation sequence'. The 'observation set definition' describes the instrumental characteristics of the observations and the 'observation sequence' defines the temporal sequence of the observations.
4. Populate the 'observation sequence' with 'perfect' observations by integrating the model and using the information in the 'observation sequence' file to create simulated observations. This entails operating on the model state at the time of the observation with an appropriate forward operator (a function that operates on the model state vector to produce the expected value of the particular observation) and then adding a random sample from the observation error distribution specified in the observation set definition. At the same time, diagnostic output about the 'true' state trajectory can be created.
5. Assimilate the synthetic observations by running the filter; diagnostic output is generated.

1. Integrate the L63 model for a ‘long’ time

`perfect_model_obs` integrates the model for all the times specified in the ‘observation sequence definition’ file. To this end, begin by creating an ‘observation sequence definition’ file that spans a long time. Creating an ‘observation sequence definition’ file is a two-step procedure involving `create_obs_sequence` followed by `create_fixed_network_seq`. After they are both run, it is necessary to integrate the model with `perfect_model_obs`.

1.1 Create an observation set definition

`create_obs_sequence` creates an observation set definition, the time-independent part of an observation sequence. An observation set definition file only contains the `location, type,` and `observational error characteristics` (normally just the diagonal observational error variance) for a related set of observations. There are no actual observations. For spin-up, we are only interested in integrating the L63 model, not in generating any particular synthetic observations. Begin by creating a minimal observation set definition.

More information can be found in DART/assimilation_code/programs/create_obs_sequence/create_obs_sequence.html and DART/assimilation_code/modules/observations/obs_sequence_mod.html

In general, for the low-order models, only a single observation set need be defined. Next, the number of individual scalar observations (like a single surface pressure observation) in the set is needed. To spin-up an initial condition for the L63 model, only a single observation is needed. Next, the error variance for this observation must be entered. Since we do not need (nor want) this observation to have any impact on an assimilation (it will only be used for spinning up the model and the ensemble), enter a very large value for the error variance. An observation with a very large error variance has essentially no impact on deterministic filter assimilations like the default variety implemented in DART. Finally, the location and type of the observation need to be defined. For all types of models, the most elementary form of synthetic observations are called ‘identity’ observations. These observations are generated simply by adding a random sample from a specified observational error distribution directly to the value of one of the state variables. This defines the observation as being an identity observation of the first state variable in the L63 model. The program will respond by terminating after generating a file (generally named `set_def.out`) that defines the single identity observation of the first state variable of the L63 model. The following is a screenshot (much of the verbose logging has been left off for clarity), the user input looks *like this*.

```
[unixprompt]$ ./create_obs_sequence
 Initializing the utilities module.
 Trying to log to unit           10
 Trying to open file dart_log.out

 Registering module :
 $source$
 $revision: 3169 $
 $date: 2007-12-07 16:40:53 -0700 (Fri, 07 Dec 2007) $
 Registration complete.

 &UTILITIES_NML
 TERMLEVEL= 2,LOGFILENAME=dart_log.out
 /

{ ... }

 Registering module :
 $source$
 $revision: 3169 $
```

(continues on next page)

(continued from previous page)

```
 $date: 2007-12-07 16:40:53 -0700 (Fri, 07 Dec 2007) $
 Registration complete.

 static_init_obs_sequence obs_sequence_nml values are
 &OBS_SEQUENCE_NML
 READ_BINARY_OBS_SEQUENCE= F,WRITE_BINARY_OBS_SEQUENCE= F
 /
 Input upper bound on number of observations in sequence
10000
 Input number of copies of data (0 for just a definition)
0
 Input number of quality control values per field (0 or greater)
0
 input a -1 if there are no more obs
0

 Registering module :
 $source$
 $revision: 3169 $
 $date: 2007-12-07 16:40:53 -0700 (Fri, 07 Dec 2007) $
 Registration complete.


 Registering module :
 $source$
 $revision: 3169 $
 $date: 2007-12-07 16:40:53 -0700 (Fri, 07 Dec 2007) $
 Registration complete.

 input obs kind: u =            1  v =            2  ps =            3  t =
           4  qv =            5  p =            6  w =            7  qr =
           8  Td =           10  rho =           11  Vr =          100  Ref =
         101  U10 =          200  V10 =          201  T2 =          202  Q2 =
         203
 input -1 times the state variable index for an identity observation
-2
 input time in days and seconds
1 0
 input error variance for this observation definition
1000000
 calling insert obs in sequence
 back from insert obs in sequence
 input a -1 if there are no more obs
-1
 Input filename for sequence (  set_def.out   usually works well)
set_def.out
 write_obs_seq  opening formatted file set_def.out
 write_obs_seq  closed file set_def.out
```

Two files are created. `set_def.out` is the empty template containing the metadata for the observation(s). `dart_log.out` contains run-time diagnostics from `create_obs_sequence`.

1.2 Create a (temporal) network of observations

`create_fixed_network_seq` creates an 'observation network definition' by extending the 'observation set definition' with the temporal attributes of the observations.

The first input is the name of the file created in the previous step, *i.e.* the name of the observation set definition that you've just created. It is possible to create sequences in which the observation sets are observed at regular intervals or irregularly in time. Here, all we need is a sequence that takes observations over a long period of time - indicated by entering a 1. Although the L63 system normally is defined as having a non-dimensional time step, the DART system arbitrarily defines the model timestep as being 3600 seconds. By declaring we have 1000 observations taken once per day, we create an observation sequence definition spanning 24000 'model' timesteps; sufficient to spin-up the model onto the attractor. Finally, enter a name for the 'observation sequence definition' file. Note again: there are no observation values present in this file. Just an observation type, location, time and the error characteristics. We are going to populate the observation sequence with the `perfect_model_obs` program.

```
[thoar@ghotiol work]$ ./create_fixed_network_seq
 Initializing the utilities module.
 Trying to log to unit           10
 Trying to open file dart_log.out

 Registering module :
 $source$
 $revision: 3169 $
 $date: 2007-12-07 16:40:53 -0700 (Fri, 07 Dec 2007) $
 Registration complete.

 { ... }

 static_init_obs_sequence obs_sequence_nml values are
 &OBS_SEQUENCE_NML
 READ_BINARY_OBS_SEQUENCE= F,WRITE_BINARY_OBS_SEQUENCE= F
 /
 Input filename for network definition sequence (usually  set_def.out  )
set_def.out
set_def.out

 Registering module :
 $source$
 $revision: 3169 $
 $date: 2007-12-07 16:40:53 -0700 (Fri, 07 Dec 2007) $
 Registration complete.


 Registering module :
 $source$
 $revision: 3169 $
 $date: 2007-12-07 16:40:53 -0700 (Fri, 07 Dec 2007) $
 Registration complete.

 To input a regularly repeating time sequence enter 1
 To enter an irregular list of times enter 2
1
 Input number of observation times in sequence
1000
 Input initial time in sequence
 input time in days and seconds
```

(continues on next page)

(continued from previous page)

```
1 0
 Input period of obs in sequence in days and seconds
1 0

       { ... }

         997
         998
         999
        1000
 What is output file name for sequence (  obs_seq.in   is recommended )
obs_seq.in
 write_obs_seq  opening formatted file obs_seq.in
 write_obs_seq  closed file obs_seq.in
```

1.3 Initialize the model onto the attractor

perfect_model_obs can now advance the arbitrary initial state for 24,000 timesteps to move it onto the attractor. perfect_model_obs uses the Fortran90 namelist input mechanism instead of (admittedly gory, but temporary) interactive input. All of the DART software expects the namelists to found in a file called input.nml. When you built the executable, an example namelist was created input.nml.perfect_model_obs_default that contains all of the namelist input for the executable. We must now rename and customize the namelist file for perfect_model_obs. Copy input.nml.perfect_model_obs_default to input.nml and edit it to look like the following:

&perfect_model_obs_nml async = 0, adv_ens_command = "./advance_ens.csh", obs_seq_in_file_name = "obs_seq.in", obs_seq_out_file_name = "obs_seq.out", start_from_restart = .false., output_restart = .*true.*, restart_in_file_name = "perfect_ics", restart_out_file_name = "perfect_restart", init_time_days = 0, init_time_seconds = 0, output_interval = 1 / &ensemble_manager_nml in_core = .true., single_restart_file_in = .true., single_restart_file_out = .true. / &assim_tools_nml filter_kind = 1, sort_obs_inc = .false., cov_inflate = -1.0, cov_inflate_sd = 0.05, sd_lower_bound = 0.05, deterministic_cov_inflate = .true., start_from_assim_restart = .false., assim_restart_in_file_name = 'assim_tools_ics' assim_restart_out_file_name = 'assim_tools_restart' do_parallel = 0, num_domains = 1, parallel_command = "./assim_filter.csh" / &cov_cutoff_nml select_localization = 1 / &reg_factor_nml select_regression = 1, input_reg_file = "time_mean_reg" / &obs_sequence_nml read_binary_obs_sequence = .false., write_binary_obs_sequence = .false. / &assim_model_nml read_binary_restart_files = .true., write_binary_restart_files = .true. / &model_nml sigma = 10.0, r = 28.0, b = 2.6666666666667, deltat = 0.01 time_step_days = 0 time_step_days = 3600 / &utilities_nml TERMLEVEL = 1 logfilename = 'dart_log.out' /

For the moment, only two namelists warrant explanation. Each namelists is covered in detail in the html files accompanying the source code for the module. perfect_model_obs_nml:

namelist variable	description
`async`	For the lorenz_63, simply ignore this. Leave it set to ‘0’
`obs_seq_in_file_name`	specifies the file name that results from running `create_fixed_network_seq`, i.e. the ‘observation sequence definition’ file.
`obs_seq_out_file_name`	specifies the output file name containing the ‘observation sequence’, finally populated with (perfect?) ‘observations’.
`start_from_restart`	When set to ‘false’, `perfect_model_obs` generates an arbitrary initial condition (which cannot be guaranteed to be on the L63 attractor).
`output_restart`	When set to ‘true’, `perfect_model_obs` will record the model state at the end of this integration in the file named by `restart_out_file_name`.
`restart_in_file_name`	is ignored when ‘start_from_restart’ is ‘false’.
`restart_out_file_name`	if `output_restart` is ‘true’, this specifies the name of the file containing the model state at the end of the integration.
`init_time_`*xxxx*	the start time of the integration.
`output_interval`	interval at which to save the model state.

`utilities_nml`:

namelist variable	description
`TERMLEVEL`	When set to ‘1’ the programs terminate when a ‘warning’ is generated. When set to ‘2’ the programs terminate only with ‘fatal’ errors.
`logfilename`	Run-time diagnostics are saved to this file. This namelist is used by all programs, so the file is opened in APPEND mode. Subsequent executions cause this file to grow.

Executing `perfect_model_obs` will integrate the model 24,000 steps and output the resulting state in the file `perfect_restart`. Interested parties can check the spinup in the `True_State.nc` file.

perfect_model_obs

Five files are created/updated:

`True_State.nc`		Contains the trajectory of the model
`perfect_restart`		Contains the model state at the end of the integration.
`obs_seq.out`		Contains the ‘perfect’ observations (since this is a spinup, they are of questionable value, at best).
`go_end_filter`		A ‘flag’ file that is not used by this model.
`dart_log.out`		**Appends** the run-time diagnostic output to an existing file, or creates a new file with the output.

2. Generate a set of ensemble initial conditions

The set of initial conditions for a ‘perfect model’ experiment is created by taking the spun-up state of the model (available in `perfect_restart`), running `perfect_model_obs` to generate the ‘true state’ of the experiment and a corresponding set of observations, and then feeding the same initial spun-up state and resulting observations into `filter`.

Generating ensemble initial conditions is achieved by changing a perfect_model_obs namelist parameter, copying `perfect_restart` to `perfect_ics`, and rerunning `perfect_model_obs`. This execution of `perfect_model_obs` will advance the model state from the end of the first 24,000 steps to the end of an additional 24,000 steps and place the final state in `perfect_restart`. The rest of the namelists in `input.nml` should remain unchanged.

&perfect_model_obs_nml async = 0, adv_ens_command = “./advance_ens.csh”, obs_seq_in_file_name = “obs_seq.in”, obs_seq_out_file_name = “obs_seq.out”, start_from_restart = *.true.*, output_restart = .true., restart_in_file_name = “perfect_ics”, restart_out_file_name = “perfect_restart”, init_time_days = 0, init_time_seconds = 0, output_interval = 1 /

cp perfect_restart perfect_ics perfect_model_obs

Five files are created/updated:

`True_State.nc`		Contains the trajectory of the model
`perfect_restart`		Contains the model state at the end of the integration.
`obs_seq.out`		Contains the ‘perfect’ observations.
`go_end_filter`		A ‘flag’ file that is not used by this model.
`dart_log.out`		**Appends** the run-time diagnostic output to an existing file, or creates a new file with the output.

Generating the ensemble

is done with the program `filter`, which also uses the Fortran90 namelist mechanism for input. It is now necessary to copy the `input.nml.filter_default` namelist to `input.nml`. Having the `perfect_model_obs` namelist in the `input.nml` does not hurt anything. In fact, I generally create a single `input.nml` that has all the namelist blocks in it by copying the `perfect_model_obs` block into the `input.nml.filter_default` and then rename it `input.nml`. This same namelist file may then also be used for `perfect_model_obs`.

&filter_nml async = 0, adv_ens_command = “./advance_ens.csh”, ens_size = 20, cutoff = 0.20, cov_inflate = 1.00, start_from_restart = .false., output_restart = *.true.*, obs_sequence_in_name = “obs_seq.out”, obs_sequence_out_name = “obs_seq.out”, restart_in_file_name = *“perfect_ics”*, restart_out_file_name = “filter_restart”, init_time_days = 0, init_time_seconds = 0, output_state_ens_mean = .true., output_state_ens_spread = .true., output_obs_ens_mean = .true., output_obs_ens_spread = .true., num_output_state_members = *20*, num_output_obs_members = *20*, output_interval = 1, num_groups = 1, confidence_slope = 0.0, outlier_threshold = -1.0, save_reg_series = .false. / &perfect_model_obs_nml async = 0, adv_ens_command = “./advance_ens.csh”, obs_seq_in_file_name = “obs_seq.in”, obs_seq_out_file_name = “obs_seq.out”, start_from_restart = .true., output_restart = .true., restart_in_file_name = “perfect_ics”, restart_out_file_name = “perfect_restart”, init_time_days = 0, init_time_seconds = 0, output_interval = 1 / &ensemble_manager_nml in_core = .true., single_restart_file_in = .true., single_restart_file_out = .true. / &assim_tools_nml filter_kind = 1, sort_obs_inc = .false., cov_inflate = -1.0, cov_inflate_sd = 0.05, sd_lower_bound = 0.05, deterministic_cov_inflate = .true., start_from_assim_restart = .false., assim_restart_in_file_name = ‘assim_tools_ics’ assim_restart_out_file_name = ‘assim_tools_restart’ do_parallel = 0, num_domains = 1, parallel_command = “./assim_filter.csh” / &cov_cutoff_nml select_localization = 1 / &reg_factor_nml select_regression = 1, input_reg_file = “time_mean_reg” / &obs_sequence_nml read_binary_obs_sequence = .false., write_binary_obs_sequence = .false. / &assim_model_nml read_binary_restart_files = .true., write_binary_restart_files = .true. / &model_nml sigma = 10.0, r = 28.0, b = 2.6666666666667, deltat = 0.01 time_step_days = 0 time_step_days = 3600 / &utilities_nml TERMLEVEL = 1 logfilename = ‘dart_log.out’ /

Only the non-obvious(?) entries for `filter_nml` will be discussed.

namelist variable	description
ens_size	Number of ensemble members. 20 is sufficient for most of the L63 exercises.
cutoff	to limit the impact of an observation, set to 0.0 (i.e. spin-up)
cov_inflate	A value of 1.0 results in no inflation.(spin-up)
start_from_restart	when '.false.', filter will generate its own set of initial conditions. It is important to note that the filter still makes use of perfect_ics by randomly perturbing these state variables.
num_output_state_members	may be a value from 0 to ens_size
num_output_obs_members	may be a value from 0 to ens_size
output_state_ens_mean	when '.true.' the mean of all ensemble members is output.
output_state_ens_spread	when '.true.' the spread of all ensemble members is output.
output_obs_ens_mean	when '.true.' the mean of all ensemble members observations is output.
output_obs_ens_spread	when '.true.' the spread of all ensemble members observations is output.
output_interval	Jeff - units for interval?

The filter is told to generate its own ensemble initial conditions since start_from_restart is '.false.'. However, it is important to note that the filter still makes use of perfect_ics which is set to be the restart_in_file_name. This is the model state generated from the first 24,000 step model integration by perfect_model_obs. Filter generates its ensemble initial conditions by randomly perturbing the state variables of this state.

The arguments output_state_ens_mean and output_state_ens_spread are '.true.' so that these quantities are output at every time for which there are observations (once a day here) and num_output_state_members means that the same diagnostic files, Posterior_Diag.nc and Prior_Diag.nc also contain values for all 20 ensemble members once a day. Once the namelist is set, execute filter to integrate the ensemble forward for 24,000 steps with the final ensemble state written to the filter_restart. Copy the perfect_model_obs restart file perfect_restart (the `true state') to perfect_ics, and the filter restart file filter_restart to filter_ics so that future assimilation experiments can be initialized from these spun-up states.

filter cp perfect_restart perfect_ics cp filter_restart filter_ics

The spin-up of the ensemble can be viewed by examining the output in the netCDF files True_State.nc generated by perfect_model_obs and Posterior_Diag.nc and Prior_Diag.nc generated by filter. To do this, see the detailed discussion of matlab diagnostics in Appendix I.

3. Simulate a particular observing system

Begin by using create_obs_sequence to generate an observation set in which each of the 3 state variables of L63 is observed with an observational error variance of 1.0 for each observation. To do this, use the following input sequence (the text including and after # is a comment and does not need to be entered):

set_def.out	# Output file name
1	# Number of sets
3	# Number of observations in set (x, y, and z)
1.0	# Variance of first observation
1	# First ob is identity observation of state variable 1 (x)
1.0	# Variance of second observation
2	# Second is identity observation of state variable 2 (y)
1.0	# Variance of third ob
3	# Identity ob of third state variable (z)

Now, generate an observation sequence definition by running create_fixed_network_seq with the following input sequence:

set_def.out	# Input observation set definition file
1	# Regular spaced observation interval in time
1000	# 1000 observation times
0, 43200	# First observation after 12 hours (0 days, 3600 * 12 seconds)
0, 43200	# Observations every 12 hours
obs_seq.in	# Output file for observation sequence definition

4. Generate a particular observing system and true state

An observation sequence file is now generated by running `perfect_model_obs` with the namelist values (unchanged from step 2):

&perfect_model_obs_nml async = 0, obs_seq_in_file_name = “obs_seq.in”, obs_seq_out_file_name = “obs_seq.out”, start_from_restart = .true., output_restart = .true., restart_in_file_name = “perfect_ics”, restart_out_file_name = “perfect_restart”, init_time_days = 0, init_time_seconds = 0, output_interval = 1 /

This integrates the model starting from the state in `perfect_ics` for 1000 12-hour intervals outputting synthetic observations of the three state variables every 12 hours and producing a netCDF diagnostic file, `True_State.nc`.

5. Filtering

Finally, `filter` can be run with its namelist set to:

&filter_nml async = 0, ens_size = 20, cutoff = *22222222.0*, cov_inflate = 1.00, start_from_restart = *.true.*, output_restart = .true., obs_sequence_file_name = “obs_seq.out”, restart_in_file_name = “*filter_ics*”, restart_out_file_name = “filter_restart”, init_time_days = 0, init_time_seconds = 0, output_state_ens_mean = .true., output_state_ens_spread = .true., num_output_ens_members = 20, output_interval = 1, num_groups = 1, confidence_slope = 0.0, output_obs_diagnostics = .false., get_mean_reg = .false., get_median_reg = .false. /

The large value for the cutoff allows each observation to impact all other state variables (see Appendix V for localization). `filter` produces two output diagnostic files, `Prior_Diag.nc` which contains values of the ensemble members, ensemble mean and ensemble spread for 12- hour lead forecasts before assimilation is applied and `Posterior_Diag.nc` which contains similar data for after the assimilation is applied (sometimes referred to as analysis values).

Now try applying all of the matlab diagnostic functions described in the Matlab Diagnostics section.

6.83.7 Matlab diagnostics

The output files are netCDF files, and may be examined with many different software packages. We happen to use Matlab, and provide our diagnostic scripts in the hopes that they are useful.

The Matlab diagnostic scripts and underlying functions reside in the `DART/matlab` directory. They are reliant on the public-domain netcdf toolbox from `http://woodshole.er.usgs.gov/staffpages/cdenham/public_html/MexCDF/nc4ml5.html` as well as the public-domain CSIRO matlab/netCDF interface from `http://www.marine.csiro.au/sw/matlab-netcdf.html`. If you do not have them installed on your system and want to use Matlab to peruse netCDF, you must follow their installation instructions.

Once you can access the `getnc` function from within Matlab, you can use our diagnostic scripts. It is necessary to prepend the location of the DART/matlab scripts to the matlabpath. Keep in mind the location of the netcdf operators on your system WILL be different from ours … and that’s OK.

```
0[269]0 ghotiol:/<5>models/lorenz_63/work]$ matlab -nojvm

                                             < M A T L A B >
                                 Copyright 1984-2002 The MathWorks, Inc.
                                     Version 6.5.0.180913a Release 13
                                               Jun 18 2002

  Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.

  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

>> which getnc
/contrib/matlab/matlab_netcdf_5_0/getnc.m
>>ls *.nc

ans =

Posterior_Diag.nc  Prior_Diag.nc  True_State.nc


>>path('../../../matlab',path)
>>which plot_ens_err_spread
../../../matlab/plot_ens_err_spread.m
>>help plot_ens_err_spread

  DART : Plots summary plots of the ensemble error and ensemble spread.
                         Interactively queries for the needed information.
                         Since different models potentially need different
                         pieces of information ... the model types are
                         determined and additional user input may be queried.

  Ultimately, plot_ens_err_spread will be replaced by a GUI.
  All the heavy lifting is done by PlotEnsErrSpread.

  Example 1 (for low-order models)

  truth_file = 'True_State.nc';
  diagn_file = 'Prior_Diag.nc';
  plot_ens_err_spread

>>plot_ens_err_spread
```

And the matlab graphics window will display the spread of the ensemble error for each state variable. The scripts are designed to do the "obvious" thing for the low-order models and will prompt for additional information if needed. The philosophy of these is that anything that starts with a lower-case *plot_some_specific_task* is intended to be user-callable and should handle any of the models. All the other routines in `DART/matlab` are called BY the high-level routines.

Matlab script	description
`plot_bins`	plots ensemble rank histograms
`plot_correl`	Plots space-time series of correlation between a given variable at a given time and other variables at all times in a n ensemble time sequence.
`plot_ens_err_spread`	Plots summary plots of the ensemble error and ensemble spread. Interactively queries for the needed information. Since different models potentially need different pieces of information … the model types are determined and additional user input may be queried.
`plot_ens_mean_time_series`	Queries for the state variables to plot.
`plot_ens_time_series`	Queries for the state variables to plot.
`plot_phase_space`	Plots a 3D trajectory of (3 state variables of) a single ensemble member. Additional trajectories may be superimposed.
`plot_total_err`	Summary plots of global error and spread.
`plot_var_var_correl`	Plots time series of correlation between a given variable at a given time and another variable at all times in an ensemble time sequence.

6.83.8 Bias, filter divergence and covariance inflation (with the l63 model)

One of the common problems with ensemble filters is filter divergence, which can also be an issue with a variety of other flavors of filters including the classical Kalman filter. In filter divergence, the prior estimate of the model state becomes too confident, either by chance or because of errors in the forecast model, the observational error characteristics, or approximations in the filter itself. If the filter is inappropriately confident that its prior estimate is correct, it will then tend to give less weight to observations than they should be given. The result can be enhanced overconfidence in the model's state estimate. In severe cases, this can spiral out of control and the ensemble can wander entirely away from the truth, confident that it is correct in its estimate. In less severe cases, the ensemble estimates may not diverge entirely from the truth but may still be too confident in their estimate. The result is that the truth ends up being farther away from the filter estimates than the spread of the filter ensemble would estimate. This type of behavior is commonly detected using rank histograms (also known as Talagrand diagrams). You can see the rank histograms for the L63 initial assimilation by using the matlab script `plot_bins`.

A simple, but surprisingly effective way of dealing with filter divergence is known as covariance inflation. In this method, the prior ensemble estimate of the state is expanded around its mean by a constant factor, effectively increasing the prior estimate of uncertainty while leaving the prior mean estimate unchanged. The program `filter` has a namelist parameter that controls the application of covariance inflation, `cov_inflate`. Up to this point, `cov_inflate` has been set to 1.0 indicating that the prior ensemble is left unchanged. Increasing `cov_inflate` to values greater than 1.0 inflates the ensemble before assimilating observations at each time they are available. Values smaller than 1.0 contract (reduce the spread) of prior ensembles before assimilating.

You can do this by modifying the value of `cov_inflate` in the namelist, (try 1.05 and 1.10 and other values at your discretion) and run the filter as above. In each case, use the diagnostic matlab tools to examine the resulting changes to the error, the ensemble spread (via rank histogram bins, too), etc. What kind of relation between spread and error is seen in this model?

6.83.9 Synthetic observations

Synthetic observations are generated from a `perfect' model integration, which is often referred to as the `truth' or a `nature run'. A model is integrated forward from some set of initial conditions and observations are generated as $y = H(x) + e$ where H is an operator on the model state vector, x, that gives the expected value of a set of observations, y, and e is a random variable with a distribution describing the error characteristics of the observing instrument(s) being simulated. Using synthetic observations in this way allows students to learn about assimilation algorithms while being isolated from the additional (extreme) complexity associated with model error and unknown observational error characteristics. In other words, for the real-world assimilation problem, the model has (often substantial) differences from what happens in the real system and the observational error distribution may be very complicated and is certainly not well known. Be careful to keep these issues in mind while exploring the capabilities of the ensemble filters with synthetic observations.

6.84 DART "pre_guam release" Documentation

6.85 Overview of DART

The Data Assimilation Research Testbed (DART) is designed to facilitate the combination of assimilation algorithms, models, and observation sets to allow increased understanding of all three. The DART programs have been compiled with the Intel 7.1 Fortran compiler and run on a linux compute-server. If your system is different, you will definitely need to read the Customizations section.

The latest software release – "pre_guam"[22 Mb – .tar.gz]

DART programs can require three different types of input. First, some of the DART programs, those for creating synthetic observational datasets, require interactive input from the keyboard. For simple cases, this interactive input can be made directly from the keyboard. In more complicated cases, a file containing the appropriate keyboard input can be created and this file can be directed to the standard input of the DART program. Second, many DART programs expect one or more input files in DART specific formats to be available. For instance, `perfect_model_obs` creates a synthetic observation set given a particular model and a description of a sequence of observations requires an input file that describes this observation sequence. At present, the observation files for DART are inefficient but human-readable ascii files in a custom format. Third, many DART modules (including main programs) make use of the Fortan90 namelist facility to obtain values of certain parameters at run-time. All programs look for a namelist input file called `input.nml` in the directory in which the program is executed. The `input.nml` file can contain a sequence of individual Fortran90 namelists which specify values of particular parameters for modules that compose the executable program. Unfortunately, the Fortran90 namelist interface is poorly defined in the language standard, leaving considerable leeway to compiler developers in implementing the facility. The Intel 7.1 compiler has some particularly unpleasant behavior when a namelist file contains an entry that is NOT defined in the program reading the namelist. Error behavior is unpredictable, but often results in read errors for other input files opened by DART programs. If you encounter run-time read errors, the first course of action should be to ensure the components of the namelist are actual components. Changing the names of the namelist components **will** create unpleasantries. DART provides a mechanism that automatically generates namelists with the default values for each program to be run.

DART uses the netCDF self-describing data format with a particular metadata convention to describe output that is used to analyze the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. A set of Matlab scripts, designed to produce graphical diagnostics from DART netCDF output files are available. DART users have also used ncview to create rudimentary graphical displays of output data fields. The NCO tools, produced by UCAR's Unidata group, are available to do operations like concatenating, slicing, and dicing of netCDF files.

6.85.1 Requirements: an F90 compiler

The DART software has been successfully built on several Linux/x86 platforms with the Intel Fortran Compiler 7.1 for Linux, which is free for individual scientific use. It has also been built and successfully run with the Portland Group Fortran Compiler (5.02), and again with the Intel 8.0.034 compiler. Since recompiling the code is a necessity to experiment with different models, there are no binaries to distribute.

DART uses the netCDF self-describing data format for the results of assimilation experiments. These files have the extension `.nc` and can be read by a number of standard data analysis tools. In particular, DART also makes use of the F90 interface to the library which is available through the `netcdf.mod` and `typesizes.mod` modules. *IMPORTANT*: different compilers create these modules with different "case" filenames, and sometimes they are not **both** installed into the expected directory. It is required that both modules be present. The normal place would be in the `netcdf/include` directory, as opposed to the `netcdf/lib` directory.

If the netCDF library does not exist on your system, you must build it (as well as the F90 interface modules). The library and instructions for building the library or installing from an RPM may be found at the netCDF home page: http://www.unidata.ucar.edu/packages/netcdf/ Pay particular attention to the compiler-specific patches that must be applied for the Intel Fortran Compiler. (Or the PG compiler, for that matter.)

The location of the netCDF library, `libnetcdf.a`, and the locations of both `netcdf.mod` and `typesizes.mod` will be needed by the makefile template, as described in the compiling section.

DART also uses the **very** common udunits library for manipulating units of physical quantities. If, somehow, it is not installed on your system, you will need to install it (instructions are available from Unidata's Downloads page).

The location of the udunits library, `libudunits.a`, will be needed by the makefile template, as described in the compiling section.

6.85.2 Unpacking the distribution

The DART source code is distributed as a compressed tar file. DART_fiji.tar.gz [22347692 bytes]. When untarred, the source tree will begin with a directory named `DART` and will be approximately 105 Mb. Compiling the code in this tree (as is usually the case) will necessitate much more space.

gunzip `DART_fiji.tar.gz` tar -xvf `DART_fiji.tar`

The code tree is very "bushy"; there are many directories of support routines, etc. but only a few directories involved with the customization and installation of the DART software. If you can compile and run ONE of the low-order models, you should be able to compile and run ANY of the low-order models. For this reason, we can focus on the Lorenz `63 model. Subsequently, the only directories with files to be modified to check the installation are: `DART/mkmf`, `DART/models/lorenz_63/work`, and `DART/matlab` (but only for analysis).

6.85.3 Customizing the build scripts – overview

DART executable programs are constructed using two tools: `make` and `mkmf`. The `make` utility is a relatively common piece of software that requires a user-defined input file that records dependencies between different source files. `make` then performs a hierarchy of actions when one or more of the source files is modified. The `mkmf` utility is a custom preprocessor that generates a `make` input file (named `Makefile`) and an example namelist `input.nml.mkmf` with the default values. The `Makefile` is designed specifically to work with object-oriented Fortran90 (and other languages) for systems like DART.

`mkmf` requires two separate input files. The first is a `template' file which specifies details of the commands required for a specific Fortran90 compiler and may also contain pointers to directories containing pre-compiled utilities required by the DART system. **This template file will need to be modified to reflect your system**. The second input file is a `path_names' file which includes a complete list of the locations (either relative or absolute) of all Fortran90 source files that are required to produce a particular DART program. Each 'path_names' file must contain a path for exactly one Fortran90 file containing a main program, but may contain any number of additional paths pointing to files containing Fortran90 modules. An `mkmf` command is executed which uses the 'path_names' file and the mkmf template file to produce a `Makefile` which is subsequently used by the standard `make` utility.

Shell scripts that execute the mkmf command for all standard DART executables are provided as part of the standard DART software. For more information on `mkmf` see the FMS mkmf description.
One of the benefits of using `mkmf` is that it also creates an example namelist file for each program. The example namelist is called `input.nml.mkmf`, so as not to clash with any exising `input.nml` that may exist in that directory.

Building and customizing the 'mkmf.template' file

A series of templates for different compilers/architectures exists in the `DART/mkmf/` directory and have names with extensions that identify either the compiler, the architecture, or both. This is how you inform the build process of the specifics of your system. Our intent is that you copy one that is similar to your system into `mkmf.template` and customize it. For the discussion that follows, knowledge of the contents of one of these templates (i.e. `mkmf.template.pgi`) is needed: (note that only the first few lines are shown here)

# Makefile template for PGI f90 FC = pgf90 CPPFLAGS = FFLAGS = -r8 -Ktrap=fp -pc 64 -I/usr/local/netcdf/include LD = pgf90 LDFLAGS = $(LIBS) LIBS = -L/usr/local/netcdf/lib -lnetcdf -L/usr/local/udunits-1.11.7/lib -ludunits LIST = -Mlist # you should never need to change any lines below. . . .

Essentially, each of the lines defines some part of the resulting `Makefile`. Since `make` is particularly good at sorting out dependencies, the order of these lines really doesn't make any difference. The `FC = pgf90` line ultimately defines the Fortran90 compiler to use, etc. The lines which are most likely to need site-specific changes start with `FFLAGS` and `LIBS`, which indicate where to look for the netCDF F90 modules and the location of the netCDF and udunits libraries.

` <fflags>`__

FFLAGS

Each compiler has different compile flags, so there is really no way to exhaustively cover this other than to say the templates as we supply them should work – depending on the location of the netCDF modules `netcdf.mod` and `typesizes.mod`. Change the `/usr/local/netcdf/include` string to reflect the location of your modules. The low-order models can be compiled without the `-r8` switch, but the `bgrid_solo` model cannot.

` <libs>`__

Libs

Modifying the `LIBS` value should be relatively straightforward.
Change the `/usr/local/netcdf/lib` string to reflect the location of your `libnetcdf.a`.
Change the `/usr/local/udunits-1.11.7/lib` string to reflect the location of your `libudunits.a`.

Customizing the 'path_names_*' file

Several `path_names_*` files are provided in the `work` directory for each specific model, in this case: `DART/models/lorenz_63/work`.

1. `path_names_create_obs_set_def`
2. `path_names_create_obs_sequence`
3. `path_names_perfect_model_obs`
4. `path_names_filter`

Since each model comes with its own set of files, no further customization is needed.

6.85.4 Building the Lorenz_63 DART project

Currently, DART executables are constructed in a `work` subdirectory under the directory containing code for the given model. In the top-level DART directory, change to the L63 work directory and list the contents:

cd DART/models/lorenz_63/work ls -1

With the result:

```
filter_ics
mkmf_create_obs_sequence
mkmf_create_obs_set_def
mkmf_filter
mkmf_perfect_model_obs
path_names_create_obs_sequence
path_names_create_obs_set_def
path_names_filter
path_names_perfect_model_obs
perfect_ics
```

There are four `mkmf_`*xxxxxx* files for the programs `create_obs_set_def`, `create_obs_sequence`, `perfect_model_obs`, and `filter` along with the corresponding `path_names_`*xxxxxx* files. You can examine the contents of one of the `path_names_`*xxxxxx* files, for instance `path_names_filter`, to see a list of the relative paths of all files that contain Fortran90 modules required for the program `filter` for the L63 model. All of these paths are relative to your `DART` directory. The first path is the main program (`filter.f90`) and is followed by all the Fortran90 modules used by this program.

The `mkmf_`*xxxxxx* scripts are cryptic but should not need to be modified – as long as you do not restructure the code tree (by moving directories, for example). The only function of the `mkmf_`*xxxxxx* script is to generate a `Makefile` and an `input.nml.mkmf` file. It is not supposed to compile anything:

csh mkmf_create_obs_set_def mv input.nml.mkmf input.nml.create_obs_set_def make

The first command generates an appropriate `Makefile` and the `input.nml.mkmf` file. The second saves the example namelist to a unique name (the next DART release will do this automatically – no harm is done by omitting this step) and the last command results in the compilation of a series of Fortran90 modules which ultimately produces an executable file: `create_obs_set_def`. Should you need to make any changes to the `DART/mkmf/mkmf.template`, you will need to regenerate the `Makefile`. A series of object files for each module compiled will also be left in the work directory, as some of these are undoubtedly needed by the build of the other DART components. You can proceed to create the other three programs needed to work with L63 in DART as follows:

csh mkmf_create_obs_sequence mv input.nml.mkmf input.nml.create_obs_sequence make csh mkmf_perfect_model_obs mv input.nml.mkmf input.nml.perfect_model_obs make csh mkmf_filter mv input.nml.mkmf input.nml.filter make

The result (hopefully) is that four executables now reside in your work directory. The most common problem is that the netCDF libraries and include files (particularly `typesizes.mod`) are not found. Edit the `DART/mkmf/mkmf.template`, recreate the `Makefile`, and try again.

program	purpose
`create_obs_set_def`	specify a (set) of observation characteristics taken by a particular (set of) instruments
`create_obs_sequence`	specify the temporal attributes of the observation sets
`perfect_model_obs`	spinup, generate "true state" for synthetic observation experiments, . . .
`filter`	perform experiments

6.85.5 Running Lorenz_63

This initial sequence of exercises includes detailed instructions on how to work with the DART code and allows investigation of the basic features of one of the most famous dynamical systems, the 3-variable Lorenz-63 model. The remarkable complexity of this simple model will also be used as a case study to introduce a number of features of a simple ensemble filter data assimilation system. To perform a synthetic observation assimilation experiment for the L63 model, the following steps must be performed (an overview of the process is given first, followed by detailed procedures for each step):

6.85.6 Experiment overview

1. Integrate the L63 model for a long time starting from arbitrary initial conditions to generate a model state that lies on the attractor. The ergodic nature of the L63 system means a 'lengthy' integration always converges to some point on the computer's finite precision representation of the model's attractor.
2. Generate a set of ensemble initial conditions from which to start an assimilation. Since L63 is ergodic, the ensemble members can be designed to look like random samples from the model's 'climatological distribution'. To generate an ensemble member, very small perturbations can be introduced to the state on the attractor generated by step 1. This perturbed state can then be integrated for a very long time until all memory of its initial condition can be viewed as forgotten. Any number of ensemble initial conditions can be generated by repeating this procedure.
3. Simulate a particular observing system by first creating an 'observation set definition' and then creating an 'observation sequence'. The 'observation set definition' describes the instrumental characteristics of the observations and the 'observation sequence' defines the temporal sequence of the observations.
4. Populate the 'observation sequence' with 'perfect' observations by integrating the model and using the information in the 'observation sequence' file to create simulated observations. This entails operating on the model state at the time of the observation with an appropriate forward operator (a function that operates on the model state vector to produce the expected value of the particular observation) and then adding a random sample from the observation error distribution specified in the observation set definition. At the same time, diagnostic output about the 'true' state trajectory can be created.
5. Assimilate the synthetic observations by running the filter; diagnostic output is generated.

1. Integrate the L63 model for a 'long' time

perfect_model_obs integrates the model for all the times specified in the 'observation sequence definition' file. To this end, begin by creating an 'observation sequence definition' file that spans a long time. Creating an 'observation sequence definition' file is a two-step procedure involving create_obs_sequence followed by create_fixed_network_seq. After they are both run, it is necessary to integrate the model with perfect_model_obs.

1.1 Create an observation set definition

create_obs_sequence creates an observation set definition, the time-independent part of an observation sequence. An observation set definition file only contains the location, type, and observational error characteristics (normally just the diagonal observational error variance) for a related set of observations. There are no actual observations, nor are there any times associated with the definition. For spin-up, we are only interested in integrating the L63 model, not in generating any particular synthetic observations. Begin by creating a minimal observation set definition.

In general, for the low-order models, only a single observation set need be defined. Next, the number of individual scalar observations (like a single surface pressure observation) in the set is needed. To spin-up an initial condition for the L63 model, only a single observation is needed. Next, the error variance for this observation must be entered. Since we do not need (nor want) this observation to have any impact on an assimilation (it will only be used for spinning up the model and the ensemble), enter a very large value for the error variance. An observation with a very large error variance has essentially no impact on deterministic filter assimilations like the default variety implemented in DART. Finally, the location and type of the observation need to be defined. For all types of models, the most elementary form of synthetic observations are called 'identity' observations. These observations are generated simply by adding a random sample from a specified observational error distribution directly to the value of one of the state variables. This defines the observation as being an identity observation of the first state variable in the L63 model. The program will respond by terminating after generating a file (generally named set_def.out) that defines the single identity observation of the first state variable of the L63 model. The following is a screenshot (much of the verbose logging has been left off for clarity), the user input looks *like this*.

```
[unixprompt]$ ./create_obs_set_def
 Initializing the utilities module.
 Registering module :
 $source$
 $revision: 3169 $
 $date: 2007-12-07 16:40:53 -0700 (Fri, 07 Dec 2007) $
 Registration complete.

 &UTILITIES_NML
 TERMLEVEL =            2,
 LOGFILENAME = dart_log.out
 /

 Registering module :
 $source$
 $revision: 3169 $
 $date: 2007-12-07 16:40:53 -0700 (Fri, 07 Dec 2007) $
 Registration complete.

 Input the filename for output of observation set_def_list? [set_def.out]
set_def.out

{ ... }
```

(continues on next page)

(continued from previous page)

```

 Input the number of unique observation sets you might define
1
 How many observations in set            1
1
 Defining observation            1
 Input error variance for this observation definition
1000000
 Input an integer index if this is identity observation, else -1
1

 Registering module :
 $source$
 $revision: 3169 $
 $date: 2007-12-07 16:40:53 -0700 (Fri, 07 Dec 2007) $
 Registration complete.

 set_def.out successfully created.
 Terminating normally.
```

1.2 Create an observation sequence definition

`create_obs_sequence` creates an 'observation sequence definition' by extending the 'observation set definition' with the temporal attributes of the observations.

The first input is the name of the file created in the previous step, i.e. the name of the observation set definition that you've just created. It is possible to create sequences in which the observation sets are observed at regular intervals or irregularly in time. Here, all we need is a sequence that takes observations over a long period of time - indicated by entering a 1. Although the L63 system normally is defined as having a non-dimensional time step, the DART system arbitrarily defines the model timestep as being 3600 seconds. By declaring we have 1000 observations taken once per day, we create an observation sequence definition spanning 24000 'model' timesteps; sufficient to spin-up the model onto the attractor. Finally, enter a name for the 'observation sequence definition' file. Note again: there are no observation values present in this file. Just an observation type, location, time and the error characteristics. We are going to populate the observation sequence with the `perfect_model_obs` program.

```
[thoar@ghotiol work]$ ./create_obs_sequence
 Registering module :
 $source$
 $revision: 3169 $
 $date: 2007-12-07 16:40:53 -0700 (Fri, 07 Dec 2007) $
 Registration complete.

 &UTILITIES_NML
 TERMLEVEL =            2,
 LOGFILENAME = dart_log.out
 /

 Registering module :
 $source$
 $revision: 3169 $
 $date: 2007-12-07 16:40:53 -0700 (Fri, 07 Dec 2007) $
 Registration complete.

```

(continues on next page)

(continued from previous page)

```
 What is name of set_def_list? [set_def.out]
set_def.out

 { ... }

 Setting times for obs_def             1
 To input a regularly repeating time sequence enter 1
 To enter an irregular list of times enter 2
1
 Input number of observations in sequence
1000
 Input time of initial ob in sequence in days and seconds
1, 0
 Input period of obs in days and seconds
1, 0
 time             1  is             0           1
 time             2  is             0           2
 time             3  is             0           3
...
 time           998  is             0         998
 time           999  is             0         999
 time          1000  is             0        1000
 Input file name for output of obs_sequence? [obs_seq.in]
obs_seq.in
```

1.3 Initialize the model onto the attractor

`perfect_model_obs` can now advance the arbitrary initial state for 24,000 timesteps to move it onto the attractor. `perfect_model_obs` uses the Fortran90 namelist input mechanism instead of (admittedly gory, but temporary) interactive input. All of the DART software expects the namelists to found in a file called `input.nml`. When you built the executable, an example namelist was created `input.nml.mkmf` that contains all of the namelist input for the executable. If you followed the example, each namelist was saved to a unique name. We must now rename and edit the namelist file for `perfect_model_obs`. Copy `input.nml.perfect_model_obs` to `input.nml` and edit it to look like the following:

&perfect_model_obs_nml async = 0, obs_seq_in_file_name = “obs_seq.in”, obs_seq_out_file_name = “obs_seq.out”, start_from_restart = .false., output_restart = *.true.*, restart_in_file_name = “perfect_ics”, restart_out_file_name = “perfect_restart”, init_time_days = 0, init_time_seconds = 0, output_interval = 1 &end &assim_tools_nml prior_spread_correction = .false., filter_kind = 1, slope_threshold = 1.0 &end &cov_cutoff_nml select_localization = 1 &end &assim_model_nml binary_restart_files = .true. &end &model_nml sigma = 10.0, r = 28.0, b = 2.6666666666667, deltat = 0.01 &end &utilities_nml TERMLEVEL = 1 logfilename = ‘dart_log.out’ &end

For the moment, only two namelists warrant explanation. Each namelists is covered in detail in the html files accompanying the source code for the module.

perfect_model_obs_nml

namelist variable	description
async	For the lorenz_63, simply ignore this. Leave it set to '0'
obs_seq_in_file_name	specifies the file name that results from running create_obs_sequence, i.e. the 'observation sequence definition' file.
obs_seq_out_file_name	specifies the output file name containing the 'observation sequence', finally populated with (perfect?) 'observations'.
start_from_restart	When set to 'false', perfect_model_obs generates an arbitrary initial condition (which cannot be guaranteed to be on the L63 attractor).
output_restart	When set to 'true', perfect_model_obs will record the model state at the end of this integration in the file named by restart_out_file_name.
restart_in_file_name	is ignored when 'start_from_restart' is 'false'.
restart_out_file_name	if output_restart is 'true', this specifies the name of the file containing the model state at the end of the integration.
init_time_*xxxx*	the start time of the integration.
output_interval	interval at which to save the model state.

utilities_nml

namelist variable	description
TERMLEVEL	When set to '1' the programs terminate when a 'warning' is generated. When set to '2' the programs terminate only with 'fatal' errors.
logfilename	Run-time diagnostics are saved to this file. This namelist is used by all programs, so the file is opened in APPEND mode. Subsequent executions cause this file to grow.

Executing perfect_model_obs will integrate the model 24,000 steps and output the resulting state in the file perfect_restart. Interested parties can check the spinup in the True_State.nc file.

perfect_model_obs

2. Generate a set of ensemble initial conditions

The set of initial conditions for a 'perfect model' experiment is created by taking the spun-up state of the model (available in perfect_restart), running perfect_model_obs to generate the 'true state' of the experiment and a corresponding set of observations, and then feeding the same initial spun-up state and resulting observations into filter.
Generating ensemble initial conditions is achieved by changing a perfect_model_obs namelist parameter, copying perfect_restart to perfect_ics, and rerunning perfect_model_obs. This execution of perfect_model_obs will advance the model state from the end of the first 24,000 steps to the end of an additional 24,000 steps and place the final state in perfect_restart. The rest of the namelists in input.nml should remain unchanged.

&perfect_model_obs_nml async = 0, obs_seq_in_file_name = "obs_seq.in", obs_seq_out_file_name = "obs_seq.out", start_from_restart = *.true.*, output_restart = .true., restart_in_file_name = "perfect_ics", restart_out_file_name = "perfect_restart", init_time_days = 0, init_time_seconds = 0, output_interval = 1 &end

cp perfect_restart perfect_ics perfect_model_obs

A True_State.nc file is also created. It contains the 'true' state of the integration.

Generating the ensemble

is done with the program `filter`, which also uses the Fortran90 namelist mechanism for input. It is now necessary to copy the `input.nml.filter` namelist to `input.nml` or you may simply insert the `filter_nml` namelist into the existing `input.nml`. Having the `perfect_model_obs` namelist in the input.nml does not hurt anything. In fact, I generally create a single `input.nml` that has all the namelist blocks in it.

&perfect_model_obs_nml async = 0, obs_seq_in_file_name = "obs_seq.in", obs_seq_out_file_name = "obs_seq.out", start_from_restart = .true., output_restart = .true., restart_in_file_name = "perfect_ics", restart_out_file_name = "perfect_restart", init_time_days = 0, init_time_seconds = 0, output_interval = 1 &end &assim_tools_nml prior_spread_correction = .false., filter_kind = 1, slope_threshold = 1.0 &end &cov_cutoff_nml select_localization = 1 &end &assim_model_nml binary_restart_files = .true. &end &model_nml sigma = 10.0, r = 28.0, b = 2.6666666666667 deltat = 0.01 &end &utilities_nml TERMLEVEL = 1 logfilename = 'dart_log.out' &end &reg_factor_nml select_regression = 1, input_reg_file = "time_mean_reg" &end &filter_nml async = 0, ens_size = 20, cutoff = 0.20, cov_inflate = 1.00, start_from_restart = .false., output_restart = .*true.*, obs_sequence_file_name = "obs_seq.out", restart_in_file_name = "perfect_ics", restart_out_file_name = "filter_restart", init_time_days = 0, init_time_seconds = 0, output_state_ens_mean = .true., output_state_ens_spread = .true., num_output_ens_members = *20*, output_interval = 1, num_groups = 1, confidence_slope = 0.0, output_obs_diagnostics = .false., get_mean_reg = .false., get_median_reg = .false. &end

Only the non-obvious(?) entries for `filter_nml` will be discussed.

namelist variable	description
`ens_size`	Number of ensemble members. 20 is sufficient for most of the L63 exercises.
`cutoff`	to limit the impact of an observation, set to 0.0 (i.e. spin-up)
`cov_inflate`	A value of 1.0 results in no inflation.(spin-up)
`start_from_restart`	when '.false.', `filter` will generate its own set of initial conditions. It is important to note that the filter still makes use of `perfect_ics` by randomly perturbing these state variables.
`num_output_ens_members`	may be a value from 0 to `ens_size`
`output_state_ens_mean`	when '.true.' the mean of all ensemble members is output.
`output_state_ens_spread`	when '.true.' the spread of all ensemble members is output.
`output_interval`	Jeff - units for interval?

The filter is told to generate its own ensemble initial conditions since `start_from_restart` is '.false.'. However, it is important to note that the filter still makes use of `perfect_ics` which is set to be the `restart_in_file_name`. This is the model state generated from the first 24,000 step model integration by `perfect_model_obs`. `Filter` generates its ensemble initial conditions by randomly perturbing the state variables of this state.

The arguments `output_state_ens_mean` and `output_state_ens_spread` are '.true.' so that these quantities are output at every time for which there are observations (once a day here) and `num_output_ens_members` means that the same diagnostic files, `Posterior_Diag.nc` and `Prior_Diag.nc` also contain values for all 20 ensemble members once a day. Once the namelist is set, execute `filter` to integrate the ensemble forward for 24,000 steps with the final ensemble state written to the `filter_restart`. Copy the `perfect_model_obs` restart file `perfect_restart` (the `true state') to `perfect_ics`, and the `filter` restart file `filter_restart` to `filter_ics` so that future assimilation experiments can be initialized from these spun-up states.

filter cp perfect_restart perfect_ics cp filter_restart filter_ics

The spin-up of the ensemble can be viewed by examining the output in the netCDF files `True_State.nc` generated by `perfect_model_obs` and `Posterior_Diag.nc` and `Prior_Diag.nc` generated by `filter`. To do this, see the detailed discussion of matlab diagnostics in Appendix I.

3. Simulate a particular observing system

Begin by using `create_obs_set_def` to generate an observation set in which each of the 3 state variables of L63 is observed with an observational error variance of 1.0 for each observation. To do this, use the following input sequence (the text including and after # is a comment and does not need to be entered):

set_def.out	# Output file name
1	# Number of sets
3	# Number of observations in set (x, y, and z)
1.0	# Variance of first observation
1	# First ob is identity observation of state variable 1 (x)
1.0	# Variance of second observation
2	# Second is identity observation of state variable 2 (y)
1.0	# Variance of third ob
3	# Identity ob of third state variable (z)

Now, generate an observation sequence definition by running `create_obs_sequence` with the following input sequence:

set_def.out	# Input observation set definition file
1	# Regular spaced observation interval in time
1000	# 1000 observation times
0, 43200	# First observation after 12 hours (0 days, 3600 * 12 seconds)
0, 43200	# Observations every 12 hours
obs_seq.in	# Output file for observation sequence definition

4. Generate a particular observing system and true state

An observation sequence file is now generated by running `perfect_model_obs` with the namelist values (unchanged from step 2):

&perfect_model_obs_nml async = 0, obs_seq_in_file_name = “obs_seq.in”, obs_seq_out_file_name = “obs_seq.out”, start_from_restart = .true., output_restart = .true., restart_in_file_name = “perfect_ics”, restart_out_file_name = “perfect_restart”, init_time_days = 0, init_time_seconds = 0, output_interval = 1 &end

This integrates the model starting from the state in `perfect_ics` for 1000 12-hour intervals outputting synthetic observations of the three state variables every 12 hours and producing a netCDF diagnostic file, `True_State.nc`.

5. Filtering

Finally, `filter` can be run with its namelist set to:

&filter_nml async = 0, ens_size = 20, cutoff = *22222222.0*, cov_inflate = 1.00, start_from_restart = *.true.*, output_restart = .true., obs_sequence_file_name = “obs_seq.out”, restart_in_file_name = “*filter_ics*”, restart_out_file_name = “filter_restart”, init_time_days = 0, init_time_seconds = 0, output_state_ens_mean = .true., output_state_ens_spread = .true., num_output_ens_members = 20, output_interval = 1, num_groups = 1, confidence_slope = 0.0, output_obs_diagnostics = .false., get_mean_reg = .false., get_median_reg = .false. &end

The large value for the cutoff allows each observation to impact all other state variables (see Appendix V for localization). `filter` produces two output diagnostic files, `Prior_Diag.nc` which contains values of the ensemble members, ensemble mean and ensemble spread for 12- hour lead forecasts before assimilation is applied and `Posterior_Diag.nc` which contains similar data for after the assimilation is applied (sometimes referred to as analysis values).

Now try applying all of the matlab diagnostic functions described in the Matlab Diagnostics section.

6.85.7 Matlab® diagnostics

The output files are netCDF files, and may be examined with many different software packages. We happen to use Matlab®, and provide our diagnostic scripts in the hopes that they are useful.

The Matlab diagnostic scripts and underlying functions reside in the `DART/matlab` directory. They are reliant on the public-domain netcdf toolbox from `http://woodshole.er.usgs.gov/staffpages/cdenham/public_html/MexCDF/nc4ml5.html` as well as the public-domain CSIRO matlab/netCDF interface from `http://www.marine.csiro.au/sw/matlab-netcdf.html`. If you do not have them installed on your system and want to use Matlab to peruse netCDF, you must follow their installation instructions.

Once you can access the `getnc` function from within Matlab, you can use our diagnostic scripts. It is necessary to prepend the location of the DART/matlab scripts to the matlabpath. Keep in mind the location of the netcdf operators on your system WILL be different from ours … and that's OK.

```
0[269]0 ghotiol:/<5>models/lorenz_63/work]$ matlab -nojvm

                                             < M A T L A B >
                                 Copyright 1984-2002 The MathWorks, Inc.
                                     Version 6.5.0.180913a Release 13
                                               Jun 18 2002

  Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.

  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

>> which getnc
/contrib/matlab/matlab_netcdf_5_0/getnc.m
>>ls *.nc

ans =

Posterior_Diag.nc  Prior_Diag.nc  True_State.nc


>>path('../../../matlab',path)
>>which plot_ens_err_spread
../../../matlab/plot_ens_err_spread.m
>>help plot_ens_err_spread

  DART : Plots summary plots of the ensemble error and ensemble spread.
                         Interactively queries for the needed information.
                         Since different models potentially need different
                         pieces of information ... the model types are
                         determined and additional user input may be queried.

  Ultimately, plot_ens_err_spread will be replaced by a GUI.
  All the heavy lifting is done by PlotEnsErrSpread.

  Example 1 (for low-order models)

  truth_file = 'True_State.nc';
  diagn_file = 'Prior_Diag.nc';
  plot_ens_err_spread
```

(continues on next page)

(continued from previous page)

```

>>plot_ens_err_spread
```

And the matlab graphics window will display the spread of the ensemble error for each state variable. The scripts are designed to do the “obvious” thing for the low-order models and will prompt for additional information if needed. The philosophy of these is that anything that starts with a lower-case *plot_some_specific_task* is intended to be user-callable and should handle any of the models. All the other routines in `DART/matlab` are called BY the high-level routines.

Matlab script	description
`plot_bins`	plots ensemble rank histograms
`plot_correl`	Plots space-time series of correlation between a given variable at a given time and other variables at all times in a n ensemble time sequence.
`plot_ens_err_spread`	Plots summary plots of the ensemble error and ensemble spread. Interactively queries for the needed information. Since different models potentially need different pieces of information ... the model types are determined and additional user input may be queried.
`plot_ens_mean_time_series`	Queries for the state variables to plot.
`plot_ens_time_series`	Queries for the state variables to plot.
`plot_phase_space`	Plots a 3D trajectory of (3 state variables of) a single ensemble member. Additional trajectories may be superimposed.
`plot_total_err`	Summary plots of global error and spread.
`plot_var_var_correl`	Plots time series of correlation between a given variable at a given time and another variable at all times in an ensemble time sequence.

6.85.8 Bias, filter divergence and covariance inflation (with the l63 model)

One of the common problems with ensemble filters is filter divergence, which can also be an issue with a variety of other flavors of filters including the classical Kalman filter. In filter divergence, the prior estimate of the model state becomes too confident, either by chance or because of errors in the forecast model, the observational error characteristics, or approximations in the filter itself. If the filter is inappropriately confident that its prior estimate is correct, it will then tend to give less weight to observations than they should be given. The result can be enhanced overconfidence in the model’s state estimate. In severe cases, this can spiral out of control and the ensemble can wander entirely away from the truth, confident that it is correct in its estimate. In less severe cases, the ensemble estimates may not diverge entirely from the truth but may still be too confident in their estimate. The result is that the truth ends up being farther away from the filter estimates than the spread of the filter ensemble would estimate. This type of behavior is commonly detected using rank histograms (also known as Talagrand diagrams). You can see the rank histograms for the L63 initial assimilation by using the matlab script `plot_bins`.

A simple, but surprisingly effective way of dealing with filter divergence is known as covariance inflation. In this method, the prior ensemble estimate of the state is expanded around its mean by a constant factor, effectively increasing the prior estimate of uncertainty while leaving the prior mean estimate unchanged. The program `filter` has a namelist parameter that controls the application of covariance inflation, `cov_inflate`. Up to this point, `cov_inflate` has been set to 1.0 indicating that the prior ensemble is left unchanged. Increasing `cov_inflate` to values greater than 1.0 inflates the ensemble before assimilating observations at each time they are available. Values smaller than 1.0 contract (reduce the spread) of prior ensembles before assimilating.

You can do this by modifying the value of `cov_inflate` in the namelist, (try 1.05 and 1.10 and other values at your discretion) and run the filter as above. In each case, use the diagnostic matlab tools to examine the resulting changes to the error, the ensemble spread (via rank histogram bins, too), etc. What kind of relation between spread and error is seen in this model?

6.85.9 Synthetic observations

Synthetic observations are generated from a `perfect' model integration, which is often referred to as the `truth' or a `nature run'. A model is integrated forward from some set of initial conditions and observations are generated as $y = H(x) + e$ where H is an operator on the model state vector, x, that gives the expected value of a set of observations, y, and e is a random variable with a distribution describing the error characteristics of the observing instrument(s) being simulated. Using synthetic observations in this way allows students to learn about assimilation algorithms while being isolated from the additional (extreme) complexity associated with model error and unknown observational error characteristics. In other words, for the real-world assimilation problem, the model has (often substantial) differences from what happens in the real system and the observational error distribution may be very complicated and is certainly not well known. Be careful to keep these issues in mind while exploring the capabilities of the ensemble filters with synthetic observations.

6.86 DART Tutorial

The DART Tutorial is intended to aid in the understanding of ensemble data assimilation theory and consists of step-by-step concepts and companion exercises with DART. The diagnostics in the tutorial use Matlab® (to see how to configure your environment to use Matlab and the DART diagnostics, see the documentation for Configuring Matlab® for netCDF & DART).

Section 1	[pdf]	Filtering For a One Variable System
Section 2	[pdf]	The DART Directory Tree
Section 3	[pdf]	DART Runtime Control and Documentation
Section 4	[pdf]	How should observations of a state variable impact an unobserved state variable? Multivariate assimilation.
Section 5	[pdf]	Comprehensive Filtering Theory: Non-Identity Observations and the Joint Phase Space
Section 6	[pdf]	Other Updates for An Observed Variable
Section 7	[pdf]	Some Additional Low-Order Models
Section 8	[pdf]	Dealing with Sampling Error
Section 9	[pdf]	More on Dealing with Error; Inflation
Section 10	[pdf]	Regression and Nonlinear Effects
Section 11	[pdf]	Creating DART Executables
Section 12	[pdf]	Adaptive Inflation
Section 13	[pdf]	Hierarchical Group Filters and Localization
Section 14	[pdf]	Observation Quality Control
Section 15	[pdf]	DART Experiments: Control and Design
Section 16	[pdf]	Diagnostic Output
Section 17	[pdf]	Creating Observation Sequences
Section 18	[pdf]	Lost in Phase Space: The Challenge of Not Knowing the Truth
Section 19	[pdf]	DART-Compliant Models and Making Models Compliant: Coming Soon
Section 20	[pdf]	Model Parameter Estimation
Section 21	[pdf]	Observation Types and Observing System Design
Section 22	[pdf]	Parallel Algorithm Implementation: Coming Soon
Section 23	[pdf]	Location Module Design
Section 24		Fixed Lag Smoother (not available yet)
Section 25	[pdf]	A Simple 1D Advection Model: Tracer Data Assimilation

6.87 DART_LAB Tutorial

6.87.1 Overview

The files in this directory contain PDF tutorial materials on DART, and Matlab exercises. See below for links to the PDF files and a list of the corresponding matlab scripts.

This tutorial begins at a more introductory level than the materials in the tutorial directory, and includes hands-on exercises at several points. In a workshop setting, these materials and exercises took about 1.5 days to complete.

6.87.2 DART tutorial presentations

Here are the PDF files for the presentation part of the tutorial:

- Section 1: The basics in 1D.
- Section 2: How should observations of a state variable impact an unobserved state variable? Multivariate assimilation.
- Section 3: Sampling error and localization.
- Section 4: The Ensemble Kalman Filter (Perturbed Observations).
- Section 5: Adaptive Inflation.

6.87.3 Matlab hands-on exercises

In the `matlab` subdirectory are a set of Matlab scripts and GUI (graphical user interface) programs which are exercises that go with the tutorial. Each is interactive with settings that can be changed and rerun to explore various options. A valid Matlab license is needed to run these scripts.

The exercises use the following functions:

- gaussian_product
- oned_model
- oned_ensemble
- run_lorenz_63
- run_lorenz_96
- twod_ensemble

To run these, cd into the DART_LAB/matlab directory, start matlab, and type the names at the prompt.

6.88 MODULE location_mod (channel)

6.88.1 Overview

THIS HAS NOT BEEN UPDATED YET - ONLY COPIED FROM 3D SPHERE VERSION

THIS HAS NOT BEEN UPDATED YET - ONLY COPIED FROM 3D SPHERE VERSION

THIS HAS NOT BEEN UPDATED YET - ONLY COPIED FROM 3D SPHERE VERSION

The DART framework needs to be able to compute distances between locations, to pass location information to and from the model interface code (model_mod.f90), and to be able to read and write location information to files. DART isolates all this location information into separate modules so that the main algorithms can operate with the same code independent of whether the model uses latitude/longitude/height, 1D unit sphere coordinates, cylindrical coordinates, etc. DART provides about half a dozen possible coordinate systems, and others can be added. The most common one for geophysical models is this one: threed_sphere.

This location module provides a representation of a physical location on a 3-D spherical shell, using latitude and longitude plus a vertical component with choices of vertical coordinate type such as pressure or height in meters. A type that abstracts the location is provided along with operators to set, get, read, write, and compute distances between locations. This is a member of a class of similar location modules that provide the same abstraction for different represenations of physical space.

Location-independent code

All types of location modules define the same module name `location_mod`. Therefore, the DART framework and any user code should include a Fortran 90 `use` statement of `location_mod`. The selection of which location module will be compiled into the program is controlled by which source file name is specified in the `path_names_xxx` file, which is used by the `mkmf_xxx` scripts.

All types of location modules define the same Fortran 90 derived type `location_type`. Programs that need to pass location information to subroutines but do not need to interpret the contents can declare, receive, and pass this derived type around in their code independent of which location module is specified at compile time. Model and location-independent utilities should be written in this way. However, as soon as the contents of the location type needs to be accessed by user code then it becomes dependent on the exact type of location module that it is compiled with.

Usage of distance routines

Regardless of the fact that the distance subroutine names include the string 'obs', there is nothing specific to observations in these routines. They work to compute distances between any set of locations. The most frequent use of these routines in the filter code is to compute the distance between a single observation and items in the state vector, and also between a single observation and other nearby observations. However, any source for locations is supported.

In simpler location modules (like the `oned` version) there is no need for anything other than a brute force search between the base location and all available state vector locations. However in the case of large geophysical models which typically use the `threed_sphere` locations code, the brute-force search time is prohibitive. The location code pre-processes all locations into a set of *bins* and then only needs to search the lists of locations in nearby bins when looking for locations that are within a specified distance.

The expected calling sequence of the `get_close` routines is as follows:

```
call get_close_maxdist_init()  ! is called before get_close_obs_init()
call get_close_obs_init()

call get_close_obs()           ! called many, many times

call get_close_obs_destroy()
```

In the `threed_sphere` implementation the first routine initializes some data structures, the second one bins up the list of locations, and then the third one is called multiple times to find all locations within a given radius of some reference location, and to optionally compute the exact separation distance from the reference location. The last routine deallocates the space. See the documentation below for the specific details for each routine.

All 4 of these routines must be present in every location module but in most other versions all but `get_close_obs()` are stubs. In this `threed_sphere` version of the locations module all are fully implemented.

Interaction with model_mod.f90 code

The filter and other DART programs could call the `get_close` routines directly, but typically do not. They declare them (in a `use` statement) to be in the `model_mod` module, and all model interface modules are required to supply them. However in many cases the model_mod only needs to contain another `use` statement declaring them to come from the `location_mod` module. Thus they 'pass through' the model_mod but the user does not need to provide a subroutine or any code for them.

However, if the model interface code wants to intercept and alter the default behavior of the get_close routines, it is able to. Typically the model_mod still calls the location_mod routines and then adjusts the results before passing them back to the calling code. To do that, the model_mod must be able to call the routines in the location_mod which have the same names as the subroutines it is providing. To allow the compiler to distinguish which routine is to be called where, we use the Fortran 90 feature which allows a module routine to be renamed in the use statement. For example, a common case is for the model_mod to want to supply additions to the get_close_obs() routine only. At the top of the model_mod code it would declare:

```
use location_mod, only :: location_get_close_obs => get_close_obs,    &
                          get_close_maxdist_init, get_close_obs_init, &
                          get_close_obs_destroy
```

That makes calls to the maxdist_init, init, and destroy routines simply pass through to the code in the location_mod, but the model_mod must supply a get_close_obs() subroutine. When it wants to call the code in the location_mod it calls `location_get_close_obs()`.

One use pattern is for the model_mod to call the location get_close_obs() routine without the `dist` argument. This returns a list of any potentially close locations without computing the exact distance from the base location. At this point the list of locations is a copy and the model_mod routine is free to alter the list in any way it chooses: it can change the locations to make certain types of locations appear closer or further away from the base location; it can convert the vertical coordinates into a common coordinate type so that calls to the `get_dist()` routine can do full 3d distance computations and not just 2d (the vertical coordinates must match between the base location and the locations in the list in order to compute a 3d distance). Then typically the model_mod code loops over the list calling the `get_dist()` routine to get the actual distances to be returned to the calling code. To localize in the vertical in a particular unit type, this is the place where the conversion to that vertical unit should be done.

Horizontal distance only

If *horiz_distance_only* is .true. in the namelist, then the vertical coordinate is ignored and only the great-circle distance between the two locations is computed, as if they were both on the surface of the sphere.

If *horiz_distance_only* is .false. in the namelist then the appropriate normalization constant determines the relative impact of vertical and horizontal separation. Since only a single localization distance is specified, and the vertical scales might have very different distance characteristics, the vert_normalization_xxx values can be used to scale the vertical appropriately to control the desired influence of observations in the vertical.

Precomputation for run-time search efficiency

For search efficiency all locations are pre-binned. The surface of the sphere is divided up into *nlon* by *nlat* boxes and the index numbers of all items (both state vector entries and observations) are stored in the appropriate box. To locate all points close to a given location, only the locations listed in the boxes within the search radius must be checked. This speeds up the computations, for example, when localization controls which state vector items are impacted by any given observation. The search radius is the localization distance and only those state vector items in boxes closer than the radius to the observation location are processed.

The default values have given good performance on many of our existing model runs, but for tuning purposes the box counts have been added to the namelist to allow adjustment. By default the code prints some summary information about how full the average box is, how many are empty, and how many items were in the box with the largest count. The namelist value *output_box_info* can be set to .true. to get even more information about the box statistics. The best performance will be obtained somewhere between two extremes; the worst extreme is all the points are located in just a few boxes. This degenerates into a (slow) linear search through the index list. The other extreme is a large number of empty or sparsely filled boxes. The overhead of creating, managing, and searching a long list of boxes will impact performance. The best performance lies somewhere in the middle, where each box contains a reasonable number of values, more or less evenly distributed across boxes. The absolute numbers for best performance will certainly vary from case to case.

For latitude, the *nlat* boxes are distributed evenly across the actual extents of the data. (Locations are in radians, so the maximum limits are the poles at $-\pi/2$ and $+\pi/2$. For longitude, the code automatically determines if the data is spread around more than half the sphere, and if so, the boxes are distributed evenly across the entire sphere (longitude range 0 to 2π). If the data spans less than half the sphere in longitude, the actual extent of the data is determined (including correctly handling the cyclic boundary at 0) and the boxes are distributed only within the data extent. This simplifies the actual distance calculations since the distance from the minimum longitude box to the maximum latitude box cannot be shorter going the other way around the sphere. In practice, for a global model the boxes are evenly distributed across the entire surface of the sphere. For local or regional models, the boxes are distributed only across the the extent of the local grid.

For efficiency in the case where the boxes span less than half the globe, the 3D location module needs to be able to determine the greatest longitude difference between a base point at latitude ϕ_s and all points that are separated from that point by a central angle of θ. We might also want to know the latitude, ϕ_f, at which the largest separation occurs. Note also that an intermediate form below allows the computation of the maximum longitude difference at a particular latitude.

The central angle between a point at latitude ϕ_s and a second point at latitude ϕ_f that are separated in longitude by $\Delta\lambda$ is:

$$\theta = cos^{-1}(sin\phi_s sin\phi_f + cos\phi_s cos\phi_f cos\Delta\lambda)$$

Taking the cos of both sides gives:

$$cos\theta = (sin\phi_s sin\phi_f + cos\phi_s cos\phi_f cos\Delta\lambda)$$

Solving for $cos\Delta\lambda$ gives:

$$cos\Delta\lambda = \frac{a - bsin\phi_f}{ccos\phi_f}$$
$$cos\Delta\lambda = \frac{a}{csec\phi_f} - \frac{b}{ctan\phi_f}$$

where $a = cos\theta$, $b = sin\phi_s$, and $c = cos\phi_s$. We want to maximize $\Delta\lambda$ which implies minimizing $cos\Delta\lambda$ subject to constraints.

Taking the derivative with respect to ϕ_f gives:

$$\frac{dcos\Delta\lambda}{d\phi_f} = \frac{a}{csec\phi_f tan\phi_f} - \frac{b}{csec^2\phi_f} = 0$$

Factoring out $sec\phi_f$ which can never be 0 and using the definitions of sec and tan gives:

$$\frac{asin\phi_f}{ccos\phi_f} - \frac{b}{ccos\phi_f} = 0$$

Solving in the constrained range from 0 to $\pi/2$ gives:

$$sin\phi_f = \frac{b}{a} = \frac{sin\phi_s}{cos\theta}$$

So knowing base point (ϕ_s, λ_s), latitude ϕ_f, and distance θ we can use the great circle equation to find the longitude difference at the greatest separation point:

$$\Delta\lambda = cos^{-1}\left(\frac{a - bsin\phi_f}{ccos\phi_f}\right)$$

Note that if the angle between the base point and a pole is less than or equal to the central angle, all longitude differences will occur as the pole is approached.

6.88.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&location_nml
   use_octree      = .false.
   nboxes          = 1000
   maxdepth        = 4
   filled          = 10
   output_box_info = .false.
   print_box_level = 0
   compare_to_correct = .false.
/
```

Items in this namelist either control the way in which distances are computed and/or influence the code performance.

Item	Type	Description
use_octree	logical	There are two variations of search code. For now, this must be set to .false.
nboxes	integer	An optimization parameter which controls how many boxes the space is divided up into for precomputing nearby points. Larger numbers use more memory but may make searching faster if the model contains a large grid.
filled	integer	An optimization parameter for the octree code. Set the lower item count limit where a box no longer splits.
output_box_info	logical	If true, print more details about the distribution of locations across the array of boxes.
print_box_level	integer	If output_box_info is true, controls the amount of output.
compare_to_correct	logical	If true do an exhaustive (and slow) search to ensure the results are the same as using optimized search code. Should only be used for debugging.

6.88.3 Other modules used

```
types_mod
utilities_mod
random_seq_mod
```

6.88.4 Public interfaces

use location_mod, only :	location_type
	get_close_type
	get_location
	set_location
	write_location
	read_location
	interactive_location
	set_location_missing
	query_location
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	get_close_obs_destroy
	get_dist
	LocationDims
	LocationName
	LocationLName
	horiz_dist_only
	vert_is_undef
	vert_is_surface
	vert_is_pressure
	vert_is_scale_height
	vert_is_level
	vert_is_height
	VERTISUNDEF
	VERTISSURFACE
	VERTISLEVEL
	VERTISPRESSURE
	VERTISHEIGHT
	VERTISSCALEHEIGHT
	operator(==)
	operator(/=)

Namelist interface `&location_nml` must be read from file `input.nml`.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

type location_type

```
   private
   real(r8) :: lon, lat, vloc
   integer  :: which_vert
end type location_type
```

Provides an abstract representation of physical location on a three-d spherical shell.

Component	Description
lon	longitude in radians
lat	latitude in radians
vloc	vertical location, units as selected by which_vert
which_vert	type of vertical location: -2=no specific vert location; -1=surface; 1=level; 2=pressure; 3=height, 4=scale height

The vertical types have parameters defined for them so they can be referenced by name instead of number.

type get_close_type

```
   private
   integer  :: num
   real(r8) :: maxdist
   integer, pointer :: lon_offset(:, :)
   integer, pointer :: obs_box(:)
   integer, pointer :: count(:, :)
   integer, pointer :: start(:, :)
end type get_close_type
```

Provides a structure for doing efficient computation of close locations.

Component	Description
num	Number of locations in list
maxdist	Threshhold distance. Anything closer is close.
lon_offset	Dimensioned nlon by nlat. For a given offset in longitude boxes and difference in latitudes, gives max distance from base box to a point in offset box.
obs_box	Dimensioned num. Gives index of what box each location is in.
count	Dimensioned nlon by nlat. Number of obs in each box.
start	Dimensioned nlon by nlat. Index in straight storage list where obs in each box start.

var = get_location(loc)

```
real(r8), dimension(3)           :: get_location
type(location_type), intent(in)  :: loc
```

Extracts the longitude and latitude (converted to degrees) and the vertical location from a location type and returns in a 3 element real array.

`get_location`	The longitude and latitude (in degrees) and vertical location
`loc`	A location type

var = set_location(lon, lat, vert_loc, which_vert)

```
type(location_type)   :: set_location
real(r8), intent(in)    :: lon
real(r8), intent(in)    :: lat
real(r8), intent(in)    :: vert_loc
integer,  intent(in)    :: which_vert
```

Returns a location type with the input longitude and latitude (input in degrees) and the vertical location of type specified by which_vert.

`set_location`	A location type
`lon`	Longitude in degrees
`lat`	Latitude in degrees
`vert_loc`	Vertical location consistent with which_vert
`which_vert`	The vertical location type

call write_location(locfile, loc [, fform, charstring])

```
integer,                intent(in)       ::  locfile
type(location_type),    intent(in)       ::  loc
character(len=*), optional, intent(in)   ::  fform
character(len=*), optional, intent(out)  ::  charstring
```

Given an integer IO channel of an open file and a location, writes the location to this file. The *fform* argument controls whether write is "FORMATTED" or "UNFORMATTED" with default being formatted. If the final *charstring* argument is specified, the formatted location information is written to the character string only, and the `locfile` argument is ignored.

`locfile`	the unit number of an open file.
`loc`	location type to be written.
fform	Format specifier ("FORMATTED" or "UNFORMATTED"). Default is "FORMATTED" if not specified.
charstring	Character buffer where formatted location string is written if present, and no output is written to the file unit.

var = read_location(locfile [, fform])

```
type(location_type)                       :: read_location
integer, intent(in)                       :: locfile
character(len=*), optional, intent(in) :: fform
```

Reads a location_type from a file open on channel locfile using format *fform* (default is formatted).

`read_location`	Returned location type read from file
`locfile`	Integer channel opened to a file to be read
fform	Optional format specifier ("FORMATTED" or "UNFORMATTED"). Default "FORMATTED".

call interactive_location(location [, set_to_default])

```
type(location_type), intent(out) :: location
logical, optional, intent(in)    :: set_to_default
```

Use standard input to define a location type. With set_to_default true get one with all elements set to 0.

`location`	Location created from standard input
set_to_default	If true, sets all elements of location type to 0

var = query_location(loc [, attr])

```
real(r8)                                  :: query_location
type(location_type), intent(in)           :: loc
character(len=*), optional, intent(in) :: attr
```

Returns the value of which_vert, latitude, longitude, or vertical location from a location type as selected by the string argument attr. If attr is not present or if it is 'WHICH_VERT', the value of which_vert is converted to real and returned. Otherwise, attr='LON' returns longitude, attr='LAT' returns latitude and attr='VLOC' returns the vertical location.

`query_location`	Returns longitude, latitude, vertical location, or which_vert (converted to real)
`loc`	A location type
attr	Selects 'WHICH_VERT', 'LON', 'LAT' or 'VLOC'

var = set_location_missing()

```
type(location_type) :: set_location_missing
```

Returns a location with all elements set to missing values defined in types module.

`set_location_missing`	A location with all elements set to missing values

call get_close_maxdist_init(gc,maxdist, [maxdist_list])

```
type(get_close_type), intent(inout) :: gc
real(r8), intent(in)                :: maxdist
real(r8), intent(in), optional      :: maxdist_list(:)
```

Sets the threshhold distance. `maxdist` is in units of radians. Anything closer than this is deemed to be close. This routine must be called first, before the other `get_close` routines. It allocates space so it is necessary to call `get_close_obs_destroy` when completely done with getting distances between locations.

If the last optional argument is not specified, maxdist applies to all locations. If the last argument is specified, it must be a list of exactly the length of the number of specific types in the obs_kind_mod.f90 file. This length can be queried with the get_num_types_of_obs() function to get count of obs types. It allows a different maximum distance to be set per base type when get_close() is called.

`gc`	Data for efficiently finding close locations.
`maxdist`	Anything closer than this number of radians is a close location.
maxdist_list	If specified, must be a list of real values. The length of the list must be exactly the same length as the number of observation types defined in the obs_def_kind.f90 file. (See get_num_types_of_obs() to get count of obs types.) The values in this list are used for the obs types as the close distance instead of the maxdist argument.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type),             intent(inout) :: gc
integer,                          intent(in)    :: num
type(location_type), dimension(:) intent(in)    :: obs
```

Initialize storage for efficient identification of locations close to a given location. Allocates storage for keeping track of which 'box' each location in the list is in. Must be called after `get_close_maxdist_init`, and the list of locations here must be the same as the list of locations passed into `get_close_obs()`. If the list changes, `get_close_obs_destroy()` must be called, and both the initialization routines must be called again. It allocates space so it is necessary to call `get_close_obs_destroy` when completely done with getting distances between locations.

`gc`	Structure that contains data to efficiently find locations close to a given location.
`num`	The number of locations in the list.
`obs`	The locations of each element in the list, not used in 1D implementation.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, num_close, close_ind, dist)

```
type(get_close_type),                 intent(in)  :: gc
type(location_type),                  intent(in)  :: base_obs_loc
integer,                              intent(in)  :: base_obs_kind
type(location_type), dimension(:),    intent(in)  :: obs
integer,             dimension(:),    intent(in)  :: obs_kind
integer,                              intent(out) :: num_close
integer,             dimension(:),    intent(out) :: close_ind
real(r8), optional,  dimension(:),    intent(out) :: dist
```

Given a single location and a list of other locations, returns the indices of all the locations close to the single one along with the number of these and the distances for the close ones. The list of locations passed in via the obs argument must be identical to the list of obs passed into the most recent call to get_close_obs_init(). If the list of locations of interest changes get_close_obs_destroy() must be called and then the two initialization routines must be called before using get_close_obs() again.

If called without the optional *dist* argument, all locations that are potentially close are returned, which is likely a superset of the locations that are within the threshold distance specified in the get_close_maxdist_init() call. This can be useful to collect a list of potential locations, and then to convert all the vertical coordinates into one consistent unit (pressure, height in meters, etc), and then the list can be looped over, calling get_dist() directly to get the exact distance, either including vertical or not depending on the setting of horiz_dist_only.

gc	Structure to allow efficient identification of locations close to a given location.
base_obs_loc	Single given location.
base_obs_kind	Kind of the single location.
obs	List of locations from which close ones are to be found.
obs_kind	Kind associated with locations in obs list.
num_close	Number of locations close to the given location.
close_ind	Indices of those locations that are close.
dist	Distance between given location and the close ones identified in close_ind.

call get_close_obs_destroy(gc)

```
type(get_close_type), intent(inout) :: gc
```

Releases memory associated with the gc derived type. Must be called whenever the list of locations changes, and then get_close_maxdist_init and get_close_obs_init must be called again with the new locations list.

gc	Data for efficiently finding close locations.

var = get_dist(loc1, loc2, [, kind1, kind2, no_vert])

```
real(r8)                            :: get_dist
type(location_type), intent(in)  :: loc1
type(location_type), intent(in)  :: loc2
```

(continues on next page)

(continued from previous page)

```
integer, optional,   intent(in) :: kind1
integer, optional,   intent(in) :: kind2
logical, optional,   intent(in) :: no_vert
```

Returns the distance between two locations in radians. If `horiz_dist_only` is set to .TRUE. in the locations namelist, it computes great circle distance on sphere. If `horiz_dist_only` is false, then it computes an ellipsoidal distance with the horizontal component as above and the vertical distance determined by the types of the locations and the normalization constants set by the namelist for the different vertical coordinate types. The vertical normalization gives the vertical distance that is equally weighted as a horizontal distance of 1 radian. If *no_vert* is present, it overrides the value in the namelist and controls whether vertical distance is included or not.

The kind arguments are not used by the default location code, but are available to any user-supplied distance routines which want to do specialized calculations based on the kinds associated with each of the two locations.

`loc1`	First of two locations to compute distance between.
`loc2`	Second of two locations to compute distance between.
kind1	DART kind associated with location 1.
kind2	DART kind associated with location 2.
no_vert	If true, no vertical component to distance. If false, vertical component is included.
`var`	distance between loc1 and loc2.

var = vert_is_undef(loc)

```
logical                          :: vert_is_undef
type(location_type), intent(in)  :: loc
```

Returns true if which_vert is set to undefined, else false. The meaning of 'undefined' is specific; it means there is no particular vertical location associated with this type of measurement; for example a column-integrated value.

`vert_is_undef`	Returns true if vertical coordinate is set to undefined.
`loc`	A location type

var = vert_is_surface(loc)

```
logical                          :: vert_is_surface
type(location_type), intent(in)  :: loc
```

Returns true if which_vert is for surface, else false.

`vert_is_surface`	Returns true if vertical coordinate type is surface
`loc`	A location type

var = vert_is_pressure(loc)

```
logical                             :: vert_is_pressure
type(location_type), intent(in)  :: loc
```

Returns true if which_vert is for pressure, else false.

`vert_is_pressure`	Returns true if vertical coordinate type is pressure
`loc`	A location type

var = vert_is_scale_height(loc)

```
logical                             :: vert_is_scale_height
type(location_type), intent(in)  :: loc
```

Returns true if which_vert is for scale_height, else false.

`vert_is_scale_height`	Returns true if vertical coordinate type is scale_height
`loc`	A location type

var = vert_is_level(loc)

```
logical                             :: vert_is_level
type(location_type), intent(in)  :: loc
```

Returns true if which_vert is for level, else false.

`vert_is_level`	Returns true if vertical coordinate type is level
`loc`	A location type

var = vert_is_height(loc)

```
logical                             :: vert_is_height
type(location_type), intent(in)  :: loc
```

Returns true if which_vert is for height, else false.

`vert_is_height`	Returns true if vertical coordinate type is height
`loc`	A location type

var = has_vertical_localization()

```
logical :: has_vertical_localization
```

Returns .TRUE. if the namelist variable `horiz_dist_only` is .FALSE. meaning that vertical separation between locations is going to be computed by `get_dist()` and by `get_close_obs()`.

This routine should perhaps be renamed to something like 'using_vertical_for_distance' or something similar. The current use for it is in the localization code inside filter, but that doesn't make this a representative function name. And at least in current usage, returning the opposite setting of the namelist item makes the code read more direct (fewer double negatives).

loc1 == loc2

```
type(location_type), intent(in) :: loc1, loc2
```

Returns true if the two location types have identical values, else false.

loc1 /= loc2

```
type(location_type), intent(in) :: loc1, loc2
```

Returns true if the two location types do NOT have identical values, else false.

```
integer, parameter :: VERTISUNDEF       = -2
integer, parameter :: VERTISSURFACE     = -1
integer, parameter :: VERTISLEVEL       =  1
integer, parameter :: VERTISPRESSURE    =  2
integer, parameter :: VERTISHEIGHT      =  3
integer, parameter :: VERTISSCALEHEIGHT =  4
```

Constant parameters used to differentiate vertical types.

```
integer, parameter :: LocationDims = 3
```

This is a **constant**. Contains the number of real values in a location type. Useful for output routines that must deal transparently with many different location modules.

```
character(len=129), parameter :: LocationName = "loc3Dsphere"
```

This is a **constant**. A parameter to identify this location module in output metadata.

```
character(len=129), parameter :: LocationLName =

       "threed sphere locations: lon, lat, vertical"
```

This is a **constant**. A parameter set to "threed sphere locations: lon, lat, vertical" used to identify this location module in output long name metadata.

6.88.5 Files

filename	purpose
input.nml	to read the location_mod namelist

6.88.6 References

1. none

6.88.7 Private components

N/A

6.89 MODULE location_mod

6.89.1 Overview

DART provides a selection of options for the coordinate system in which all observations and all model state vector locations are described. All executables are built with a single choice from the available location modules. The names of these modules are all `location_mod`.

6.89.2 Introduction

The core algorithms of DART work with many different models which have a variety of coordinate systems. This directory provides code for creating, setting/getting, copying location information (coordinates) independently of the actual specific coordinate information. It also contains distance routines needed by the DART algorithms.

Each of the different location_mod.f90 files provides the same set of interfaces and defines a 'module location_mod', so by selecting the proper version in your path_names_xxx file you can compile your model code with the main DART routines.

- *MODULE location_mod (threed_sphere)*: The most frequently used version for real-world 3d models. It uses latitude and longitude for horizontal coordinates, plus a vertical coordinate which can be meters, pressure, model level, surface, or no specific vertical location.
- *MODULE (1D) location_mod*: The most frequently used for small models (e.g. the Lorenz family). It has a cyclic domain from 0 to 1.
- others:
  - *MODULE location_mod (threed_cartesian)*: A full 3D X,Y,Z coordinate system.
  - column: no x,y but 1d height, pressure, or model level for vertical.
  - annulus: a hollow 3d cylinder with azimuth, radius, and depth.
  - twod: a periodic 2d domain with x,y coordinates between 0 and 1.
  - twod_sphere: a 2d shell with latitude, longitude pairs.
  - threed: a periodic 3d domain with x,y,z coordinates between 0 and 1.
  - *MODULE location_mod (channel)*: a 3d domain periodic in x, limited in y, and unlimited z.

Other schemes can be added, as needed by the models. Possible ideas are a non-periodic version of the 1d, 2d cartesian versions. Email dart at ucar.edu if you have a different coordinate scheme which we might want to support.

6.89.3 Namelist

Each location module option has a different namelist. See the specific documentation for the location option of choice.

6.89.4 Files

- none

6.89.5 References

- none

6.89.6 Private components

N/A

6.90 MODULE (1D) location_mod

6.90.1 Overview

The DART framework needs to be able to compute distances between locations, to pass location information to and from the model interface code (in model_mod.f90), and to be able to read and write location information to files. DART isolates all this location information into separate modules so that the main algorithms can operate with the same code independent of whether the model uses latitude/longitude/height, one-d unit sphere coordinates, cylindrical coordinates, etc. DART provides about half a dozen possible coordinate systems, and others can be added.

This locations module provides a representation of a physical location on a periodic 1D domain with location values between 0 and 1. A type that abstracts the location is provided along with operators to set, get, read, write, and compute distances between locations. This is a member of a class of similar location modules that provide the same abstraction for different represenations of physical space.

All possible location modules define the same module name `location_mod`. Therefore, the DART framework and any user code should include a Fortran 90 'use' statement of 'location_mod'. The selection of exactly which location module is compiled is specified by the source file name in the `path_names_xxx` file, which is read by the `mkmf_xxx` scripts.

The model-specific `model_mod.f90` files need to define four `get_close` routines, but in most cases they can simply put a `use` statement at the top which uses the routines in the locations module, and they do not have to provide any additional code.

However, if the model interface code wants to intercept and alter the default behavior of the get_close routines, they are able to. The correct usage of the `get_close` routines is as follows:

```
call get_close_maxdist_init()  ! must be called before get_close_obs_init()
call get_close_obs_init()
...
call get_close_obs()           ! many, many times
...
call get_close_obs_destroy()
```

Regardless of the fact that the names include the string 'obs', they are intended for use with any group of locations in the system, frequently state vector items or observations, but any location is acceptable.

6.90.2 Namelist

This version of the locations module does not have any namelist input.

6.90.3 Other modules used

```
types_mod
utilities_mod
random_seq_mod
```

6.90.4 Public interfaces

use location_mod, only :	location_type
	get_close_type
	get_location
	set_location
	write_location
	read_location
	interactive_location
	set_location_missing
	query_location
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	get_close_obs_destroy
	get_dist
	LocationDims
	LocationName
	LocationLName
	horiz_dist_only
	vert_is_undef
	vert_is_surface
	vert_is_pressure
	vert_is_level
	vert_is_height
	VERTISUNDEF
	VERTISSURFACE
	VERTISLEVEL
	VERTISPRESSURE
	VERTISHEIGHT
	operator(==)

There is currently no namelist interface for the 1D location module.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

type location_type

```
   private
   real(r8) :: x
end type location_type
```

Provides an abstract representation of physical location on a one-dimensional periodic domain.

Component	Description
x	Location has range 0 to 1

type get_close_type

```
   private
   integer  :: num
   real(r8) :: maxdist
end type get_close_type
```

Provides a structure for doing efficient computation of close locations. Doesn't do anything in the 1D implementation except provide appropriate stubs.

Component	Description
num	Number of locations in list
maxdist	Threshhold distance. Anything closer is close.

var = get_location(loc)

```
real(r8)                            :: get_location
type(location_type), intent(in)  :: loc
```

Extracts the real location value, range 0 to 1, from a location type.

`get_location`	The real value for a location
`loc`	A location derived type

var = set_location(x)

```
type(location_type)   :: set_location
real(r8), intent(in)  :: x
```

Returns a location type with the location x.

`set_location`	A location derived type
`x`	Location value in the range 0. to 1.

call write_location(locfile, loc [, fform, charstring])

```
integer,                 intent(in)       ::  locfile
type(location_type),     intent(in)       ::  loc
character(len=*), optional, intent(in)   ::  fform
character(len=*), optional, intent(out)  ::  charstring
```

Given an integer IO channel of an open file and a location, writes the location to this file. The *fform* argument controls whether write is "FORMATTED" or "UNFORMATTED" with default being formatted. If the final *charstring* argument is specified, the formatted location information is written to the character string only, and the `locfile` argument is ignored.

`locfile`	the unit number of an open file.
`loc`	location type to be written.
fform	Format specifier ("FORMATTED" or "UNFORMATTED"). Default is "FORMATTED" if not specified.
charstring	Character buffer where formatted location string is written if present, and no output is written to the file unit.

var = read_location(locfile [, fform])

```
type(location_type)                        :: read_location
integer, intent(in)                        :: locfile
character(len=*), optional, intent(in)   :: fform
```

Reads a location_type from a file open on channel locfile using format *fform* (default is formatted).

`read_location`	Returned location type read from file
`locfile`	Integer channel opened to a file to be read
fform	Optional format specifier ("FORMATTED" or "UNFORMATTED"). Default "FORMATTED".

call interactive_location(location [, set_to_default])

```
type(location_type), intent(out) :: location
logical, optional, intent(in)    :: set_to_default
```

Use standard input to define a location type. With set_to_default true get one with all elements set to 0.

`location`	Location created from standard input
set_to_default	If true, sets all elements of location type to 0

var = query_location(loc [, attr])

```
real(r8)                                  :: query_location
type(location_type), intent(in)           :: loc
character(len=*), optional, intent(in)    :: attr
```

Returns the location value if attr = ‘X’ or if attr is not passed.

`query_location`	Returns value of x.
`loc`	A location type
attr	Selects ‘X’

var = set_location_missing()

```
type(location_type) :: set_location_missing
```

Returns a location with location set to missing value from types_mod.

`set_location_missing`	A location set to missing value

call get_close_maxdist_init(gc,maxdist , [maxdist_list])

```
type(get_close_type), intent(inout) :: gc
real(r8), intent(in)                :: maxdist
real(r8), intent(in), optional      :: maxdist_list(:)
```

Sets the threshhold distance. Anything closer than this is deemed to be close. This routine must be called first, before the other `get_close` routines. It allocates space so it is necessary to call `get_close_obs_destroy` when completely done with getting distances between locations.

gc	Data for efficiently finding close locations.
maxdist	Anything closer than this distance is a close location.
maxdist_list	Ignored for this location type.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type),              intent(inout) :: gc
integer,                           intent(in)    :: num
type(location_type), dimension(:) intent(in)    :: obs
```

Initialize storage for efficient identification of locations close to a given location. The oned implementation is minimal and just records the number of locations here. Must be called after `get_close_maxdist_init`, and the list of locations here must be the same as the list of locations passed into `get_close_obs()`. If the list changes, `get_close_obs_destroy()` must be called, and both the initialization routines must be called again. It allocates space so it is necessary to call `get_close_obs_destroy` when completely done with getting distances between locations.

gc	Structure that contains data to efficiently find locations close to a given location.
num	The number of locations in the list.
obs	The locations of each element in the list, not used in 1D implementation.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, num_close, close_ind, dist)

```
type(get_close_type),              intent(in)  :: gc
type(location_type),               intent(in)  :: base_obs_loc
integer,                           intent(in)  :: base_obs_kind
type(location_type), dimension(:), intent(in)  :: obs
integer, dimension(:),             intent(in)  :: obs_kind
integer,                           intent(out) :: num_close
integer, dimension(:),             intent(out) :: close_ind
real(r8), dimension(:),            intent(out) :: dist
```

Given a single location and a list of other locations, returns the indices of all the locations close to the single one along with the number of these and the distances for the close ones. The list of locations passed in via the `obs` argument must be identical to the list of `obs` passed into the most recent call to `get_close_obs_init()`. If the list of locations of interest changes `get_close_obs_destroy()` must be called and then the two initialization routines must be called before using `get_close_obs()` again.

gc	Structure to allow efficient identification of locations close to a given location.
base_obs_loc	Single given location.
base_obs_kind	Kind of the single location.
obs	List of locations from which close ones are to be found.
obs_kind	Kind associated with locations in obs list.
num_close	Number of locations close to the given location.
close_ind	Indices of those locations that are close.
dist	Distance between given location and the close ones identified in close_ind.

call get_close_obs_destroy(gc)

```
type(get_close_type), intent(inout) :: gc
```

Releases memory associated with the `gc` derived type. Must be called whenever the list of locations changes, and then `get_close_maxdist_init` and `get_close_obs_init` must be called again with the new locations list.

gc	Data for efficiently finding close locations.

var = get_dist(loc1, loc2, [, kind1, kind2])

```
real(r8)                             :: get_dist
type(location_type), intent(in)  :: loc1
type(location_type), intent(in)  :: loc2
integer, optional,   intent(in)  :: kind1
integer, optional,   intent(in)  :: kind2
```

Return the distance between 2 locations. Since this is a periodic domain, the shortest distance may wrap around.

The kind arguments are not used by the default location code, but are available to any user-supplied distance routines which want to do specialized calculations based on the kinds associated with each of the two locations.

loc1	First of two locations to compute distance between.
loc2	Second of two locations to compute distance between.
kind1	DART kind associated with location 1.
kind2	DART kind associated with location 2.
var	distance between loc1 and loc2.

var = vert_is_undef(loc)

```
logical                              :: vert_is_undef
type(location_type), intent(in)  :: loc
```

Always returns false; this locations module has no vertical coordinates. Provided only for compile-time compatibility with other location modules.

`vert_is_undef`	Always returns .FALSE.
`loc`	A location type

var = vert_is_surface(loc)

```
logical                          :: vert_is_surface
type(location_type), intent(in)  :: loc
```

Always returns false; this locations module has no vertical coordinates. Provided only for compile-time compatibility with other location modules.

`vert_is_surface`	Always returns .FALSE.
`loc`	A location type

var = vert_is_pressure(loc)

```
logical                          :: vert_is_pressure
type(location_type), intent(in)  :: loc
```

Always returns false; this locations module has no vertical coordinates. Provided only for compile-time compatibility with other location modules.

`vert_is_pressure`	Always returns .FALSE.
`loc`	A location type

var = vert_is_level(loc)

```
logical                          :: vert_is_level
type(location_type), intent(in)  :: loc
```

Always returns false; this locations module has no vertical coordinates. Provided only for compile-time compatibility with other location modules.

`vert_is_level`	Always returns .FALSE.
`loc`	A location type

var = vert_is_height(loc)

```
logical                          :: vert_is_height
type(location_type), intent(in)  :: loc
```

Always returns false; this locations module has no vertical coordinates. Provided only for compile-time compatibility with other location modules.

`vert_is_height`	Always returns .FALSE.
`loc`	A location type

var = has_vertical_localization()

```
logical :: has_vertical_localization
```

Always returns false; this locations module has no vertical coordinates. Provided only for compile-time compatibility with other location modules.

See note in threed_sphere locations module about the function name.

loc1 == loc2

```
type(location_type), intent(in) :: loc1, loc2
```

Returns true if the two location types have identical values, else false.

loc1 /= loc2

```
type(location_type), intent(in) :: loc1, loc2
```

Returns true if the two location types do NOT have identical values, else false.

```
integer, parameter :: VERTISUNDEF    = -2
integer, parameter :: VERTISSURFACE  = -1
integer, parameter :: VERTISLEVEL    =  1
integer, parameter :: VERTISPRESSURE =  2
integer, parameter :: VERTISHEIGHT   =  3
```

This locations module has no vertical coordinate, but for compatibility with other location modules, these are defined.

```
integer, parameter :: LocationDims = 1
```

This is a **constant**. Contains the number of real values in a location type. Useful for output routines that must deal transparently with many different location modules.

```
character(len=129), parameter :: LocationName = "loc1d"
```

This is a **constant**. A parameter to identify this location module in output metadata.

```
character(len=129), parameter :: LocationLName = "location on unit circle"
```

This is a **constant**. A parameter to identify this location module in output long name metadata.

6.90.5 Files

None.

6.90.6 References

1. none

6.90.7 Private components

N/A

6.91 MODULE location_mod (threed_cartesian)

6.91.1 Overview

The DART framework needs to be able to compute distances between locations, to pass location information to and from the model interface code (model_mod.f90), and to be able to read and write location information to files. DART isolates all this location information into separate modules so that the main algorithms can operate with the same code independent of whether the model uses latitude/longitude/height, 1D unit cartesian coordinates, cylindrical coordinates, etc. DART provides about half a dozen possible coordinate systems, and others can be added. The most common one for geophysical models is the *MODULE location_mod (threed_sphere)* version. This document describes an alternative 3D cartesian coordinate system.

Note that only one location module can be compiled into any single DART executable, and most earth observational data is generated in [latitude, longitude, vertical pressure or height] coordinates - the threed_sphere option. The cartesian and 3D sphere locations cannot be mixed or used together.

This location module provides a representation of a physical location in an [X, Y, Z] cartesian coordinate space. A type that abstracts the location is provided along with operators to set, get, read, write, and compute distances between locations. This is a member of a class of similar location modules that provide the same abstraction for different represenations of physical space.

Location-independent code

All types of location modules define the same module name `location_mod`. Therefore, the DART framework and any user code should include a Fortran 90 `use` statement of `location_mod`. The selection of which location module will be compiled into the program is controlled by which source file name is specified in the `path_names_xxx` file, which is used by the `mkmf_xxx` scripts.

All types of location modules define the same Fortran 90 derived type `location_type`. Programs that need to pass location information to subroutines but do not need to interpret the contents can declare, receive, and pass this derived type around in their code independent of which location module is specified at compile time. Model and location-independent utilities should be written in this way. However, as soon as the contents of the location type needs to be accessed by user code then it becomes dependent on the exact type of location module that it is compiled with.

Usage of distance routines

Regardless of the fact that the distance subroutine names include the string 'obs', there is nothing specific to observations in these routines. They work to compute distances between any set of locations. The most frequent use of these routines in the filter code is to compute the distance between a single observation and items in the state vector, and also between a single observation and other nearby observations. However, any source for locations is supported.

In simpler location modules (like the `oned` version) there is no need for anything other than a brute force search between the base location and all available state vector locations. However in the case of large geophysical models which typically use the `threed_cartesian` locations code, the brute-force search time is prohibitive. The location code pre-processes all locations into a set of *bins* and then only needs to search the lists of locations in nearby bins when looking for locations that are within a specified distance.

The expected calling sequence of the `get_close` routines is as follows:

```
call get_close_maxdist_init()  ! is called before get_close_obs_init()
call get_close_obs_init()

call get_close_obs()           ! called many, many times

call get_close_obs_destroy()
```

In the `threed_cartesian` implementation the first routine initializes some data structures, the second one bins up the list of locations, and then the third one is called multiple times to find all locations within a given radius of some reference location, and to optionally compute the exact separation distance from the reference location. The last routine deallocates the space. See the documentation below for the specific details for each routine.

All 4 of these routines must be present in every location module but in most other versions all but `get_close_obs()` are stubs. In this `threed_cartesian` version of the locations module all are fully implemented.

Interaction with model_mod.f90 code

The filter and other DART programs could call the `get_close` routines directly, but typically do not. They declare them (in a `use` statement) to be in the `model_mod` module, and all model interface modules are required to supply them. However in many cases the model_mod only needs to contain another `use` statement declaring them to come from the `location_mod` module. Thus they 'pass through' the model_mod but the user does not need to provide a subroutine or any code for them.

However, if the model interface code wants to intercept and alter the default behavior of the get_close routines, it is able to. Typically the model_mod still calls the location_mod routines and then adjusts the results before passing them back to the calling code. To do that, the model_mod must be able to call the routines in the location_mod which have the same names as the subroutines it is providing. To allow the compiler to distinguish which routine is to be called where, we use the Fortran 90 feature which allows a module routine to be renamed in the use statement. For example, a common case is for the model_mod to want to supply additions to the get_close_obs() routine only. At the top of the model_mod code it would declare:

```
use location_mod, only :: location_get_close_obs => get_close_obs,    &
                          get_close_maxdist_init, get_close_obs_init, &
                          get_close_obs_destroy
```

That makes calls to the maxdist_init, init, and destroy routines simply pass through to the code in the location_mod, but the model_mod must supply a get_close_obs() subroutine. When it wants to call the code in the location_mod it calls `location_get_close_obs()`.

One use pattern is for the model_mod to call the location get_close_obs() routine without the `dist` argument. This returns a list of any potentially close locations without computing the exact distance from the base location. At this point the list of locations is a copy and the model_mod routine is free to alter the list in any way it chooses: for example, it can change the locations to make certain types of locations appear closer or further away from the base location. Then typically the model_mod code loops over the list calling the `get_dist()` routine to get the actual distances to be returned to the calling code.

Horizontal distance only

This option is not supported for the threed_cartesian option.

Precomputation for run-time search efficiency

For search efficiency all locations are pre-binned. For the non-octree option, the total list of locations is divided up into *nx* by *ny* by *nz* boxes and the index numbers of all items (both state vector entries and observations) are stored in the appropriate box. To locate all points close to a given location, only the locations listed in the boxes within the search radius must be checked. This speeds up the computations, for example, when localization controls which state vector items are impacted by any given observation. The search radius is the localization distance and only those state vector items in boxes closer than the radius to the observation location are processed.

The default values have given good performance on many of our existing model runs, but for tuning purposes the box counts have been added to the namelist to allow adjustment. By default the code prints some summary information about how full the average box is, how many are empty, and how many items were in the box with the largest count. The namelist value *output_box_info* can be set to .true. to get even more information about the box statistics. The best performance will be obtained somewhere between two extremes; the worst extreme is all the points are located in just a few boxes. This degenerates into a (slow) linear search through the index list. The other extreme is a large number of empty or sparsely filled boxes. The overhead of creating, managing, and searching a long list of boxes will impact performance. The best performance lies somewhere in the middle, where each box contains a reasonable number of values, more or less evenly distributed across boxes. The absolute numbers for best performance will certainly vary from case to case.

6.91.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&location_nml
   nx                   = 10
   ny                   = 10
   nz                   = 10
   x_is_periodic        = .false.
   min_x_for_periodic   = -888888.0
   max_x_for_periodic   = -888888.0
   y_is_periodic        = .false.
   min_y_for_periodic   = -888888.0
   max_y_for_periodic   = -888888.0
   z_is_periodic        = .false.
   min_z_for_periodic   = -888888.0
   max_z_for_periodic   = -888888.0
   compare_to_correct   = .false.
   output_box_info      = .false.
   print_box_level      = 0
   debug                = 0
  /
```

Items in this namelist either control the way in which distances are computed and/or influence the code performance.

Item	Type	Description
nx, ny, nz	integer	The number of boxes in each dimension to use to speed the searches. This is **not** the number of gridcells.
x_is_periodic, y_is_periodic, z_is_periodic	logical	If .true., the domain wraps in the coordinate.
min_x_for_periodic, max_x_for_periodic	real(r8)	The minimum and maximum values that are considered to be identical locations if `x_is_periodic = .true.`
min_y_for_periodic, max_y_for_periodic	real(r8)	The minimum and maximum values that are considered to be identical locations if `y_is_periodic = .true.`
min_z_for_periodic, max_z_for_periodic	real(r8)	The minimum and maximum values that are considered to be identical locations if `z_is_periodic = .true.`
compare_to_correct	logical	If true, do an exhaustive search for the closest point. Only useful for debugging because the performance cost is prohibitive.
output_box_info	logical	Print out debugging info.
print_box_level	logical	If output_box_info is true, how detailed should the output be.
debug	integer	The higher the number, the more verbose the run-time output. 0 (zero) is the minimum run-time output.

6.91.3 Other modules used

```
types_mod
utilities_mod
random_seq_mod
```

6.91.4 Public interfaces

`use location_mod, only :`	location_type
	get_close_type
	get_location
	set_location
	write_location
	read_location
	interactive_location
	set_location_missing
	query_location
	get_close_maxdist_init
	get_close_obs_init
	get_close_obs
	get_close_obs_destroy
	get_dist
	LocationDims
	LocationName
	LocationLName
	horiz_dist_only
	vert_is_undef
	vert_is_surface
	vert_is_pressure
	vert_is_scale_height
	vert_is_level
	vert_is_height
	operator(==)
	operator(/=)

Namelist interface `&location_nml` must be read from file `input.nml`.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

type location_type

```
   private
   real(r8) :: x, y, z
end type location_type
```

Provides an abstract representation of physical location in a 3D cartesian space.

Component	Description
x, y, z	location in each dimension

type get_close_type

```
   private
   integer, pointer  :: loc_box(:)           ! (nloc); List of loc indices in boxes
   integer, pointer  :: count(:, :, :)       ! (nx, ny, nz); # of locs in each box
   integer, pointer  :: start(:, :, :)       ! (nx, ny, nz); Start of list of locs in␣
↪this box
   real(r8)          :: bot_x, top_x         ! extents in x, y, z
   real(r8)          :: bot_y, top_y
   real(r8)          :: bot_z, top_z
   real(r8)          :: x_width, y_width, z_width    ! widths of boxes in x,y,z
   real(r8)          :: nboxes_x, nboxes_y, nboxes_z ! based on maxdist how far to␣
↪search
end type get_close_type
```

Provides a structure for doing efficient computation of close locations.

var = get_location(loc)

```
real(r8), dimension(3)          :: get_location
type(location_type), intent(in) :: loc
```

Extracts the x, y, z locations from a location type and returns in a 3 element real array.

`get_location`	The x,y,z values
`loc`	A location type

var = set_location(x, y, z) var = set_location(lon, lat, height, radius)

```
type(location_type) :: set_location
real(r8), intent(in)     :: x
real(r8), intent(in)     :: y
real(r8), intent(in)     :: z
```

or

```
type(location_type) :: set_location
real(r8), intent(in)     :: lon
real(r8), intent(in)     :: lat
real(r8), intent(in)     :: height
real(r8), intent(in)     :: radius
```

Returns a location type with the input [x,y,z] or allows the input to be specified as locations on the surface of a sphere with a specified radius and height above the surface.

`set_location`	A location type
`x, y, z`	Coordinates along each axis
`lon, lat`	Longitude, Latitude in degrees
`height`	Vertical location in same units as radius (e.g. meters)
`radius`	The radius of the sphere in same units as height (e.g. meters)

call write_location(locfile, loc [, fform, charstring])

```
integer,                intent(in)       ::  locfile
type(location_type),    intent(in)       ::  loc
character(len=*), optional, intent(in)   ::  fform
character(len=*), optional, intent(out)  ::  charstring
```

Given an integer IO channel of an open file and a location, writes the location to this file. The *fform* argument controls whether write is “FORMATTED” or “UNFORMATTED” with default being formatted. If the final *charstring* argument is specified, the formatted location information is written to the character string only, and the `locfile` argument is ignored.

`locfile`	the unit number of an open file.
`loc`	location type to be written.
fform	Format specifier (“FORMATTED” or “UNFORMATTED”). Default is “FORMATTED” if not specified.
charstring	Character buffer where formatted location string is written if present, and no output is written to the file unit.

var = read_location(locfile [, fform])

```
type(location_type)                      :: read_location
integer, intent(in)                      :: locfile
character(len=*), optional, intent(in) :: fform
```

Reads a location_type from a file open on channel locfile using format *fform* (default is formatted).

`read_location`	Returned location type read from file
`locfile`	Integer channel opened to a file to be read
fform	Optional format specifier ("FORMATTED" or "UNFORMATTED"). Default "FORMATTED".

call interactive_location(location [, set_to_default])

```
type(location_type), intent(out) :: location
logical, optional, intent(in)    :: set_to_default
```

Use standard input to define a location type. With set_to_default true get one with all elements set to 0.

`location`	Location created from standard input
set_to_default	If true, sets all elements of location type to 0

var = query_location(loc [, attr])

```
real(r8)                                 :: query_location
type(location_type), intent(in)          :: loc
character(len=*), optional, intent(in) :: attr
```

Returns the value of x, y, z depending on the attribute specification. If attr is not present, returns 'x'.

`query_location`	Returns x, y, or z.
`loc`	A location type
attr	Selects 'X', 'Y', 'Z'. If not specified, 'X' is returned.

var = set_location_missing()

```
type(location_type) :: set_location_missing
```

Returns a location with all elements set to missing values defined in types module.

`set_location_missing`	A location with all elements set to missing values

call get_close_maxdist_init(gc,maxdist, [maxdist_list])

```
type(get_close_type), intent(inout) :: gc
real(r8), intent(in)                :: maxdist
real(r8), intent(in), optional      :: maxdist_list(:)
```

Sets the threshhold distance. `maxdist` is in units of radians. Anything closer than this is deemed to be close. This routine must be called first, before the other `get_close` routines. It allocates space so it is necessary to call `get_close_obs_destroy` when completely done with getting distances between locations.

If the last optional argument is not specified, maxdist applies to all locations. If the last argument is specified, it must be a list of exactly the length of the number of specific types in the obs_kind_mod.f90 file. This length can be queried with the get_num_types_of_obs() function to get count of obs types. It allows a different maximum distance to be set per base type when get_close() is called.

`gc`	Data for efficiently finding close locations.
`maxdist`	Anything closer than this number of radians is a close location.
maxdist	If specified, must be a list of real values. The length of the list must be exactly the same length as the number of observation types defined in the obs_def_kind.f90 file. (See get_num_types_of_obs() to get count of obs types.) The values in this list are used for the obs types as the close distance instead of the maxdist argument.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type),              intent(inout) :: gc
integer,                           intent(in)    :: num
type(location_type), dimension(:) intent(in)    :: obs
```

Initialize storage for efficient identification of locations close to a given location. Allocates storage for keeping track of which 'box' each location in the list is in. Must be called after `get_close_maxdist_init`, and the list of locations here must be the same as the list of locations passed into `get_close_obs()`. If the list changes, `get_close_obs_destroy()` must be called, and both the initialization routines must be called again. It allocates space so it is necessary to call `get_close_obs_destroy` when completely done with getting distances between locations.

`gc`	Structure that contains data to efficiently find locations close to a given location.
`num`	The number of locations in the list.
`obs`	The locations of each element in the list, not used in 1D implementation.

call get_close_obs(gc, base_obs_loc, base_obs_type, obs, obs_kind, num_close, close_ind, dist)

```
type(get_close_type),              intent(in)  :: gc
type(location_type),               intent(in)  :: base_obs_loc
integer,                           intent(in)  :: base_obs_type
```

(continues on next page)

(continued from previous page)

```
type(location_type), dimension(:), intent(in)  :: obs
integer,             dimension(:), intent(in)  :: obs_kind
integer,                           intent(out) :: num_close
integer,             dimension(:), intent(out) :: close_ind
real(r8), optional,  dimension(:), intent(out) :: dist
```

Given a single location and a list of other locations, returns the indices of all the locations close to the single one along with the number of these and the distances for the close ones. The list of locations passed in via the `obs` argument must be identical to the list of `obs` passed into the most recent call to `get_close_obs_init()`. If the list of locations of interest changes `get_close_obs_destroy()` must be called and then the two initialization routines must be called before using `get_close_obs()` again.

Note that the base location is passed with the specific type associated with that location. The list of potential close locations is matched with a list of generic kinds. This is because in the current usage in the DART system the base location is always associated with an actual observation, which has both a specific type and generic kind. The list of potentially close locations is used both for other observation locations but also for state variable locations which only have a generic kind.

If called without the optional *dist* argument, all locations that are potentially close are returned, which is likely a superset of the locations that are within the threshold distance specified in the `get_close_maxdist_init()` call.

`gc`	Structure to allow efficient identification of locations close to a given location.
`base_obs_loc`	Single given location.
`base_obs_type`	Specific type of the single location.
`obs`	List of locations from which close ones are to be found.
`obs_kind`	Generic kind associated with locations in obs list.
`num_close`	Number of locations close to the given location.
`close_ind`	Indices of those locations that are close.
dist	Distance between given location and the close ones identified in close_ind.

call get_close_obs_destroy(gc)

```
type(get_close_type), intent(inout) :: gc
```

Releases memory associated with the `gc` derived type. Must be called whenever the list of locations changes, and then `get_close_maxdist_init` and `get_close_obs_init` must be called again with the new locations list.

`gc`	Data for efficiently finding close locations.

var = get_dist(loc1, loc2, [, type1, kind2, no_vert])

```
real(r8)                         :: get_dist
type(location_type), intent(in)  :: loc1
type(location_type), intent(in)  :: loc2
```

(continues on next page)

(continued from previous page)

```
integer, optional,   intent(in) :: type1
integer, optional,   intent(in) :: kind2
```

Returns the distance between two locations.

The type and kind arguments are not used by the default location code, but are available to any user-supplied distance routines which want to do specialized calculations based on the types/kinds associated with each of the two locations.

`loc1`	First of two locations to compute distance between.
`loc2`	Second of two locations to compute distance between.
type1	DART specific type associated with location 1.
kind2	DART generic kind associated with location 2.
`var`	distance between loc1 and loc2.

var = vert_is_undef(loc)

```
logical                              :: vert_is_undef
type(location_type), intent(in)  :: loc
```

Always returns .false.

`vert_is_undef`	Always returns .false.
`loc`	A location type

var = vert_is_surface(loc)

```
logical                              :: vert_is_surface
type(location_type), intent(in)  :: loc
```

Always returns .false.

`vert_is_surface`	Always returns .false.
`loc`	A location type

var = vert_is_pressure(loc)

```
logical                              :: vert_is_pressure
type(location_type), intent(in)  :: loc
```

Always returns .false.

`vert_is_pressure`	Always returns .false.
`loc`	A location type

var = vert_is_scale_height(loc)

```
logical                            :: vert_is_scale_height
type(location_type), intent(in)  :: loc
```

Always returns .false.

`vert_is_scale_height`	Always returns .false.
`loc`	A location type

var = vert_is_level(loc)

```
logical                            :: vert_is_level
type(location_type), intent(in)  :: loc
```

Always returns .false.

`vert_is_level`	Always returns .false.
`loc`	A location type

var = vert_is_height(loc)

```
logical                            :: vert_is_height
type(location_type), intent(in)  :: loc
```

Always returns .false.

`vert_is_height`	Always returns .false.
`loc`	A location type

var = has_vertical_localization()

```
logical :: has_vertical_localization
```

Always returns .false.

This routine should perhaps be renamed to something like ‘using_vertical_for_distance’ or something similar. The current use for it is in the localization code inside filter, but that doesn’t make this a representative function name. And at least in current usage, returning the opposite setting of the namelist item makes the code read more direct (fewer double negatives).

loc1 == loc2

```
type(location_type), intent(in) :: loc1, loc2
```

Returns true if the two location types have identical values, else false.

loc1 /= loc2

```
type(location_type), intent(in) :: loc1, loc2
```

Returns true if the two location types do NOT have identical values, else false.

```
integer, parameter :: LocationDims = 3
```

This is a **constant**. Contains the number of real values in a location type. Useful for output routines that must deal transparently with many different location modules.

```
character(len=129), parameter :: LocationName = "loc3Dcartesian"
```

This is a **constant**. A parameter to identify this location module in output metadata.

```
character(len=129), parameter :: LocationLName =

       "threed cartesian locations: x, y, z"
```

This is a **constant**. A parameter set to “threed cartesian locations: x, y, z” used to identify this location module in output long name metadata.

6.91.5 Files

filename	purpose
input.nml	to read the location_mod namelist

6.91.6 References

1. none

6.91.7 Private components

N/A

6.92 MODULE location_mod (threed_sphere)

6.92.1 Overview

The DART framework needs to be able to compute distances between locations, to pass location information to and from the model interface code (`model_mod.f90`), and to be able to read and write location information to files. DART isolates all this location information into separate modules so that the main algorithms can operate with the same code independent of whether the model uses latitude/longitude/height, 1D unit sphere coordinates, cylindrical coordinates, etc. DART provides about half a dozen possible coordinate systems, and others can be added. The most common one for geophysical models is this one: threed_sphere.

This location module provides a representation of a physical location on a 3-D spherical shell, using latitude and longitude plus a vertical component with choices of vertical coordinate type such as pressure or height in meters. A type that abstracts the location is provided along with operators to set, get, read, write, and compute great circle distances between locations. This is a member of a class of similar location modules that provide the same abstraction for different represenations of physical space.

6.92.2 Usage

The location routines are general purpose code that can be used for a variety of utilities. The following discussion is specifically restricted to how the location namelist settings affect the execution of the `filter` assimilation program.

Issues related to changing the results of an assimilation based on the location module settings:

- Whether and how to treat the vertical separation when computing distances between two locations.
- Whether to use different distances in the vertical for different observation types.

Issues related to changing the results of an assimilation based on code in the model-specific `model_mod.f90` module:

- Whether the model-specific code needs to convert vertical coordinates.
- Whether the model-specific code alters the distances in some other way.

Issues related to the speed/efficiency of an assimilation based on the location module settings:

- Whether to use a faster but less precise distance computation.
- Whether to change the number of internal bins used to more quickly find nearby locations.

Vertical issues

The localization distance during an assimilation – the maximum separation between an observation and a state vector item potentially affected by the assimilation – is set in the &assim_tools_nml namelist (the `cutoff` item).

Setting `horiz_dist_only = .TRUE.` in the namelist means the great circle distances will be computed using only the latitude and longitudes of the two locations, ignoring the vertical components of the locations. The cutoff is specified in radians to be independent of the radius of the sphere. For the Earth the radius is nominally 6,371 Km. To compute the horizontal only localization distance, multiply 6,371 Km by the cutoff to get the distance in Km. The cutoff is by definition 1/2 the distance to where the increments go to 0, so multiply that result by 2 to get the maximum distance at which an observation can alter the state.

Setting `horiz_dist_only = .FALSE.` in the namelist means the code will compute a 3D distance, including the vertical separation. In this case, the `vert_normalization_xxx` namelist values will be used to convert from pressure, height, model level, or scale heights into radians so the distances are computed in a consistent unit system. In practice, multiply the cutoff by the normalization factor (and then again by 2) to get the maximum vertical separation in each of the given units.

When including vertical separation the potential area of influence of an assimilated observation is an ellipsoid with the observation at the center. The horizontal radius is defined by the cutoff and the vertical radius is defined by the normalization factors.

See examples below for specific examples that highlight some vertical localization issues.

Different vertical factors per observation type

Generally a single cutoff value and a single set of normalization factors are sufficient for most assimilations. The localization distances define the maximum range of impact of an observation, but there still must be a positive or negative correlation between the state ensemble and the forward operator/expected obs ensemble for the values to change.

However, the &assim_tools_nml namelist includes the option to set a different cutoff on a per-observation-type basis. There are corresponding items in the location module namelist to similiarly control the vertical distance contribution on a per-observation, per-vertical-type basis.

Model-dependent vertical conversion issues

If the model supports either a different vertical coordinate than the vertical coordinate of the observations, or if the user wants to localize in a different vertical coordinate than the observations or state vector items, the model-specific `model_mod.f90` code will have to provide a conversion between different vertical coordinates. This cannot be done by the location module since most vertical conversions require additional model-specific information such as temperature, moisture, depth, surface elevation, model levels, etc.

Once the locations have the same vertical units the location module code can compute the distances between them. It is an error to ask for the distance between two locations with different vertical coordinates unless you have set the namelist to horizontal distances only.

There is a vertical type of VERTISUNDEF (Vertical is Undefined). This is used to describe observations where there is no specific single vertical location, for example the position of a hurricane or a column integrated quantity. In this case the location code computes only horizontal distances between any pair of observations in which one or both have an undefined vertical location.

Model-dependent distance adjustments

The calls to routines that collect the distances between locations for the assimilation code pass through the model-specific `model_mod.f90` code. This allows the code to alter the actual distances to either increase or decrease the effect of an observation on the state or on other observations.

For example, if the top of a model is externally constrained then modifications by the assimilation code may lead to bad results. The model-specific code can compute the actual distances between two locations and then increase it artificially as you reach the top of the model, so observations have smaller and smaller impacts. For ocean models, the distances to points on land can be set to a very large value and those points will be unaffected by the assimilation.

Approximate distances

For regional models this should usually be `.FALSE.` in the namelist.

For global models this is usually set to `.TRUE.` which allows the code to run slightly faster by precomputing tables of sines, cosines, and arc cosines used in the distance computations. Values are linearly interpolated between entries in the table which leads to minor roundoff errors. These are negligible in a global model but may be significant in models that over a small region of the globe.

Internal bin counts

The default settings for `nlon` and `nlat` are usually sufficient. However if this is a high resolution model with a large state vector the assimilation may run faster by doubling these values or multiplying them by 4. (The `nlon` item must be odd; compute the value and subtract 1.) These values set the number of internal bins used inside the code to pre-sort locations and make it faster to retrieve all locations close to another location. A larger bin count uses more memory but shortens the linear part of the location search.

Examples and questions involving vertical issues

Example of specifying a cutoff based on a distance in kilometers

The Earth radius is nominally 6,371 Km. If you want the maximum horizontal distance that an observation can possibly influence something in the model state to be X km, then set the cutoff to be (X / 6,371) / 2. Remember the actual impact will depend on a combination of this distance and the regression coefficient computed from the distribution of forward operator values and the ensemble of values in the model state.

Cutoff and half-widths

Q: Why is the cutoff specified as half the distance to where the impact goes to 0, and why is it called 'cutoff'?

A: Because the original paper by Gaspari & Cohn used that definition in this paper which our localization function is based on.

Gaspari, G. and Cohn, S. E. (1999), Construction of correlation functions in two and three dimensions. Q.J.R. Meteorol. Soc., 125: 723-757. doi:10.1002/qj.49712555417

Computing vertical normalization values

Because distances are computed in radians, the vertical distances have to be translated to radians. To get a maximum vertical separation of X meters (if localizing in height), specify the vert_normalization_height of X / cutoff. If localizing in pressure, specify vert_normalization_pressure as X pascals / cutoff, etc.

Single vertical coordinate type

Vertical distances can only be computed between two locations that have the same vertical type. In practice this means if vertical localization is enabled all observations which have a vertical location need to be converted to a single vertical coordinate type, which matches the desired localization unit. The model state must also be able to be converted to the same vertical coordinate type.

For example, if some observations come with a vertical coordinate type of pressure and some with height, and you want to localize in height, the pressure coordinates need to be converted to an equivalant height. This usually requires information only available to the model interface code in the model_mod.f90 file, so a convert_vertical_obs() routine is called to do the conversion.

The locations of the model state are returned by the get_state_meta_data() routine in the model_mod.f90 file. If the vertical coordinate used in the state is not the same as the desired vertical localization type, they must also be converted using a convert_vertical_state() routine.

6.92.3 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&location_nml
    horiz_dist_only                              = .true.
    vert_normalization_pressure                  = 100000.0
    vert_normalization_height                    = 10000.0
    vert_normalization_level                     = 20.0
    vert_normalization_scale_height              = 5.0
    approximate_distance                         = .false.
    nlon                                         = 71
    nlat                                         = 36
    output_box_info                              = .false.
    print_box_level                              = 0
    special_vert_normalization_obs_types         = 'null'
    special_vert_normalization_pressures         = -888888.0
    special_vert_normalization_heights           = -888888.0
    special_vert_normalization_levels            = -888888.0
    special_vert_normalization_scale_heights = -888888.0
  /
```

Items in this namelist either control the way in which distances are computed and/or influence the code performance.

Item	Type	Description
horiz_dist_only	logical	If .TRUE. compute great-circle distance using the horizontal distance component only. If .FALSE. compute distances by including the vertical and horizontal separation. All distances are computed in radians; the corresponding vertical normalization factors are used to compute the vertical distance. The vertical coordinate system must be the same for both locations in order to compute a distance. However, if either location is VERTISUNDEF, or both are VERTISSURFACE, only a horizontal distance is computed. For any other combination of vertical coordinate systems this routine will fail because it cannot convert between vertical coordinate systems without model-specific information. The model_mod interface code may supply a get_close_obs() routine to intercept and convert the vertical coordinates before calling this get_close_obs() routine.
vert_normalization_pressure	real(r8)	The number of pascals equivalent to a horizontal distance of one radian.
vert_normalization_height	real(r8)	The number of meters equivalent to a horizontal distance of one radian.
vert_normalization_level	real(r8)	The number of model levels equivalent to a horizontal distance of one radian.
vert_normalization_scale_height	real(r8)	The number of scale heights equivalent to a horizontal distance of one radian.
approximate_distance	logical	If true, uses a table lookup for fast approximate computation of distances on sphere. Distance computation can be a first order cost for some spherical problems so this can increase speed significantly at a loss of some precision. WARNING: This should be set to .FALSE. if you need to compute small distances accurately or you have a regional model.
nlon	integer	Used internally by the search code to speed the search for nearby locations. Number of boxes (bins) created in the longitude direction. Must be an odd number. (See discussion above for more information about this item.)
nlat	integer	Used internally by the search code to speed the search for nearby locations. Number of boxes (bins) created in the latitude direction. (See discussion above for more information about this item.)
output_box_info	logical	If true, print details about the distribution of locations across the array of boxes. `print_box_level` controls how much detail is printed.
print_box_level	integer	If `output_box_info = .true.`, `print_box_level` controls how much detail is printed. 0 = no detail. 1,2,3 are progressively more and more detail.
special_vert_normalization_obs_types	character(len=32), dimension(500)	If specified, must be a string array of observation specific types (e.g. RADIOSONDE_TEMPERATURE, AIRCRAFT_TEMPERATURE, etc). For each type listed here, a vertical normalization value must be given which overrides the default vertical normalization values. Even if only one is going to be used, all 4 normalization values must be specified for each special type.
special_vert_normalization_pressures	real(r8), dimension(500)	The number of pascals equivalent to a horizontal distance of one radian, one value for each special observation type listed in the ‘ special_vert_normalization_obs_types’ list.
special_vert_normalization_heights	real(r8), dimension(500)	The number of geopotential meters equivalent to a horizontal distance of one radian, one value for each special observation type listed in the ‘ special_vert_normalization_obs_types’ list.
sp ecial_vert_normalization_scale_heights	real(r8), dimension(500)	The number of scale heights equivalent to a horizontal distance of one radian, one value for each special observation type listed in the ‘ special_vert_normalization_obs_types’ list.
special_vert_normalization_levels	real(r8), dimension(500)	The number of model levels equivalent to a horizontal distance of one radian, one value for each special observation type listed in the ‘ special_vert_normalization_obs_types’ list.

6.92.4 Discussion

Location-independent code

All types of location modules define the same module name `location_mod`. Therefore, the DART framework and any user code should include a Fortran 90 `use` statement of `location_mod`. The selection of which location module will be compiled into the program is controlled by which source file name is specified in the `path_names_xxx` file, which is used by the `mkmf_xxx` scripts.

All types of location modules define the same Fortran 90 derived type `location_type`. Programs that need to pass location information to subroutines but do not need to interpret the contents can declare, receive, and pass this derived type around in their code independent of which location module is specified at compile time. Model and location-independent utilities should be written in this way. However, as soon as the contents of the location type needs to be accessed by user code then it becomes dependent on the exact type of location module that it is compiled with.

Usage of distance routines

Regardless of the fact that the distance subroutine names include the string 'obs', there is nothing specific to observations in these routines. They work to compute distances between any set of locations. The most frequent use of these routines in the filter code is to compute the distance between a single observation and items in the state vector, and also between a single observation and other nearby observations. However, any source for locations is supported.

In simpler location modules (like the `oned` version) there is no need for anything other than a brute force search between the base location and all available state vector locations. However in the case of large geophysical models which typically use the `threed_sphere` locations code, the brute-force search time is prohibitive. The location code pre-processes all locations into a set of *bins* and then only needs to search the lists of locations in nearby bins when looking for locations that are within a specified distance.

The expected calling sequence of the `get_close` routines is as follows:

```
call get_close_init()
...
call get_close_obs()           ! called many, many times
...
call get_close_destroy()
```

`get_close_init()` initializes the data structures, `get_close_obs()` is called multiple times to find all locations within a given radius of some reference location, and to optionally compute the exact separation distance from the reference location. `get_close_destroy()` deallocates the space. See the documentation below for the specific details for each routine.

All 3 of these routines must be present in every location module but in most other versions all but `get_close_obs()` are stubs. In this `threed_sphere` version of the locations module all are fully implemented.

Interaction with model_mod.f90 code

The filter and other DART programs could call the `get_close` routines directly, but typically do not. They declare them (in a `use` statement) to be in the `model_mod` module, and all model interface modules are required to supply them. However in many cases the model_mod only needs to contain another `use` statement declaring them to come from the `location_mod` module. Thus they 'pass through' the model_mod but the user does not need to provide a subroutine or any code for them.

However, if the model interface code wants to intercept and alter the default behavior of the get_close routines, it is able to. Typically the model_mod still calls the location_mod routines and then adjusts the results before passing them back to the calling code. To do that, the model_mod must be able to call the routines in the location_mod which have the same names as the subroutines it is providing. To allow the compiler to distinguish which routine is to be called where, we use the Fortran 90 feature which allows a module routine to be renamed in the use statement. For example, a common case is for the model_mod to want to supply additions to the get_close_obs() routine only. At the top of the model_mod code it would declare:

```
use location_mod, only :: get_close_init, get_close_destroy, &
                          location_get_close_obs => get_close_obs
```

That makes calls to the maxdist_init, init, and destroy routines simply pass through to the code in the location_mod, but the model_mod must supply a get_close_obs() subroutine. When it wants to call the code in the location_mod it calls `location_get_close_obs()`.

One use pattern is for the model_mod to call the location get_close_obs() routine without the `dist` argument. This returns a list of any potentially close locations without computing the exact distance from the base location. At this point the list of locations is a copy and the model_mod routine is free to alter the list in any way it chooses: it can change the locations to make certain types of locations appear closer or further away from the base location; it can convert the vertical coordinates into a common coordinate type so that calls to the `get_dist()` routine can do full 3d distance computations and not just 2d (the vertical coordinates must match between the base location and the locations in the list in order to compute a 3d distance). Then typically the model_mod code loops over the list calling the `get_dist()` routine to get the actual distances to be returned to the calling code. To localize in the vertical in a particular unit type, this is the place where the conversion to that vertical unit should be done.

Horizontal distance only

If *horiz_distance_only* is .true. in the namelist then the vertical coordinate is ignored and only the great-circle distance between the two locations is computed, as if they were both on the surface of the sphere.

If *horiz_distance_only* is .false. in the namelist then the appropriate normalization constant determines the relative impact of vertical and horizontal separation. Since only a single localization distance is specified, and the vertical scales might have very different distance characteristics, the vert_normalization_xxx values can be used to scale the vertical appropriately to control the desired influence of observations in the vertical.

Precomputation for run-time search efficiency

For search efficiency all locations are pre-binned. The surface of the sphere is divided up into *nlon* by *nlat* boxes and the index numbers of all items (both state vector entries and observations) are stored in the appropriate box. To locate all points close to a given location, only the locations listed in the boxes within the search radius must be checked. This speeds up the computations, for example, when localization controls which state vector items are impacted by any given observation. The search radius is the localization distance and only those state vector items in boxes closer than the radius to the observation location are processed.

The default values have given good performance on many of our existing model runs, but for tuning purposes the box counts have been added to the namelist to allow adjustment. By default the code prints some summary information about how full the average box is, how many are empty, and how many items were in the box with the largest count. The namelist value *output_box_info* can be set to .true. to get even more information about the box statistics. The best performance will be obtained somewhere between two extremes; the worst extreme is all the points are located in just a few boxes. This degenerates into a (slow) linear search through the index list. The other extreme is a large number of empty or sparsely filled boxes. The overhead of creating, managing, and searching a long list of boxes will impact performance. The best performance lies somewhere in the middle, where each box contains a reasonable number of values, more or less evenly distributed across boxes. The absolute numbers for best performance will certainly vary from case to case.

For latitude, the *nlat* boxes are distributed evenly across the actual extents of the data. (Locations are in radians, so the maximum limits are the poles at $-\pi/2$ and $+\pi/2$. For longitude, the code automatically determines if the data is spread around more than half the sphere, and if so, the boxes are distributed evenly across the entire sphere (longitude range 0 to 2π). If the data spans less than half the sphere in longitude, the actual extent of the data is determined (including correctly handling the cyclic boundary at 0) and the boxes are distributed only within the data extent. This simplifies the actual distance calculations since the distance from the minimum longitude box to the maximum latitude box cannot be shorter going the other way around the sphere. In practice, for a global model the boxes are evenly distributed across the entire surface of the sphere. For local or regional models, the boxes are distributed only across the the extent of the local grid.

For efficiency in the case where the boxes span less than half the globe, the 3D location module needs to be able to determine the greatest longitude difference between a base point at latitude ϕ_s and all points that are separated from that point by a central angle of θ. We might also want to know the latitude, ϕ_f, at which the largest separation occurs. Note also that an intermediate form below allows the computation of the maximum longitude difference at a particular latitude.

The central angle between a point at latitude ϕ_s and a second point at latitude ϕ_f that are separated in longitude by $\Delta\lambda$ is:

$$\theta = cos^{-1}(sin\phi_s sin\phi_f + cos\phi_s cos\phi_f cos\Delta\lambda)$$

Taking the cos of both sides gives:

$$cos\theta = (sin\phi_s sin\phi_f + cos\phi_s cos\phi_f cos\Delta\lambda)$$

Solving for $cos\Delta\lambda$ gives:

$$cos\Delta\lambda = \frac{a - bsin\phi_f}{ccos\phi_f}$$

$$cos\Delta\lambda = \frac{a}{csec\phi_f} - \frac{b}{ctan\phi_f}$$

where $a = cos\theta$, $b = sin\phi_s$, and $c = cos\phi_s$. We want to maximize $\Delta\lambda$ which implies minimizing $cos\Delta\lambda$ subject to constraints.

Taking the derivative with respect to ϕ_f gives:

$$\frac{dcos\Delta\lambda}{d\phi_f} = \frac{a}{csec\phi_f tan\phi_f} - \frac{b}{csec^2\phi_f} = 0$$

Factoring out $sec\phi_f$ which can never be 0 and using the definitions of sec and tan gives:

$$\frac{asin\phi_f}{ccos\phi_f} - \frac{b}{ccos\phi_f} = 0$$

Solving in the constrained range from 0 to $\pi/2$ gives:

$$sin\phi_f = \frac{b}{a} = \frac{sin\phi_s}{cos\theta}$$

So knowing base point (ϕ_s, λ_s), latitude ϕ_f, and distance θ we can use the great circle equation to find the longitude difference at the greatest separation point:

$$\Delta\lambda = cos^{-1}\left(\frac{a - bsin\phi_f}{ccos\phi_f}\right)$$

Note that if the angle between the base point and a pole is less than or equal to the central angle, all longitude differences will occur as the pole is approached.

6.92.5 Other modules used

```
types_mod
utilities_mod
random_seq_mod
obs_kind_mod
ensemble_manager_mod
```

6.92.6 Public interfaces

`use location_mod, only :`	location_type
	get_close_type
	get_location
	set_location
	write_location
	read_location
	interactive_location
	set_location_missing
	query_location
	get_close_init
	get_close_obs
	get_close_destroy
	get_dist
	get_maxdist
	LocationDims
	LocationName
	LocationLName
	horiz_dist_only
	vert_is_undef
	vert_is_surface
	vert_is_pressure
	vert_is_scale_height
	vert_is_level
	vert_is_height
	VERTISUNDEF
	VERTISSURFACE
	VERTISLEVEL
	VERTISPRESSURE
	VERTISHEIGHT
	VERTISSCALEHEIGHT
	operator(==)
	operator(/=)

Namelist interface `&location_nml` must be read from file `input.nml`.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

type location_type

```
   private
   real(r8) :: lon, lat, vloc
   integer  :: which_vert
end type location_type
```

Provides an abstract representation of physical location on a three-d spherical shell.

Component	Description
lon	longitude in radians
lat	latitude in radians
vloc	vertical location, units as selected by which_vert
which_vert	type of vertical location: -2=no specific vert location; -1=surface; 1=level; 2=pressure; 3=height, 4=scale height

The vertical types have parameters defined for them so they can be referenced by name instead of number.

type get_close_type

```
   private
   integer  :: num
   real(r8) :: maxdist
   integer, pointer :: lon_offset(:, :)
   integer, pointer :: obs_box(:)
   integer, pointer :: count(:, :)
   integer, pointer :: start(:, :)
end type get_close_type
```

Provides a structure for doing efficient computation of close locations.

Component	Description
num	Number of locations in list
maxdist	Threshhold distance. Anything closer is close.
lon_offset	Dimensioned nlon by nlat. For a given offset in longitude boxes and difference in latitudes, gives max distance from base box to a point in offset box.
obs_box	Dimensioned num. Gives index of what box each location is in.
count	Dimensioned nlon by nlat. Number of obs in each box.
start	Dimensioned nlon by nlat. Index in straight storage list where obs in each box start.

var = get_location(loc)

```
real(r8), dimension(3)          :: get_location
type(location_type), intent(in) :: loc
```

Extracts the longitude and latitude (converted to degrees) and the vertical location from a location type and returns in a 3 element real array.

`get_location`	The longitude and latitude (in degrees) and vertical location
`loc`	A location type

var = set_location(lon, lat, vert_loc, which_vert)

```
type(location_type) :: set_location
real(r8), intent(in)     :: lon
real(r8), intent(in)     :: lat
real(r8), intent(in)     :: vert_loc
integer,  intent(in)     :: which_vert
```

Returns a location type with the input longitude and latitude (input in degrees) and the vertical location of type specified by which_vert.

`set_location`	A location type
`lon`	Longitude in degrees
`lat`	Latitude in degrees
`vert_loc`	Vertical location consistent with which_vert
`which_vert`	The vertical location type

call write_location(locfile, loc [, fform, charstring])

```
integer,                intent(in)       ::  locfile
type(location_type),    intent(in)       ::  loc
character(len=*), optional, intent(in)   ::  fform
character(len=*), optional, intent(out)  ::  charstring
```

Given an integer IO channel of an open file and a location, writes the location to this file. The *fform* argument controls whether write is “FORMATTED” or “UNFORMATTED” with default being formatted. If the final *charstring* argument is specified, the formatted location information is written to the character string only, and the `locfile` argument is ignored.

`locfile`	the unit number of an open file.
`loc`	location type to be written.
fform	Format specifier (“FORMATTED” or “UNFORMATTED”). Default is “FORMATTED” if not specified.
charstring	Character buffer where formatted location string is written if present, and no output is written to the file unit.

var = read_location(locfile [, fform])

```
type(location_type)                      :: read_location
integer, intent(in)                      :: locfile
character(len=*), optional, intent(in)  :: fform
```

Reads a location_type from a file open on channel locfile using format *fform* (default is formatted).

`read_location`	Returned location type read from file
`locfile`	Integer channel opened to a file to be read
fform	Optional format specifier ("FORMATTED" or "UNFORMATTED"). Default "FORMATTED".

call interactive_location(location [, set_to_default])

```
type(location_type), intent(out) :: location
logical, optional, intent(in)    :: set_to_default
```

Use standard input to define a location type. With set_to_default true get one with all elements set to 0.

`location`	Location created from standard input
set_to_default	If true, sets all elements of location type to 0

var = query_location(loc [, attr])

```
real(r8)                                 :: query_location
type(location_type), intent(in)          :: loc
character(len=*), optional, intent(in)  :: attr
```

Returns the value of which_vert, latitude, longitude, or vertical location from a location type as selected by the string argument attr. If attr is not present or if it is 'WHICH_VERT', the value of which_vert is converted to real and returned. Otherwise, attr='LON' returns longitude, attr='LAT' returns latitude and attr='VLOC' returns the vertical location.

`query_location`	Returns longitude, latitude, vertical location, or which_vert (converted to real)
`loc`	A location type
attr	Selects 'WHICH_VERT', 'LON', 'LAT' or 'VLOC'

var = set_location_missing()

```
type(location_type) :: set_location_missing
```

Returns a location with all elements set to missing values defined in types module.

`set_location_missing`	A location with all elements set to missing values

call get_close_init(gc, num, maxdist, locs [,maxdist_list])

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
real(r8),             intent(in)    :: maxdist
type(location_type),  intent(in)    :: locs(:)
real(r8), optional,   intent(in)    :: maxdist_list(:)
```

Initializes the get_close accelerator. `maxdist` is in units of radians. Anything closer than this is deemed to be close. This routine must be called first, before the other `get_close` routines. It allocates space so it is necessary to call `get_close_destroy` when completely done with getting distances between locations.

If the last optional argument is not specified, `maxdist` applies to all locations. If the last argument is specified, it must be a list of exactly the length of the number of specific types in the `obs_kind_mod.f90` file. This length can be queried with the get_num_types_of_obs() function to get count of obs types. It allows a different maximum distance to be set per base type when `get_close()` is called.

`gc`	Data for efficiently finding close locations.
`num`	The number of locations, i.e. the length of the `locs` array.
`maxdist`	Anything closer than this number of radians is a close location.
`locs`	The list of locations in question.
maxdist	If specified, must be a list of real values. The length of the list must be exactly the same length as the number of observation types defined in the obs_def_kind.f90 file. (See get_num_types_of_obs() to get count of obs types.) The values in this list are used for the obs types as the close distance instead of the maxdist argument.

call get_close_obs(gc, base_obs_loc, base_obs_type, obs, obs_kind, num_close, close_ind [, dist, ens_handle])

```
type(get_close_type),                 intent(in)  :: gc
type(location_type),                  intent(in)  :: base_obs_loc
integer,                              intent(in)  :: base_obs_type
type(location_type), dimension(:),    intent(in)  :: obs
integer,             dimension(:),    intent(in)  :: obs_kind
integer,                              intent(out) :: num_close
integer,             dimension(:),    intent(out) :: close_ind
real(r8), optional,  dimension(:),    intent(out) :: dist
type(ensemble_type), optional,        intent(in)  :: ens_handle
```

Given a single location and a list of other locations, returns the indices of all the locations close to the single one along with the number of these and the distances for the close ones. The list of locations passed in via the `obs` argument must be identical to the list of `obs` passed into the most recent call to `get_close_init()`. If the list of locations of interest changes `get_close_destroy()` must be called and then the two initialization routines must be called before using `get_close_obs()` again.

Note that the base location is passed with the specific type associated with that location. The list of potential close locations is matched with a list of generic kinds. This is because in the current usage in the DART system the base location is always associated with an actual observation, which has both a specific type and generic kind. The list of potentially close locations is used both for other observation locations but also for state variable locations which only have a generic kind.

If called without the optional *dist* argument, all locations that are potentially close are returned, which is likely a superset of the locations that are within the threshold distance specified in the `get_close_init()` call. This can be useful to collect a list of potential locations, and then to convert all the vertical coordinates into one consistent unit (pressure, height in meters, etc), and then the list can be looped over, calling get_dist() directly to get the exact distance, either including vertical or not depending on the setting of `horiz_dist_only`.

`gc`	Structure to allow efficient identification of locations close to a given location.
`base_obs_loc`	Single given location.
`base_obs_type`	Specific type of the single location.
`obs`	List of locations from which close ones are to be found.
`obs_kind`	Generic kind associated with locations in obs list.
`num_close`	Number of locations close to the given location.
`close_ind`	Indices of those locations that are close.
dist	Distance between given location and the close ones identified in close_ind.
ens_handle	Handle to an ensemble of interest.

call get_close_destroy(gc)

```
type(get_close_type), intent(inout) :: gc
```

Releases memory associated with the `gc` derived type. Must be called whenever the list of locations changes, and then `get_close_init` must be called again with the new locations list.

`gc`	Data for efficiently finding close locations.

var = get_dist(loc1, loc2, [, type1, kind2, no_vert])

```
real(r8)                            :: get_dist
type(location_type), intent(in)     :: loc1
type(location_type), intent(in)     :: loc2
integer, optional,   intent(in)     :: type1
integer, optional,   intent(in)     :: kind2
logical, optional,   intent(in)     :: no_vert
```

Returns the distance between two locations in radians. If `horiz_dist_only` is set to .TRUE. in the locations namelist, it computes great circle distance on sphere. If `horiz_dist_only` is false, then it computes an ellipsoidal distance with the horizontal component as above and the vertical distance determined by the types of the locations and the normalization constants set by the namelist for the different vertical coordinate types. The vertical normalization gives the vertical distance that is equally weighted as a horizontal distance of 1 radian. If *no_vert* is present, it overrides the value in the namelist and controls whether vertical distance is included or not.

The type and kind arguments are not used by the default location code, but are available to any user-supplied distance routines which want to do specialized calculations based on the types/kinds associated with each of the two locations.

`loc1`	First of two locations to compute distance between.
`loc2`	Second of two locations to compute distance between.
type1	DART specific type associated with location 1.
kind2	DART generic kind associated with location 2.
no_vert	If true, no vertical component to distance. If false, vertical component is included.
`var`	distance between loc1 and loc2.

var = get_maxdist(gc [, obs_type])

```
real(r8)                              :: var
type(get_close_type), intent(inout) :: gc
integer, optional,    intent(in)    :: obs_type
```

Since it is possible to have different cutoffs for different observation types, an optional argument *obs_type* may be used to specify which maximum distance is of interest. The cutoff is specified as the half-width of the tapering function, `get_maxdist` returns the full width of the tapering function.

`gc`	Data for efficiently finding close locations.
obs_type	The integer code specifying the type of observation.
`var`	The distance at which the tapering function is zero. Put another way, anything closer than this number of radians is a close location.

var = vert_is_undef(loc)

```
logical                          :: vert_is_undef
type(location_type), intent(in) :: loc
```

Returns true if which_vert is set to undefined, else false. The meaning of ‘undefined’ is specific; it means there is no particular vertical location associated with this type of measurement; for example a column-integrated value.

`vert_is_undef`	Returns true if vertical coordinate is set to undefined.
`loc`	A location type

var = vert_is_surface(loc)

```
logical                          :: vert_is_surface
type(location_type), intent(in) :: loc
```

Returns true if which_vert is for surface, else false.

vert_is_surface	Returns true if vertical coordinate type is surface
loc	A location type

var = vert_is_pressure(loc)

```
logical                            :: vert_is_pressure
type(location_type), intent(in)    :: loc
```

Returns true if which_vert is for pressure, else false.

vert_is_pressure	Returns true if vertical coordinate type is pressure
loc	A location type

var = vert_is_scale_height(loc)

```
logical                            :: vert_is_scale_height
type(location_type), intent(in)    :: loc
```

Returns true if which_vert is for scale_height, else false.

vert_is_scale_height	Returns true if vertical coordinate type is scale_height
loc	A location type

var = vert_is_level(loc)

```
logical                            :: vert_is_level
type(location_type), intent(in)    :: loc
```

Returns true if which_vert is for level, else false.

vert_is_level	Returns true if vertical coordinate type is level
loc	A location type

var = vert_is_height(loc)

```
logical                            :: vert_is_height
type(location_type), intent(in)  :: loc
```

Returns true if which_vert is for height, else false.

`vert_is_height`	Returns true if vertical coordinate type is height
`loc`	A location type

var = has_vertical_localization()

```
logical :: has_vertical_localization
```

Returns .TRUE. if the namelist variable `horiz_dist_only` is .FALSE. meaning that vertical separation between locations is going to be computed by `get_dist()` and by `get_close_obs()`.

This routine should perhaps be renamed to something like 'using_vertical_for_distance' or something similar. The current use for it is in the localization code inside filter, but that doesn't make this a representative function name. And at least in current usage, returning the opposite setting of the namelist item makes the code read more direct (fewer double negatives).

loc1 == loc2

```
type(location_type), intent(in) :: loc1, loc2
```

Returns true if the two location types have identical values, else false.

loc1 /= loc2

```
type(location_type), intent(in) :: loc1, loc2
```

Returns true if the two location types do NOT have identical values, else false.

```
integer, parameter :: VERTISUNDEF       = -2
integer, parameter :: VERTISSURFACE     = -1
integer, parameter :: VERTISLEVEL       =  1
integer, parameter :: VERTISPRESSURE    =  2
integer, parameter :: VERTISHEIGHT      =  3
integer, parameter :: VERTISSCALEHEIGHT =  4
```

Constant parameters used to differentiate vertical types.

```
integer, parameter :: LocationDims = 3
```

This is a **constant**. Contains the number of real values in a location type. Useful for output routines that must deal transparently with many different location modules.

```
character(len=129), parameter :: LocationName = "loc3Dsphere"
```

This is a **constant**. A parameter to identify this location module in output metadata.

```
character(len=129), parameter :: LocationLName =

       "threed sphere locations: lon, lat, vertical"
```

This is a **constant**. A parameter set to “threed sphere locations: lon, lat, vertical” used to identify this location module in output long name metadata.

6.92.7 Files

filename	purpose
input.nml	to read the location_mod namelist

6.92.8 References

1. none

6.92.9 Error codes and conditions

Routine	Message	Comment
initialize_module	nlon must be odd	Tuning parameter for number of longitude boxes must be odd for algorithm to function.
get_dist	Dont know how to compute vertical distance for unlike vertical coordinates	Need same which_vert for distances.
set_location	longitude (#) is not within range [0,360]	Is it really a longitude?
set_location	latitude (#) is not within range [-90,90]	Is it really a latitude?
set_location	which_vert (#) must be one of -2, -1, 1, 2, 3, or 4	Vertical coordinate type restricted to: -2 = no specific vertical location -1 = surface value 1 = (model) level 2 = pressure 3 = height 4 = scale height
read_location	Expected location header “loc3d” in input file, got ___	Probable mixing of other location modules in observation sequence processing.
nc_write_location	Various NetCDF-f90 interface error messages	From one of the NetCDF calls in nc_write_location

6.92.10 Future plans

Need to provide more efficient algorithms for getting close locations and document the nlon and nlat choices and their impact on cost.

The collection of ‘val = vert_is_xxx()’ routines should probably be replaced by a single call ‘val = vert_is(loc, VERTISxxx)’.

See the note in the ‘has_vertical_localization()’ about a better name for this routine.

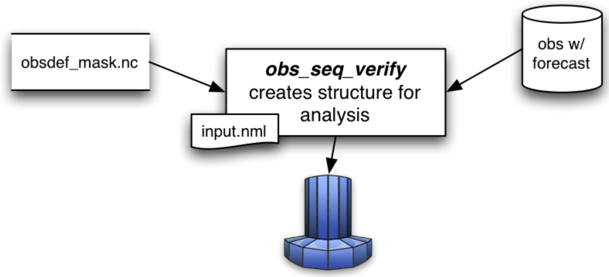
The use of ‘obs’ in all these routine names should probably be changed to ‘loc’ since there is no particular dependence that they be observations. They may need to have an associated DART kind, but these routines are used for DART state vector entries so it’s misleading to always call them ‘obs’.

6.92.11 Private components

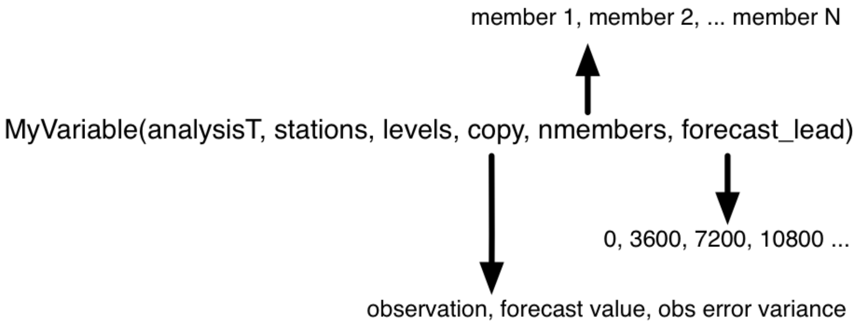
N/A

6.93 program `obs_seq_verify`

6.93.1 Overview



`obs_seq_verify` reorders the observations from a forecast run of DART into a structure that is amenable for the evaluation of the forecast. The big picture is that the verification locations and times identified in the `obsdef_mask.nc` and the observations from the forecast run (whose files **must** have an extension as in the following: `obs_seq.forecast.YYYYMMDDHH`) are put into a netCDF variable that looks like this:


member 1, member 2, ... member N
MyVariable(analysisT, stations, levels, copy, nmembers, forecast_lead)
0, 3600, 7200, 10800 ...
observation, forecast value, obs error variance

`obs_seq_verify` can read in a series of observation sequence files - each of the files **must** contain the **entire forecast from a single analysis time**. The extension of each filename is **required** to reflect the analysis time. Use *program obs_sequence_tool* to concatenate multiple files into a single observation sequence file if necessary. *Only the individual ensemble members forecast values are used - the ensemble mean and spread (as individual copies) are completely ignored.* The individual "*prior ensemble member NNNN*" copies are used. As a special case, the "*prior ensemble mean*" copy is used *if and only if* there are no individual ensemble members present (i.e. `input.nml &filter_nml:num_output_obs_members` == *0*).

Dimension	Explanation
analysisT	This is the netCDF UNLIMITED dimension, so it is easy to 'grow' this dimension. This corresponds to the number of forecasts one would like to compare.
stations	The unique horizontal locations in the verification network.
levels	The vertical level at each location. Observations with a pressure vertical coordinate are selected based on their proximity to the mandatory levels as defined in *program obs_seq_coverage*. Surface observations or observations with undefined vertical coordinates are simply put into level 1.
copy	This dimension designates the quantity of interest; the observation, the forecast value, or the observation error variance. These quantities are the ones required to calculate the evaluation statistics.
nmembers	Each ensemble member contributes a forecast value.
forecast_lead	This dimension relates to the amount of time between the start of the forecast and the verification.

The USAGE section has more on the actual use of `obs_seq_verify`.

6.93.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_seq_verify_nml
   obs_sequences      = ''
   obs_sequence_list  = ''
   station_template   = 'obsdef_mask.nc'
   netcdf_out         = 'forecast.nc'
   obtype_string      = 'RADIOSONDE_TEMPERATURE'
   print_every        = 10000
   verbose            = .true.
   debug              = .false.
   /
```

You can specify **either** `obs_sequences` **or** `obs_sequence_list` – not both. One of them has to be an empty string … i.e. ' '.

Item	Type	Description
obs_sequences	character(len=256), dimension(500)	Names of the observation sequence files - each of which **MUST** have an extension that defines the start of the forecast (the analysis time). The observation sequence filenames must be something like `obs_seq.forecast.YYYYMMDDHH` . If `obs_sequences` is specified, `obs_sequence_list` must be empty.
obs_sequence_list	character(len=256)	Name of an ascii text file which contains a list of one or more observation sequence files, one per line. The observation sequence filenames **MUST** have an extension that defines the start of the forecast (the analysis time). The observation sequence filenames must be something like `obs_seq.forecast.YYYYMMDDHH`. `obs_sequence_list` can be created by any method, including sending the output of the 'ls' command to a file, a text editor, or another program. If `obs_sequence_list` is specified, `obs_sequences` must be empty.
station_template	character(len=256)	The name of the netCDF file created by *program obs_seq_coverage* that contains the verification network description.
netcdf_out	character(len=256)	The base portion of the filename of the file that will contain the forecast quantities. Since each observation type of interest is processed with a separate run of `obs_seq_verify`, the observation type string is used to create a unique output filename.
calendar	character(len=129)	The type of the calendar used to interpret the dates.
obtype_string	character(len=32)	The observation type string that will be verified. The character string must match one of the standard DART observation types. This will be the name of the variable in the netCDF file, and will also be used to make a unique netCDF file name.
print_every	integer	Print run-time information for every `"print_every"` *n*-th observation.
verbose	logical	Print extra run-time information.
debug	logical	Print a frightening amount of run-time information.

6.93.3 Other modules used

```
assimilation_code/location/threed_sphere/location_mod.f90
assimilation_code/modules/assimilation/assim_model_mod.f90
models/your_model/model_mod.f90
assimilation_code/modules/observations/obs_kind_mod.f90
assimilation_code/modules/observations/obs_sequence_mod.f90
assimilation_code/modules/utilities/null_mpi_utilities_mod.f90
assimilation_code/modules/utilities/types_mod.f90
assimilation_code/modules/utilities/random_seq_mod.f90
assimilation_code/modules/utilities/time_manager_mod.f90
assimilation_code/modules/utilities/utilities_mod.f90
observations/forward_operators/obs_def_mod.f90
```

6.93.4 Files

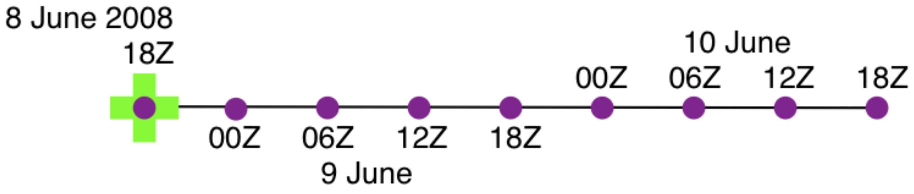
- `input.nml` is used for *obs_seq_verify_nml*
- A netCDF file containing the metadata for the verification network. This file is created by *program obs_seq_coverage* to define the desired times and locations for the verification. (`obsdef_mask.nc` is the default name)
- One or more observation sequence files from `filter` run in *forecast* mode - meaning all the observations were flagged as *evaluate_only*. It is required/presumed that all the ensemble members are output to the observation sequence file (see num_output_obs_members). Each observation sequence file contains all the forecasts from a single analysis time and the filename extension must reflect the analysis time used to start the forecast. (`obs_seq.forecast.YYYYMMDDHH` is the default name)
- Every execution of `obs_seq_verify` results in one netCDF file that contains the observation being verified. If `obtype_string = 'METAR_U_10_METER_WIND'`, and `netcdf_out = 'forecast.nc'`; the resulting filename will be `METAR_U_10_METER_WIND_forecast.nc`.

6.93.5 Usage

`obs_seq_verify` is built in . . . /DART/models/*your_model*/work, in the same way as the other DART components.

Once the forecast has completed, each observation type may be extracted from the observation sequence file and stuffed into the appropriate verification structure. Each observation type must be processed serially at this time, and each results in a separate output netCDF file. Essentially, `obs_seq_verify` sorts an unstructured, unordered set of observations into a predetermined configuration.

Example: a single 48-hour forecast that is evaluated every 6 hours



In this example, the `obsdef_mask.nc` file was created by running *program obs_seq_coverage* with the namelist specified in the single 48hour forecast evaluated every 6 hours example. The `obsdef_mask.txt` file was used to mask the input observation sequence files by *program obs_selection* and the result was run through *PROGRAM filter* with the observations marked as *evaluate_only* - resulting in a file called `obs_seq.forecast.2008060818`. This filename could also be put in a file called `verify_list.txt`.

Just to reiterate the example, both namelists for `obs_seq_coverage` and `obs_seq_verify` are provided below.

```
&obs_seq_coverage_nml
   obs_sequences       = ''
   obs_sequence_list   = 'coverage_list.txt'
   obs_of_interest     = 'METAR_U_10_METER_WIND'
                         'METAR_V_10_METER_WIND'
   textfile_out        = 'obsdef_mask.txt'
   netcdf_out          = 'obsdef_mask.nc'
   calendar            = 'Gregorian'
   first_analysis      =  2008, 6, 8, 18, 0, 0
```

(continues on next page)

(continued from previous page)

```
   last_analysis      =  2008, 6, 8, 18, 0, 0
   forecast_length_days          = 2
   forecast_length_seconds       = 0
   verification_interval_seconds = 21600
   temporal_coverage_percent     = 100.0
   lonlim1            =    0.0
   lonlim2            =  360.0
   latlim1            =  -90.0
   latlim2            =   90.0
   verbose            = .true.
   /

&obs_seq_verify_nml
   obs_sequences      = 'obs_seq.forecast.2008060818'
   obs_sequence_list  = ''
   station_template  = 'obsdef_mask.nc'
   netcdf_out        = 'forecast.nc'
   obtype_string     = 'METAR_U_10_METER_WIND'
   print_every       = 10000
   verbose           = .true.
   debug             = .false.
   /
```

The pertinent information from the `obsdef_mask.nc` file is summarized (from *ncdump -v experiment_times,analysis,forecast_lead obsdef_mask.nc*) as follows:

```
verification_times = 148812.75, 148813, 148813.25, 148813.5, 148813.75,
                                148814, 148814.25, 148814.5, 148814.75 ;

analysis           = 148812.75 ;

forecast_lead      = 0, 21600, 43200, 64800, 86400, 108000, 129600, 151200, 172800 ;
```

There is one analysis time, 9 forecast leads and 9 verification times. The analysis time is the same as the first verification time. The run-time output of `obs_seq_verify` and a dump of the resulting netCDF file follows:

```
[thoar@mirage2 work]$ ./obs_seq_verify |& tee my.verify.log
 Starting program obs_seq_verify
 Initializing the utilities module.
 Trying to log to unit           10
 Trying to open file dart_log.out

 --------------------------------------
 Starting ... at YYYY MM DD HH MM SS =
                 2011  3  1 10  2 54
 Program obs_seq_verify
 --------------------------------------

 set_nml_output Echo NML values to log file only
 Trying to open namelist log dart_log.nml
 ------------------------------------------------------



 -------------- ASSIMILATE_THESE_OBS_TYPES --------------
 RADIOSONDE_TEMPERATURE
 RADIOSONDE_U_WIND_COMPONENT
```

(continues on next page)

(continued from previous page)

```
 RADIOSONDE_V_WIND_COMPONENT
 SAT_U_WIND_COMPONENT
 SAT_V_WIND_COMPONENT
 -------------- EVALUATE_THESE_OBS_TYPES --------------
 RADIOSONDE_SPECIFIC_HUMIDITY
 ------------------------------------------------------

 find_ensemble_size:  opening obs_seq.forecast.2008060818
 location_mod: Ignoring vertical when computing distances; horizontal only
 find_ensemble_size: There are   50 ensemble members.

 fill_stations:  There are          221 stations of interest,
 fill_stations: ...  and              9 times    of interest.
 InitNetCDF:  METAR_U_10_METER_WIND_forecast.nc is fortran unit            5

 obs_seq_verify:  opening obs_seq.forecast.2008060818
 analysis            1 date is 2008 Jun 08 18:00:00

 index    6 is prior ensemble member      1
 index    8 is prior ensemble member      2
 index   10 is prior ensemble member      3
 ...
 index  100 is prior ensemble member     48
 index  102 is prior ensemble member     49
 index  104 is prior ensemble member     50

 QC index           1  NCEP QC index
 QC index           2  DART quality control

 Processing obs        10000  of        84691
 Processing obs        20000  of        84691
 Processing obs        30000  of        84691
 Processing obs        40000  of        84691
 Processing obs        50000  of        84691
 Processing obs        60000  of        84691
 Processing obs        70000  of        84691
 Processing obs        80000  of        84691

 METAR_U_10_METER_WIND dimlen            1  is            9
 METAR_U_10_METER_WIND dimlen            2  is           50
 METAR_U_10_METER_WIND dimlen            3  is            3
 METAR_U_10_METER_WIND dimlen            4  is            1
 METAR_U_10_METER_WIND dimlen            5  is          221
 METAR_U_10_METER_WIND dimlen            6  is            1
 obs_seq_verify:  Finished successfully.

 --------------------------------------
 Finished ... at YYYY MM DD HH MM SS =
                 2011  3  1 10  3  7
 --------------------------------------

[thoar@mirage2 work]$ ncdump -h METAR_U_10_METER_WIND_forecast.nc
netcdf METAR_U_10_METER_WIND_forecast {
dimensions:
        analysisT = UNLIMITED ; // (1 currently)
        copy = 3 ;
        station = 221 ;
```

(continues on next page)

(continued from previous page)

```
        level = 14 ;
        ensemble = 50 ;
        forecast_lead = 9 ;
        linelen = 129 ;
        nlines = 446 ;
        stringlength = 64 ;
        location = 3 ;
variables:
        char namelist(nlines, linelen) ;
                namelist:long_name = "input.nml contents" ;
        char CopyMetaData(copy, stringlength) ;
                CopyMetaData:long_name = "copy quantity names" ;
        double analysisT(analysisT) ;
                analysisT:long_name = "time of analysis" ;
                analysisT:units = "days since 1601-1-1" ;
                analysisT:calendar = "Gregorian" ;
                analysisT:missing_value = 0. ;
                analysisT:_FillValue = 0. ;
        int copy(copy) ;
                copy:long_name = "observation copy" ;
                copy:note1 = "1 == observation" ;
                copy:note2 = "2 == prior" ;
                copy:note3 = "3 == observation error variance" ;
                copy:explanation = "see CopyMetaData variable" ;
        int station(station) ;
                station:long_name = "station index" ;
        double level(level) ;
                level:long_name = "vertical level of observation" ;
        int ensemble(ensemble) ;
                ensemble:long_name = "ensemble member" ;
        int forecast_lead(forecast_lead) ;
                forecast_lead:long_name = "forecast lead time" ;
                forecast_lead:units = "seconds" ;
        double location(station, location) ;
                location:description = "location coordinates" ;
                location:location_type = "loc3Dsphere" ;
                location:long_name = "threed sphere locations: lon, lat, vertical" ;
                location:storage_order = "Lon Lat Vertical" ;
                location:units = "degrees degrees which_vert" ;
        int which_vert(station) ;
                which_vert:long_name = "vertical coordinate system code" ;
                which_vert:VERTISUNDEF = -2 ;
                which_vert:VERTISSURFACE = -1 ;
                which_vert:VERTISLEVEL = 1 ;
                which_vert:VERTISPRESSURE = 2 ;
                which_vert:VERTISHEIGHT = 3 ;
                which_vert:VERTISSCALEHEIGHT = 4 ;
        double METAR_U_10_METER_WIND(analysisT, station, level, copy, ensemble,␣
↪forecast_lead) ;
                METAR_U_10_METER_WIND:long_name = "forecast variable quantities" ;
                METAR_U_10_METER_WIND:missing_value = -888888. ;
                METAR_U_10_METER_WIND:_FillValue = -888888. ;
        int original_qc(analysisT, station, forecast_lead) ;
                original_qc:long_name = "original QC value" ;
                original_qc:missing_value = -888888 ;
                original_qc:_FillValue = -888888 ;
        int dart_qc(analysisT, station, forecast_lead) ;
```

(continues on next page)

(continued from previous page)

```
                dart_qc:long_name = "DART QC value" ;
                dart_qc:explanation1 = "1 == prior evaluated only" ;
                dart_qc:explanation2 = "4 == forward operator failed" ;
                dart_qc:missing_value = -888888 ;
                dart_qc:_FillValue = -888888 ;
// global attributes:
                :creation_date = "YYYY MM DD HH MM SS = 2011 03 01 10 03 00" ;
                :source = "$URL$" ;
                :revision = "$Revision$" ;
                :revdate = "$Date$" ;
                :obs_seq_file_001 = "obs_seq.forecast.2008060818" ;
}
[thoar@mirage2 work]$
```

Discussion

- the values of *ASSIMILATE_THESE_OBS_TYPES* and *EVALUATE_THESE_OBS_TYPES* are completely irrelevant - again - since `obs_seq_verify` is not actually doing an assimilation.
- The analysis time from the filename is used to determine which analysis from `obsdef_mask.nc` is being considered, and which set of verification times to look for. This is important.
- The individual `prior ensemble member` copies must be present! Since there are no observations being assimilated, there is no reason to choose the posteriors over the priors.
- There are 221 locations reporting METAR_U_10_METER_WIND observations at all 9 requested verification times.
- The `METAR_U_10_METER_WIND_forecast.nc` file has all the metadata to be able to interpret the *METAR_U_10_METER_WIND* variable.
- The *analysisT* dimension is the netCDF record/unlimited dimension. Should you want to increase the strength of the statistical results, you should be able to trivially `ncrcat` more (compatible) netCDF files together.

6.93.6 References

- none - but this seems like a good place to start: The Centre for Australian Weather and Climate Research - Forecast Verification Issues, Methods and FAQ

6.94 PROGRAM `wakeup_filter`

6.94.1 Overview

Small auxiliary program for use in the “async=4” case where the main filter program is an MPI program and the model being run with DART is also an MPI program. The main MPI job script runs each of the model advances for the ensemble members, and then runs this program to restart the filter program.

6.94.2 Modules used

```
mpi_utilities_mod
```

6.94.3 Namelist

There are no namelist options for this program. It must be run as an MPI program with the same number of tasks as filter was originally started with.

6.94.4 Files

Named pipes (fifo) files are used to synchronize with the main MPI job run script, to ensure that the filter program and the script do not do a "busy-wait" in which they consume CPU cycles while they are waiting for each other. The fifo names are:

- filter_to_model.lock
- model_to_filter.lock
- filter_lockNNNNN (where NNNNN is the task number with leading 0s)

6.94.5 References

- Anderson, J., T. Hoar, K. Raeder, H. Liu, N. Collins, R. Torn, and A. Arellano, 2009: The Data Assimilation Research Testbed: A Community Facility. Bull. Amer. Meteor. Soc., 90, 1283-1296. DOI: 10.1175/2009BAMS2618.1

6.95 PROGRAM compare_states

6.95.1 Overview

Utility program to compare fields in two NetCDF files and print out the min and max values from each file and the min and max of the differences between the two fields. The default is to compare all numeric variables in the files, but specific variables can be specified in the namelist or in a separate file. The two input NetCDF filenames are read from the console or can be echo'd into the standard input of the program.

If you want to restrict the comparison to only specific variables in the files, specify the list of field names to compare either in the namelist, or put a list of fields, one per line, in a text file and specify the name of the text file. Only data arrays can be compared, not character arrays, strings, or attribute values.

Namelist interface `&compare_states_nml` must be read from file `input.nml`.

6.95.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand ‘&’ and terminate with a slash ‘/’. Character strings that contain a ‘/’ must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&compare_states_nml
   do_all_numeric_fields   = .true.
   fieldnames              = ''
   fieldlist_file          = ''
   fail_on_missing_field   = .true.
   only_report_differences = .true.
   debug                   = .false.
  /
```

Item	Type	Description
do_all_numeric_fields	logical	If .true., all integer, float, and double variables in the NetCDF files will have their values compared. If .false. the list of specific variables to be compared must be given either directly in the namelist in the `fieldnames` item, or else the field names must be listed in an ASCII file, one name per line, and the name of that file is specified in `fieldlist_file`.
fieldnames	character list	One or more names of arrays in the NetCDF files to be compared. Only read if `do_all_numeric_fields` is .false.
fieldlist_file	character	Name of a text file containing the fieldnames, one per line. It is an error to specify both the fieldnames namelist item and this one. Only read if `do_all_numeric_fields` is .false.
fail_on_missing_field	logical	If .true. and any one of the field names is not found in both files it is a fatal error. If .false. a message is printed about the missing field but execution continues.
only_report_differences	logical	If .true. only print the name of the variable being tested; skip printing the variable value min and max if the two files are identical. If .false. print more details about both variables which differ and varibles with the same values.
debug	logical	If true print out debugging info.

6.95.3 Modules used

```
types_mod
utilities_mod
parse_args_mod
```

6.95.4 Files

- two NetCDF input files
- compare_states.nml
- field names text file (optionally)

6.95.5 References

- none

6.96 PROGRAM gen_sampling_err_table

6.96.1 Overview

Utility program which computes a table of values needed to apply Sampling Error Correction (SEC) during assimilation. These values are used to correct covariances based on small sample size statistics. See reference below.

The name of the SEC table is always sampling_error_correction_table.nc. This is a NetCDF format file. If this file already exists in the current directory any tables for new ensemble sizes will be appended to the existing file. If the file does not exist a new file will be created by this tool. The resulting file should be copied into the current working directory when filter is run.

A file with 40 common ensemble sizes is distributed with the system. Any new ensemble sizes can be generated on demand. Be aware that the computation can be time consuming. The job may need to be submitted to a batch system if many new ensemble sizes are being generated, or start the job on a laptop and leave it to run overnight.

The file contains a “sparse array” of ensemble sizes. Only sizes which have an existing table are stored in the file so large ensemble sizes do not require a large jump in the size of the output file.

This program uses the random number generator to compute the correction factors. The generator is seeded with the ensemble size so repeated runs of the program will generate the same values for the tables.

6.96.2 Namelist

This namelist is read from the file input.nml. Namelists start with an ampersand ‘&’ and terminate with a slash ‘/’. Character strings that contain a ‘/’ must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&gen_sampling_error_table_nml
   ens_sizes = -1
   debug = .false.
   /
```

Description of each namelist entry

ens_sizes *type:* integer(200)

List of ensemble sizes to compute Sampling Error Correction tables for. These do not need to be in any particular order. Duplicates will be removed and any sizes which already have tables computed in the output file will be skipped. The file which comes with the system already has tables computed for these ensemble sizes:

```
ens_sizes = 5, 6, 7, 8, 9, 10, 12, 14, 15, 16, 18, 20,
            22, 24, 28, 30, 32, 36, 40, 44, 48, 49, 50,
            52, 56, 60, 64, 70, 72, 80, 84, 88, 90, 96,
            100, 120, 140, 160, 180, 200
```

debug *type:* logical

If true print out debugging info.

6.96.3 Examples

To add tables for ensemble sizes 128 and 256 run the program with this namelist:

```
&gen_sampling_error_table_nml
   ens_sizes = 128, 256,
   debug = .false.
   /
```

6.96.4 Modules used

```
types_mod
utilities_mod
random_seq_mod
netcdf
```

6.96.5 Files

- output file is always `sampling_error_corrrection_table.nc` If one exists new ensemble sizes will be appended. If it doesn't exist a new file will be created. This is a NetCDF format file.

6.96.6 References

- Ref: Anderson, J., 2012: Localization and Sampling Error Correction in Ensemble Kalman Filter Data Assimilation. Mon. Wea. Rev., 140, 2359-2371, doi: 10.1175/MWR-D-11-00013.1.

6.97 PROGRAM perturb_single_instance

6.97.1 Overview

Utility program to generate an ensemble of perturbed ensemble member restart files. This program can be run in parallel and used as a stand alone program.

6.97.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&perturb_single_instance
   ens_size               = ''
   input_files            = ''
   output_files           = ''
   output_file_list       = ''
   perturbation_amplitude = 0.0
   single_restart_file_in = .false.
  /
```

Item	Type	Description
ens_size	integer	Total number of ensemble members.
input_files	character(len=256),dimension(num_domains)	The restart file you would like to perturb from.
output_file_list	character(len=256)	A file containing a list of the desired output names.
output_files	character(len=256)	An array of filenames
perturbation_amplitude	real(r8)	The desired perturbation amplitude. If the model provides an interface then it will use that subroutine, otherwise it will simply add gaussian noise to the entire state, and this is the standard deviation.
single_restart_file_in	logical	A boolean, specifying if you have a single file restart, such as the case for lower order models.

Below is an example of a typical namelist for the perturb_single_instance.

```
&perturb_single_instance_nml
   ens_size         = 3
   input_files      = 'caminput.nc'
   output_files     = 'cam_pert1.nc','cam_pert2.nc','cam_pert3.nc'
/
```

6.97.3 Files

- inputfile.nc (description file that will be perturbed)
- output_file_list.txt (a file containing a list of restart files) and,
- perturb_single_instance.nml

6.97.4 References

- none

6.98 system simulation programs

6.98.1 Overview

A collection of standalone programs for simulating various properties of ensembles.

- `gen_sampling_err_table.f90`
- `full_error.f90`
- `obs_sampling_err.f90`
- `sampling_error.f90`
- `system_simulation.f90`
- `test_sampling_err_table.f90`
- `correl_error.f90`

The program of most interest here is ``gen_sampling_err_table.f90`` which generates the lookup table needed when using sampling error correction in ``filter``. Talk to Jeff Anderson about the other programs in this directory.

To enable the sampling error correction algorithm in `filter`, set the namelist item &assim_tools_nml : sampling_error_correction to *.true.*, and copy the netCDF file system_simulation/sampling_error_correction_table.nc into the run directory. The supported set of precomputed ensemble sizes can be found by exploring the `ens_sizes` variable in sampling_error_correction_table.nc. To add support for another ensemble size, build the executables in the work directory, (usually by running `quickbuild.csh`) set the `ens_sizes` (it takes a list, but keep it short) namelist item in `work/input.nml`, and run `gen_sampling_err_table`. It generates a LARGE number of samples *per ensemble size* for statistical rigor. Larger ensemble sizes take longer to generate, and compiler optimizations matter - perhaps significantly. For example, the numbers below come from calculating one ensemble size at a time on my desktop machine with gfortran and basic optimization:

ensemble size	run-time (seconds)
10	57
50	273
100	548

The basic structure of sampling_error_correction_table.nc is shown below.

```
0[1095] desktop:system_simulation/work % ncdump -v ens_sizes *nc
netcdf sampling_error_correction_table {
dimensions:
        bins = 200 ;
```

(continues on next page)

(continued from previous page)

```
        ens_sizes = UNLIMITED ; // (40 currently)
variables:
        int count(ens_sizes, bins) ;
                count:description = "number of samples in each bin" ;
        double true_corr_mean(ens_sizes, bins) ;
        double alpha(ens_sizes, bins) ;
                alpha:description = "sampling error correction factors" ;
        int ens_sizes(ens_sizes) ;
                ens_sizes:description = "ensemble size used for calculation" ;

// global attributes:
                :num_samples = 100000000 ;
                :title = "Sampling Error Corrections for fixed ensemble sizes." ;
                :reference = "Anderson, J., 2012: Localization and Sampling Error
                              Correction in Ensemble Kalman Filter Data Assimilation.
                              Mon. Wea. Rev., 140, 2359-2371, doi: 10.1175/MWR-D-11-
↪00013.1." ;
                :version = "" ;
data:

These ensemble sizes are already supported!
 ens_sizes = 5,  6,  7,  8,  9, 10, 12, 14, 15, 16, 18, 20, 22, 24, 28, 30, 32, 36,␣
↪40, 44,
            48, 49, 50, 52, 56, 60, 64, 70, 72, 80, 84, 88, 90, 96, 100, 120, 140,␣
↪160, 180, 200
}
```

6.98.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&gen_sampling_error_table_nml
   ens_sizes = 5,  6,  7,  8,  9, 10, 12, 14, 15, 16, 18, 20, 22, 24, 28, 30, 32, 36,␣
↪40, 44,
              48, 49, 50, 52, 56, 60, 64, 70, 72, 80, 84, 88, 90, 96, 100, 120, 140,␣
↪160, 180, 200
   debug = .false.
   /
```

Item	Type	Description
ens_sizes	integer(200)	An array of ensemble sizes to compute. Any new size gets appended to the variables in the netCDF file. Any order is fine, the array does not have to be monotonic. The numbers listed in the example exist in the file distributed with DART. *Do not get carried away by generating a lot of new ensemble sizes in one execution.* The table of run-time above should give you some indication of how long it takes to create a new entry.
debug	logical	A switch to add some run-time output. Generally not needed.

6.98.3 Modules used

```
types_mod
utilities_mod
random_seq_mod
```

- `input.nml` for the run-time input
- `sampling_error_correction_table.nc` is both read and written. Any new ensemble sizes are simply appended to the file.
- `dart_log.out` has the run-time output.
- `input.nml` for the run-time input
- `final_full.N` are created - N is the ensemble size.
- `dart_log.out` has the run-time output.

6.98.4 References

- **Anderson, J. L.**, 2012: Localization and Sampling Error Correction in Ensemble Kalman Filter Data Assimilation. *Mon. Wea. Rev.*, **140**, 2359-2371 doi: 10.1175/MWR-D-11-00013.1

6.99 PROGRAM `compute_error`

6.99.1 Overview

Utility program to compute the time-mean ensemble error and spread in the same manner that the DART MATLAB diagnostic routine 'plot_total_err' does. It runs from the command line, opens no windows, and outputs several types of numerical results on standard output. Grep for 'Total' to get the 2 lines with total error and total spread. Intended for scripts where only the numeric results are wanted instead of a time-series plot. This routine does not do any weighted computations.

The default is to compare a True_State.nc file output from perfect_model_obs to a Prior_Diag.nc file output from filter. Other filenames can be specified in the namelist. These files must have at least one overlapping value in the 'time' array. The statistics will be done on the overlapping time region only.

The output includes the min and max error and spread values, and the time index and time value where that occurs. There is also an option to recompute the time mean ensemble error and spread after skipping the first N times. This can be useful to skip an initial error spike while the model is spinning up which can result in a larger than expected total error.

Namelist interface `&compute_error_nml` is read from file `input.nml`.

6.99.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&compute_error_nml
   truth_file_name   = 'true_state.nc'
   diag_file_name    = 'preassim.nc'
   skip_first_ntimes = 0
  /
```

Item	Type	Description
truth_file_name	character(len=256)	State-space diagnostic file from the 'perfect_model_obs' program.
diag_file_name	character(len=256)	State space diagnostic file output from the 'filter' program.
skip_first_ntimes	integer	If set to a value greater than 0, the error values will be recomputed a second time, skipping the first N times. This can be useful when running an experiment that has an initial error spike as the model spins up and then decays down to a more steady state.

6.99.3 Modules used

```
types_mod
utilities_mod
```

6.99.4 Files

- DART diagnosic files (True_State.nc, Prior_Diag.nc)
- compute_error.nml

6.99.5 References

- none

6.100 PROGRAM preprocess

6.100.1 Overview

Preprocess is a DART-supplied preprocessor program. Preprocess is used to insert observation specific code into DART at compile time.

In DART, forward operators are not specific to any one model. To achieve this separation between models and forward operators DART makes a distinction between an observation *type* and a physical *quantity*. For example, a radiosonde used to measure windspeed would be a *type* of observation. Zonal wind and meridional wind are *quantities* used to calculate windspeed. Specifying many observation types allows DART to be able to evaluate some observations and assimilate others even if the instruments measure the same quantity.

Preprocess takes user supplied observation and quantity files and combines them with template files to produce code for DART. Use the namelist option 'obs_type_files' to specify the input observation files and the namelist option 'quantity_files' to specify the input quantity files.

- If no quantity files are given, a default list of quantities is used.
- If no obs_type_files are given, only identity observations can be used in the filter (i.e. the state variable values are directly observed; forward operator is an identity)

The template files `DEFAULT_obs_def_mod.F90` and `DEFAULT_obs_kind_mod.F90` contain specially formatted comment lines. These comment lines are used as markers to insert observation specific information. Prepreocess relies these comment lines being used *verbatim*.

There is no need to to alter `DEFAULT_obs_def_mod.F90` or `DEFAULT_obs_kind_mod.F90`. Detailed instructions for adding new observation types can be found in *MODULE obs_def_mod*. New quantities should be added to a quantity file, for example a new atmosphere quantity should be added to `atmosphere_quantities_mod.f90`.

Every line in a quantity file between the start and end markers must be a comment or a quantity definition (QTY_string). Multiple name-value pairs can be specified for a quantity but are not required. For example, temperature may be defined: `!   QTY_TEMPERATURE units="K" minval=0.0`. Comments are allowed between quantity definitions or on the same line as the definition. The code snippet below shows acceptable formats for quantity definitions

```
! BEGIN DART PREPROCESS QUANTITY DEFINITIONS ! ! Formats accepted:  !  !
QTY_string ! QTY_string name=value ! QTY_string name=value name2=value2
! ! QTY_string ! comments ! ! ! comment ! ! END DART PREPROCESS QUANTITY
DEFINITIONS
```

The output files produced by preprocess are named
`assimilation_code/modules/observations/obs_kind_mod.f90` and
`observations/forward_operators/obs_def_mod.f90`, but can be renamed by namelist control if needed. Be aware that if you change the name of these output files, you will need to change the path_names files for DART executables.

6.100.2 Namelist

When you run preprocess, the namelist is read from the file `input.nml` in the directory where preprocess is run.

Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&preprocess_nml
  overwrite_output        = .true.,
  input_obs_def_mod_file  = '../../../observations/forward_operators/DEFAULT_obs_def_
↪mod.F90',
  output_obs_def_mod_file = '../../../observations/forward_operators/obs_def_mod.f90',
  input_obs_qty_mod_file  = '../../../assimilation_code/modules/observations/DEFAULT_
↪obs_kind_mod.F90',
  output_obs_qty_mod_file = '../../../assimilation_code/modules/observations/obs_kind_
↪mod.f90',
  quantity_files          = '../../../assimilation_code/modules/observations/
↪atmosphere_quantities_mod.f90',
  obs_type_files          = '../../../observations/forward_operators/obs_def_
↪reanalysis_bufr_mod.f90',
                            '../../../observations/forward_operators/obs_def_rel_
↪humidity_mod.f90',
                            '../../../observations/forward_operators/obs_def_
↪altimeter_mod.f90'
 /
```

Item	Type	Description
in-put_obs_def_mod_file	character(len=256)	Path name of the template obs def module to be preprocessed. The default is `../../../observations/forward _operators/DEFAULT_obs_def_mod.F90`. This file must have the appropriate commented lines indicating where the different parts of the input special obs definition modules are to be inserted.
out-put_obs_def_mod_file	character(len=256)	Path name of output obs def module to be created by preprocess. The default is `../../../observations /forward_operators/obs_def_mod.f90`.
in-put_obs_qty_mod_file	character(len=256)	Path name of input obs quantity file to be preprocessed. The default path name is `. ./../../assimilation_code/modules/obs ervations/DEFAULT_obs_kind_mod.F90`. This file must have the appropriate commented lines indicating where the different quantity modules are to be inserted.
out-put_obs_qty_mod_file	character(len=256)	Path name of output obs quantity module to be created by preprocess. The default is `../../../assimilation_code/mod ules/observations/obs_kind_mod.f90`.
obs_type_files	character(len=256)(:)	A list of files containing observation definitions for the type of observations you want to use with DART. The maximum number of files is limited to MAX_OBS_TYPE_FILES = 1000. The DART obs_def files are in `observations/f orward_operators/obs_def_*.mod.f90`.
over-write_output	logical	By defualt, preprocess will overwrite the existing obs_kind_mod.f90 and obs_def_mod.f90 files. Set overwrite_output = .false. if you want to preprocess to not overwrite existing files.

6.100.3 Modules used

```
parse_arges_mod
types_mod
utilities_mod
```

Namelist interface `&preprocess_nml` must be read from file `input.nml`.

6.100.4 Files

- input_obs_def_mod_file, specified by namelist; usually `DEFAULT_obs_def_mod.F90`.
- output_obs_def_mod_file, specified by namelist; usually `obs_def_mod.f90`.
- input_obs_qty_mod_file, specified by namelist; usually `DEFAULT_obs_kind_mod.F90`.
- output_obs_qty_mod_file, specified by namelist; usually `obs_kind_mod.f90`.
- obs_type_files, specified by namelist; usually files like `obs_def_reanalysis_bufr_mod.f90`.
- quantity_files, specified by namelist; usually files like `atmosphere_quantities_mod.f90`.
- namelistfile

6.100.5 References

- none

6.101 PROGRAM `obs_impact_tool`

6.101.1 Overview

The standard DART algorithms compute increments for an observation and then compute corresponding increments for each model state variable due to that observation. To do this, DART computes a sample regression coefficient using the prior ensemble distributions of a state variable and the observation. The increments for each member of the observation are multiplied by this regression coefficient and then added to the corresponding prior ensemble member for the state variable. However, in many cases, it is appropriate to reduce the impact of an observation on a state variable; this is called localization. The standard DART algorithms allow users to specify a localization that is a function of the horizontal (and optionally vertical) distance between the observation and the state variable. The localization is a value between 0 and 1 and multiplies the regression coefficient when updating state ensemble members.

Sometimes, it may be desirable to do an additional localization that is a function of the type of observation and the state vector quantity. This program allows users to construct a table that is read by filter at run-time to localize the impact of sets of observation types on sets of state vectorquantities. Users can create named sets of observation types and sets of state vector quantities and specify a localization for the impact of the specified observation types on the state vector quantities.

An example would be to create a subset of observations of tracer concentration for a variety of tracers, and a subset of dynamic state variable quantities like temperatures and wind components. It has been common to set this localization value to 0 so that tracer observations have no impact on dynamic state quantities, however, the tool allows values between 0 and 1 to be specified.

This tool allows related collections of observation types and state vector quantities to be named and then express the relationship of the named groups to each other in a concise way. It can also define relationships by exceptions.

All the listed observation types and state vector quantities must be known by the system. If they are not, look at the &preprocess_nml :: input_items namelist which specifies which obs_def_xxx_mod.f90 files are included, which is where observation types are defined. Quantities are defined in `assimilation_code/modules/observations/DEFAULT_obs_kinds_mod.F90`. (Note you must add new quantities in 2 places if you do alter this file.)

Format of the input file can be any combination of these types of sections:

```
# hash mark starts a comment.

# the GROUP keyword starts a group and must be followed
# by a name.  All types or quantities listed before the END
# line becomes members of this group.

# GROUPs cannot contain nested groups.

GROUP groupname1
 QTY_xxx  QTY_xxx  QTY_xxx
 QTY_xxx                          # comments can be here
END GROUP

GROUP groupname2
 QTY_xxx
 QTY_xxx
 QTY_xxx
 QTY_xxx
END GROUP

# GROUPs can also be defined by specifying ALL, ALLQTYS,
# or ALLTYPES and then EXCEPT and listing the types or
# quantities which should be removed from this group.
# ALL EXCEPT must be the first line in a group, and all
# subsequent items are removed from the list.
# The items listed after EXCEPT can include the names
# of other groups.

GROUP groupnameM
ALL EXCEPT QTY_xxx QTY_xxx
QTY_xxx
END GROUP

GROUP groupnameN
ALL EXCEPT groupnameY
END GROUP


# once any groups have been defined, a single instance
# of the IMPACT table is specified by listing a TYPE,
# QTY, or group in column 1, then a QTY or GROUP
# in column 2 (the second name cannot be a specific type).
# column 3 must be 0.0 or 1.0.  subsequent entries
# that overlap previous entries have precedence
# (last entry wins).

IMPACT
 QTY_xxx     QTY_xxx      0.0
 QTY_xxx     groupname1   0.0
 groupname1  QTY_xxx      0.0
```

(continues on next page)

(continued from previous page)

```
 groupname1  groupname1   0.0
END IMPACT
```

Namelist interface `&obs_impact_tool_nml` must be read from file `input.nml`.

6.101.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_impact_tool_nml
  input_filename          = 'cross_correlations.txt'
  output_filename         = 'control_impact_runtime.txt'
  debug                   = .false.
  /
```

Item	Type	Description
input_filename	character(len=512)	Name of an ascii text file which describes how the interaction of observations to state vector values and observations to other observations should be controlled. See the Overview section for details about the format of the input file entries.
output_filename	character(len=512)	Name of an ascii text file which created by this tool. It can be read at filter run time to control the impact of observations on state vector items and other observation values. The format of this file is set by this tool and should not be modified by hand. Rerun this tool to recreate the file.
debug	logical	If true print out debugging info.

6.101.3 Examples

To prevent chemistry species from impacting the meterological variables in the model state, and vice versa:

```
GROUP chem
 QTY_CO QTY_NO QTY_C2H4
END GROUP

GROUP met
 ALLQTYS EXCEPT chem
END GROUP

IMPACT
 chem   met    0.0
 met    chem   0.0
END IMPACT
```

6.101.4 Modules used

```
types_mod
utilities_mod
parse_args_mod
```

6.101.5 Files

- two text files, one input and one output.
- obs_impact_tool.nml

6.101.6 References

- none

6.102 program `create_fixed_network_seq`

6.102.1 Overview

Reads in an observation sequence file and creates a second observation sequence file. Any time information in the input file is ignored entirely. All of the observations in the input file define a set of observations. The output sequence replicates this set multiple times, either with a fixed period in time or at arbitrarily selected times. The program is driven by input from standard input, either the terminal or a text file.

First, one must select either a regularly repeating time sequence of observations (option 1) or an arbitrarily repeating sequence (option 2). For the fixed period, the total number of observation times, the first observation time and the period of the observations is input and an output observation sequence is generated. For the arbitrary period, the user is queried for the number of observing times and then a set of monotonically increasing times. Finally, the user selects a file name (traditionally obs_seq.in) to which the output file is written. The format of the output file is controlled by the namelist options in obs_sequence_mod.

Any data values or quality control flags associated with the input set are replicated to the output, but this program is typically used with perfect model experiments to create observations without data, which are then filled in by running *program perfect_model_obs*.

6.102.2 Modules used

```
types_mod
utilities_mod
obs_def_mod
obs_sequence_mod
time_manager_mod
model_mod
```

6.102.3 Files

- Input observation sequence (set_def.out is standard).
- Output observation sequence (obs_seq.in is standard).

6.102.4 References

- none

6.103 program `obs_loop`

6.103.1 Overview

This program is a template that is intended to be modified by the user to do any desired operations on an observation sequence file.

6.103.2 Usage

This program is intended to be used as a template to read in observations from one obs_seq file and write them, optionally modified in some way, to another obs_seq file. It can be compiled and run as-is, but it simply makes an exact copy of the input file.

There are comments in the code (search for `MODIFY HERE`) where you can test values, types, times, error values, and either modify them or skip copying that observation to the output.

There are build files in `observations/utilities/oned` and `observations/utilities/threed_sphere` to build the `obs_loop` program.

6.103.3 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_loop_nml
   filename_in  = ''
   filename_out = ''
   print_only   = .false.
   calendar     = 'Gregorian'
   /
```

Items in this namelist set the input and output files.

Item	Type	Description
filename_in	character(len=256)	Observation sequence file to read
filename_out	character(len=256)	Observation sequence file to create and write. If this file exists it will be overwritten.
print_only	logical	If .TRUE. then do the work but only print out information about what would be written as output without actually creating the output file.
calendar	character(len=32)	The string name of a valid DART calendar type. (See the *MODULE time_manager_mod* documentation for a list of valid types.) The setting here does not change what is written to the output file; it only changes how the date information is printed to the screen in the informational messages.

6.103.4 Discussion

See the documentation in the obs_kind and obs_def modules for things you can query about an observation, and how to set (overwrite) existing values.

6.103.5 Building

There are build files in `observations/utilities/oned` and `observations/utilities/threed_sphere` to build the `obs_loop` program.

The `preprocess` program must be built and run first to define what set of observation types will be supported. See the *PROGRAM preprocess* for more details on how to define the list and run it. The `&preprocess_nml` namelist in the `input.nml` file must contain files with definitions for the combined set of all observation types which will be encountered over all input obs_seq files.

If you have observation types which are not part of the default list in the &preprocess_nml namelist, add them to the input.nml file and then either run quickbuild.csh or make and run preprocess and then make the obs_loop tool.

Usually the directories where executables are built will include a `quickbuild.csh` script which builds and runs preprocess and then builds the rest of the executables by executing all files with names starting with `mkmf_`.

6.103.6 Files

filename	purpose
input.nml	to read the &obs_loop_nml namelist

6.103.7 References

1. none

6.103.8 Error codes and conditions

Routine	Message	Comment
obs_loop		
obs_loop		

6.103.9 Future plans

none

6.104 program perfect_model_obs

6.104.1 Overview

Main program for creating synthetic observation sequences given a model for use in filter assimilations. Reads in an observation sequence file which has only observation definitions and generates synthetic observation values for an output observation sequence file. The execution of perfect_model_obs is controlled by the input observation sequence file and the model time-stepping capabilities in a manner analogous to that used by the filter program.

6.104.2 Namelist

This namelist is read from the file input.nml. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&perfect_model_obs_nml
   single_file_in              = .false.,
   read_input_state_from_file = .false.,
   input_state_files           = "",
   init_time_days              = 0,
   init_time_seconds           = 0,

   single_file_out             = .false.,
   output_state_files          = "",
   write_output_state_to_file = .false.,
   output_interval             = 1,

   distributed_state           = .false.,
   async                       = 0,
   adv_ens_command             = "./advance_model.csh",
   tasks_per_model_advance     = 1,

   obs_seq_in_file_name        = "obs_seq.in",
   obs_seq_out_file_name       = "obs_seq.out",
   first_obs_days              = -1,
```

(continues on next page)

(continued from previous page)

```
   first_obs_seconds           = -1,
   last_obs_days               = -1,
   last_obs_seconds            = -1,
   obs_window_days             = -1,
   obs_window_seconds          = -1,

   trace_execution             = .false.,
   output_timestamps           = .false.,
   print_every_nth_obs         = 0,
   output_forward_op_errors    = .false.,
   silence                     = .false.,
/
```

Item	Type	Description
read_input_state_from_file	logical	If false, model_mod must provide the input state.
single_file_in	logical	Get all states from a single file.
input_state_files	character(len=256) dimension(MAX_NUM_DOMS)	A list of files, one per domain. Each file must be a text file containing the name of the NetCDF file to open.
write_output_state_to_file	logical	If false, state is not written out.
single_file_out	logical	Write all states to a single file.
output_state_files	character(len=256) dimension(MAX_NUM_DOMS)	A list of files, one per domain. Each file must be a text file containing the names of the NetCDF file to open.
init_time_days	integer	If negative, don't use. If non-negative, override the initial data time read from restart file.
init_time_seconds	integer	If negative don't use. If non-negative, override the initial data time read from restart file.
output_interval	integer	Output state and observation diagnostics every nth assimilation time, n is output_interval.
distributed_state	logical	True means the ensemble data is distributed across all tasks as it is read in, so a single task never has to have enough memory to store the data for an ensemble member. Large models should always set this to .true., while for small models it may be faster to set this to .false.
async	integer	Controls method for advancing model: • 0 = subroutine call • 2 = shell command, single task model • 4 = shell command, parallel model
adv_ens_command	character(len=129)	Command sent to shell if async == 2 or 4.
tasks_per_model_advance	integer	Number of tasks to use while advancing the model.
obs_seq_in_file_name	character(len=256)	File name from which to read an observation sequence.
obs_seq_out_file_name	character(len=256)	File name to which to write output observation sequence.
first_obs_days	integer	If negative, don't use. If non-negative, ignore any observations before this time.
first_obs_seconds	integer	If negative, don't use. If non-negative, ignore any observations before this time.
last_obs_days	integer	If negative, don't use. If non-negative, ignore any observations after this time.
last_obs_seconds	integer	If negative, don't use. If non-negative, ignore any observations after this time.
obs_window_days	integer	If negative, don't use. If non-negative, reserved for future use.

6.104.3 Modules used

```
types_mod
utilities_mod
time_manager_mod
obs_sequence_mod
obs_def_mod
obs_model_mod
assim_model_mod
mpi_utilities_mod
random_seq_mod
ensemble_manager_mod
```

6.104.4 Files

- observation sequence input file; name comes from obs_seq_in_file_name
- observation sequence output file; name comes from obs_seq_out_file_name
- input state vector file; name comes from restart_in_file_name
- output state vector file; name comes from restart_out_file_name
- perfect_model_mod.nml in input.nml

6.104.5 References

- none

6.105 program `obs_selection`

6.105.1 Overview

This specialized tool selects a subset of input observations from an observation sequence file. For a more general purpose observation sequence file tool, see the *program obs_sequence_tool*. This tool takes a selected list of observation types, times, and locations, and extracts only the matching observations out of one or more obs_sequence files. The tool which creates the input selection file is usually *program obs_seq_coverage*. Alternatively, the selection file can be a full observation sequence file, in which case the types, times, and locations of those observations are used as the selection criteria.

This tool processes each observation sequence file listed in the input namelist `filename_seq` or `filename_seq_list`. If the observation type, time and location matches an entry in the selection file, it is copied through to the output. Otherwise it is ignored.

The actions of the `obs_selection` program are controlled by a Fortran namelist, read from a file named `input.nml` in the current directory. A detailed description of each namelist item is described in the namelist section of this document. The names used in this discussion refer to these namelist items.

6.105.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand ‘&’ and terminate with a slash ‘/’. Character strings that contain a ‘/’ must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_selection_nml
   filename_seq          = ''
   filename_seq_list     = ''
   filename_out          = 'obs_seq.processed'
   num_input_files       = 0
   selections_file       = 'obsdef_mask.txt'
   selections_is_obs_seq = .false.
   latlon_tolerance      = 0.000001
   match_vertical        = .false.
   surface_tolerance     = 0.0001
   pressure_tolerance    = 0.001
   height_tolerance      = 0.0001
   scaleheight_tolerance = 0.001
   level_tolerance       = 0.00001
   print_only            = .false.
   partial_write         = .false.
   print_timestamps      = .false.
   calendar              = 'Gregorian'
  /
```

Item	Type	Description
filename_seq	character(len=256), dimension(500)	The array of names of the observation sequence files to process, up to a max count of 500 files. (Specify only the actual number of input files. It is not necessary to specify 500 entries.)
filename_seq_list	character(len=256)	An alternative way to specify the list of input files. The name of a text file which contains, one per line, the names of the observation sequence files to process. You can only specify one of filename_seq OR filename_seq_list, not both.
num_input_files	integer	Optional. The number of observation sequence files to process. Maximum of 500. If 0, the length is set by the number of input files given. If non-zero, must match the given input file list length. (Can be used to verify the right number of input files were processed.)
filename_out	character(len=256)	The name of the resulting output observation sequence file. There is only a single output file from this tool. If the input specifies multiple obs_seq input files, the results are concatinated into a single output file.
selections_file	character(len=256)	The name of the input file containing the mask of observation definitions (the textfile output of *program obs_seq_coverage*). Alternatively, this can be the name of a full observation sequence file. In this case, the types, times, and locations are extracted from this file and then used in the same manner as a mask file from the coverage tool.
selections_is_obs_seq	logical	If .TRUE. the filename given for the “selections_file” is a full obs_sequence file and not a text file from the coverage tool.
latlon_tolerance	real(r8)	Specified in degrees. For observations to match in the horizontal the difference in degrees for each of latitude and longitude must be less than this threshold. If less than or equal to 0, the values must match exactly.
match_vertical	logical	If .TRUE. the locations of the observations in the input files have to match the selection list not only the horizontal but also in the vertical.
surface_tolerance	real(r8)	Specified in meters. If “match_vertical” is .FALSE. this value is ignored. If “match_vertical” is .TRUE., this applies to observations with a vertical type of VERTISSURFACE. For observations which match in the horizontal, the vertical surface elevation difference must be less than this to be considered the same.
pressure_tolerance	real(r8)	Specified in pascals. If “match_vertical” is .FALSE. this value is ignored. If “match_vertical” is .TRUE., this applies to observations with a vertical type of VERTISPRESSURE. For observations which match in the horizontal, the vertical difference must be less than this to be considered the same.
height_tolerance	real(r8)	Specified in meters. If “match_vertical” is .FALSE. this value is ignored. If “match_vertical” is .TRUE., this applies to observations with a vertical type of VERTISHEIGHT. For observations which match in the horizontal, the vertical difference must be less than this to be considered the same.
scaleheight_tolerance	real(r8)	Specified in unitless values. If “match_vertical” is .FALSE. this value is ignored. If “match_vertical” is .TRUE., this applies to observations with a vertical type of VERTISSCALEHEIGHT. For observations which match in the horizontal, the vertical difference must be less than this to be considered the same.
level_tolerance	real(r8)	Specified in fractional model levels. If “match_vertical” is .FALSE. this value is ignored. If “match_vertical” is .TRUE., this applies to observations with a vertical type of VERTISLEVEL. For observations which match in the horizontal, the vertical difference must be less than this to be considered the same. Note that some models only support integer level values, but others support fractional levels. The vertical value in an observation is a floating point/real value, so fractional levels are possible to specify for an observation.
print_only	logical	If .TRUE. do not create an output file, but print a summary of the number and types of each observation in each input file, and then the number of observations and types which would have been created in an output file.
partial_write	logical	Generally only used for debugging problems. After each input obs_seq file is processed, this flag, if .TRUE., causes the code to write out the partial results to the output file. The default is to process all input files (if more than a single file is specified) and write the output file only at the end of the processing.
print_timestamps	logical	Generally only used for debugging very slow execution runs. This flag, if .TRUE., causes the code to output timestamps (wall clock time) at various locations during the processing phases. It may

6.105.3 Building

Most $DART/models/*/work directories contain files needed to build this tool along with the other executable programs. It is also possible to build this tool in the $DART/observations/utilities directory. In either case the preprocess program must be built and run first to define what set of observation types will be supported. See the *PROGRAM preprocess* for more details on how to define the list and run it. The &preprocess_nml namelist in the input.nml file must contain files with definitions for the combined set of all observation types which will be encountered over all input obs_seq files. The other important choice when building the tool is to include a compatible locations module in the path_names_obs_selection file. For the low-order models the oned module should be used; for real-world observations the threed_sphere module should be used.

Usually the directories where executables are built will include a quickbuild.csh script which builds and runs preprocess and then builds the rest of the executables by executing all files with names starting with mkmf_. If the obs_selection tool is not built because there is no mkmf_obs_selection and path_names_obs_selection file in the current directory they can be copied from another model. The path_names_obs_selection file will need to be edited to be consistent with the model you are building.

6.105.4 Modules used

```
types_mod
utilities_mod
time_manager_mod
obs_def_mod
obs_sequence_mod
```

6.105.5 Files

- input.nml
- The input files specified in the filename_seq namelist variable.
- The output file specified in the filename_out namelist variable.

6.105.6 References

- none

6.106 program obs_sequence_tool

6.106.1 Overview

DART observation sequence files are stored in a proprietary format. This tool makes it easier to manipulate these files, allowing the user to subset or combine one or more files into a single output file.

The tool has many options to select subsets of observations by time, type, data value, and location. The tool also allows the contents of observations to be changed by subsetting and/or reordering the copies and qc entries. Files with equivalent data but with different metadata labels (e.g. ‘NCEP QC’ vs. ‘QC’) can now be merged as well. The tool can be run without creating an output file, only printing a summary of the counts of each observation type in the input files, and it can be used to convert from binary to ASCII and back.

The actions of the `obs_sequence_tool` program are controlled by a Fortran namelist, read from a file named `input.nml` in the current directory. A detailed description of each namelist item is described in the namelist section below.

The examples section of this document below has extensive examples of common usages for this tool. Below that are more details about DART observation sequence files, the structure of individual observations, and general background information.

6.106.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_sequence_tool_nml
   filename_seq         = ''
   filename_seq_list    = ''
   filename_out         = 'obs_seq.processed'
   first_obs_days       = -1
   first_obs_seconds    = -1
   last_obs_days        = -1
   last_obs_seconds     = -1
   obs_types            = ''
   keep_types           = .true.
   min_box              = -888888.0
   max_box              = -888888.0
   min_lat              =  -90.0
   max_lat              =   90.0
   min_lon              =    0.0
   max_lon              =  360.0
   copy_metadata        = ''
   min_copy             = -888888.0
   max_copy             = -888888.0
   copy_type            = ''
   edit_copy_metadata   = .false.
   new_copy_metadata    = ''
   edit_copies          = .false.
   new_copy_index       = -1
   new_copy_data        = -888888.0
   qc_metadata          = ''
   min_qc               = -888888.0
   max_qc               = -888888.0
   edit_qc_metadata     = .false.
   new_qc_metadata      = ''
   edit_qcs             = .false.
   new_qc_index         = -1
   new_qc_data          = -888888.0
   synonymous_copy_list = ''
   synonymous_qc_list   = ''
   print_only           = .false.
   gregorian_cal        = .true.
   min_gps_height       = -888888.0
   /
```

Item	Type	Description
filename_seq	character(len=256), dimension(1000)	The array of names of the observation sequence files to process. (W
filename_seq_list	character(len=256)	The name of a text file which contains, one per line, the names of th
filename_out	character(len=256)	The name of the resulting output observation sequence file.
first_obs_days	integer	If non-negative, restrict the timestamps of the observations copied t
first_obs_seconds	integer	If non-negative, restrict the timestamps of the observations copied t
last_obs_days	integer	If non-negative, restrict the timestamps of the observations copied t
last_obs_seconds	integer	If non-negative, restrict the timestamps of the observations copied t
obs_types	character(len=32), dimension(500)	The array of observation type names to process. If any names speci
keep_types	logical	Ignored unless one or more observation types are specified in the o
min_box	real(r8)(:)	If the locations are 1D, set a min value here instead of using the lat/
max_box	real(r8)(:)	If the locations are 1D, set a max value here instead of using the lat
min_lat	real(r8)	If specified, the minimum latitude, in degrees, of observations to be
max_lat	real(r8)	If specified, the maximum latitude, in degrees, of observations to be
min_lon	real(r8)	If specified, the minimum longitude, in degrees, of observations to
max_lon	real(r8)	If specified, the maximum longitude, in degrees, of observations to
copy_metadata	character	If specified, the metadata string describing one of the data copy fiel
min_copy	real	If specified, the minimum value in the data copy field matching the
max_copy	real	If specified, the maximum value in the data copy field matching the
copy_type	character(len=32)	If specified, the string name of an observation type to be copied to t
edit_copy_metadata	logical	If true, replace the output file metadata strings with the list specifie
new_copy_metadata	character(len=*)(:)	List of new metadata strings. Use with care, there is no error checki
edit_copies	logical	If true, subset or rearrange the actual data copies in the output. The
new_copy_index	integer(:)	An array of integers, which control how copies in the input are mov
new_copy_data	real(:)	An array of reals. The length should correspond to the number of 0s
qc_metadata	character	If specified, the metadata string describing one of the quality contro
min_qc	real	If specified, the minimum qc value in the QC field matching the qc_
max_qc	real	If specified, the maximum qc value in the QC field matching the qc
edit_qc_metadata	logical	If true, replace the output file metadata strings with the list specifie
new_qc_metadata	character(len=*)(:)	List of new metadata strings. Use with care, there is no error checki
edit_qcs	logical	If true, subset or rearrange the actual data QCs in the output. The ne
new_qc_index	integer(:)	An array of integers, which control how QCs in the input are moved
new_qc_data	real(:)	An array of reals. The length should correspond to the number of 0s
synonymous_copy_list	character(len=*)(:)	An array of strings which are to be considered synonyms in the cop
synonymous_qc_list	character(len=*)(:)	An array of strings which are to be considered synonyms in the qc n
print_only	logical	If .TRUE., do not create an output file, but print a summary of the n
gregorian_cal	logical	If .true. the dates of the first and last observations in each file will b
num_input_files	integer	DEPRECATED. The number of observation sequence files to proce

6.106.3 Examples

Here are details on how to set up common cases using this tool:

- Merge multiple files
- Subset in Time
- Subset by Observation Type
- Subset by Location
- Binary to ASCII and back
- Merging files with incompatible Metadata
- Altering the number of Copies or QC values
- Printing only
- Subset by Observation or QC Value

Merge multiple files

Either specify a list of input files for `filename_seq`, like:

```
&obs_sequence_tool_nml
   filename_seq       = 'obs_seq20071101',
                        'qscatL2B_2007_11_01a.out',
                        'obs_seq.gpsro_2007110106',
   filename_out       = 'obs_seq20071101.all',
   gregorian_cal      = .true.
/
```

and all observations in each of the three input files will be merged in time order and output in a single observation sequence file. Or from the command line create a file containing one filename per line, either with ‘ls’:

```
ls obs_seq_in* > tlist
```

or with a text editor, or any other tool of your choice. Then,

```
&obs_sequence_tool_nml
   filename_seq_list = 'tlist',
   filename_out       = 'obs_seq20071101.all',
   gregorian_cal      = .true.
/
```

will open ‘tlist’ and read the filenames, one per line, and merge them together. The output file will be named ‘obs_seq20071101.all’. Note that the filenames inside the list file should not have delimiters (e.g. single or double quotes) around the filenames.

Subset in time

The observations copied to the output file can be restricted in time by setting the namelist items for the first and last observation timestamps (in days and seconds). It is not an error for some of the input files to have no observations in the requested time range, and multiple input files can have overlapping time ranges. For example:

```
&obs_sequence_tool_nml
   filename_seq       = 'obs_seq20071101',
                        'qscatL2B_2007_11_01a.out',
                        'obs_seq.gpsro_2007110106',
   filename_out       = 'obs_seq20071101.06hrs',
   first_obs_days     = 148592,
   first_obs_seconds  =  10801,
   last_obs_days      = 148592,
   last_obs_seconds   =  32400,
   gregorian_cal      = .true.
/
```

The time range is inclusive on both ends; observations with times equal to the boundary times will be copied to the output. To split a single input file up into proper subsets (no replicated observations), the first time of the following output sequence should be +1 second from the last time of the previous output sequence. If the goal is to match an observation sequence file with an assimilation window during the execution of the `filter` program, the windows should be centered around the assimilation time starting at minus 1/2 the window time plus 1 second, and ending at exactly plus 1/2 the window time.

Subset by observation type

You specify a list of observation types, by string name, and then specify a logical value to say whether this is the list of observations to keep, or if it's the list of observations to discard. For example,

```
&obs_sequence_tool_nml
   filename_seq       = 'obs_seq20071101.06hrs',
   filename_out       = 'obs_seq20071101.wind',
   obs_types          = 'RADIOSONDE_U_WIND_COMPONENT',
                        'RADIOSONDE_V_WIND_COMPONENT',
   keep_types         = .true.,
   gregorian_cal      = .true.
/
```

will create an output file which contains only the U and V wind observations from the given input file.

```
&obs_sequence_tool_nml
   filename_seq       = 'obs_seq20071101.06hrs',
   filename_out       = 'obs_seq20071101.notemp',
   obs_types          = 'RADIOSONDE_TEMPERATURE',
   keep_types         = .false.,
   gregorian_cal      = .true.
/
```

will strip out all the radiosonde temperature observations and leave everything else.

Subset by location

If the observations have locations specified in 3 dimensions, as latitude, longitude, and a vertical coordinate, then it can be subset by specifying the corners of a lat, lon box. There is currently no vertical subsetting option. For example:

```
min_lat             =    0.0,
max_lat             =   20.0,
min_lon             =  230.0,
max_lon             =  260.0,
```

will only output observations between 0 and 20 latitude and 230 to 260 in longitude. Latitude ranges are 90 to 90, longitude can either be specified from 180 to +180, or 0 to 360.

If the observations have 1 dimensional locations, between 0 and 1, then a bounding box can be specified like:

```
min_box = 0.2,
max_box = 0.4,
```

will keep only those observations between 0.2 and 0.4. In all these tests, points on the boundaries are considered inside the box.

Binary to ASCII and back

To convert a (more compact) binary observation sequence file to a (human readable and portable) ASCII file, a single input and single output file can be specified with no selection criteria. The output file format is specified by the `write_binary_obs_sequence` item in the `&obs_sequence_nml` namelist in the `input.nml` file. It is a Fortran logical; setting it to `.TRUE.` will write a binary file, setting it to `.FALSE.` will write an ASCII text file. If you have a binary file, it must be converted on the same kind of platform as it was created on before being moved to another architecture. At this point in time, there are only 2 remaining incompatible platforms: IBM systems based on PowerPC chips, and everything else (which is Intel or AMD).

Any number of input files and selection options can be specified, as well, but for a simple conversion, leave all other input namelist items unset.

Merging files with incompatible metadata

To merge files which have the same number of copies and qc but different labels for what is exactly the same data, you can specify a list of synonym strings that will pass the matching test. For example:

```
&obs_sequence_tool_nml
   filename_seq       = 'qscatL2B_2007_11_01.out',
                        'obs_seq20071101',
                        'obs_seq.gpsro_2007110124',
   filename_out       = 'obs_seq20071101.all',
   gregorian_cal      = .true.
   synonymous_copy_list = 'NCEP BUFR observation', 'AIRS observation', 'observation',
   synonymous_qc_list   = 'NCEP QC index', 'AIRS QC', 'QC flag - wvc quality flag',
↪'QC',
/
```

will allow any copy listed to match any other copy on that list, and same with the QC values. If the output metadata strings are not specified (see below), then the actual metadata strings from the first file which is used will set the output metadata strings.

To rename or override, with care, existing metadata strings in a file, set the appropriate edit strings to true, and set the same number of copies and/or QC values as will be in the output file. Note that this will replace, without warning, whatever is originally listed as metadata. You can really mangle things here, so use this with caution:

```
&obs_sequence_tool_nml
   filename_seq       = 'qscat_all_qc_305.out', 'qscat_all_qc_306.out',
   filename_out       = 'qscat_1_qc_2007_11.out',
   edit_copy_metadata = .true.,
   new_copy_metadata  = 'observation',
   edit_qc_metadata   = .true.,
   new_qc_metadata    = 'QC', 'DART quality control',
   gregorian_cal      = .true.
/
```

The log file will print out what input strings are being replaced; check this carefully to be sure you are doing what you expect.

If you use both a synonym list and the edit list, the output file will have the specified edit list strings for metadata.

Altering the number of copies or QC values

To delete some of the copies or QC values in each observation, specify the copy or QC index numbers which are to be passed through, and list them in the exact order they should appear in the output:

```
edit_copies = .true.,
new_copy_index = 1, 2, 81, 82,

edit_qcs = .true.,
new_qc_index = 2,
```

This will create an output sequence file with only 4 copies; the original first and second copies, and copies 81 and 82. The original metadata will be retained. It will have only the second QC value from the original file.

If you are editing the copies or QCs and also specifying new metadata strings, use the number and order appropriate to the output file regardless of how many copies or QC values there were in the original input files.

You can use these index lists to reorder copies or QC values by specifying the same number of index values as currently exist but list them in a different order. Index values can be repeated multiple times in a list. This will duplicate both the metadata string as well as the data values for the copy or QC.

To delete all copies or QCs specify -1 as the first (only) entry in the new index list.

```
edit_qcs = .true.,
new_qc_index = -1,
```

To add copies or QCs, use 0 as the index value.

```
edit_copies = .true.,
new_copy_index = 1, 2, 0, 81, 82, 0
new_copy_data = 3.0, 8.0,

edit_qcs = .true.,
new_qc_index = 2, 1, 3, 0,
new_qc_data = 1.0,
```

This will insert 2 new copies in each observation and give them values of 3.0 and 8.0 in all observations. There is no way to insert a different value on a per-obs basis. This example will also reorder the 3 existing QC values and then add 1 new QC value of 1 in all observations. The 'edit_copy_metadata' and 'edit_qc_metadata' flags with the 'new_copy_metadata' and 'new_qc_metadata' lists can be used to set the metadata names of the new copies and QCs.

```
edit_copies = .true.,
new_copy_index = 1, 0, 2, 0,
new_copy_data = 3.0, 8.0,
edit_copy_metadata = .true.,
new_copy_metadata = 'observation', 'new copy 1',
                    'truth',       'new copy 2',

edit_qcs = .true.,
new_qc_index = 0, 2,
new_qc_data = 0.0,
edit_qc_metadata = .true.,
new_qc_metadata = 'dummy QC', 'DART QC',
```

To remove an existing QC value and add a QC value of 0 for all observations, run with:

```
edit_qcs = .true.,
new_qc_index = 0,
new_qc_data = 0.0,
edit_qc_metadata = .true.,
new_qc_metadata = 'dummy QC',
```

to add a constant QC of 0 for all observations, with a metadata label of 'dummy QC'.

It would be useful to allow copies or QCs from one file to be combined, obs by obs, with those from another file. However, it isn't easy to figure out how to ensure the observations in multiple files are in exactly the same order so data from the same obs are being combined. Also how to specify what should be combined is a bit complicated. So this functionality is NOT available in this tool.

Printing only

Note that you can set all the other options and then set print true, and it will do all the work and then just print out how many of each obs type would have been created. It is an easy way to preview what your choices would do without waiting to write an output file. It only prints the type breakdown for output file, but does print a running total of how many obs are being kept from each input file. For example:

```
&obs_sequence_tool_nml
   filename_seq       = 'obs_seq20071101',
   print_only         =  .true.,
/
```

Subset by observation or QC value

You can specify a min, max data value and/or min, max qc value, and only those within the range will be kept. There is no exclude option. For the data value, you must also specify an observation type since different types have different units and valid ranges. For example:

```
# keep only observations with a DART QC of 0:
   qc_metadata        = 'Dart quality control',
   min_qc             = 0,
   max_qc             = 0,

# keep only radiosonde temp obs between 250 and 300 K:
```

(continues on next page)

(continued from previous page)

```
   copy_metadata       = 'NCEP BUFR observation',
   copy_type           = 'RADIOSONDE_TEMPERATURE',
   min_copy            = 250.0,
   max_copy            = 300.0,
```

6.106.4 Discussion

DART observation sequence files are lists of individual observations, each with a type, a time, one or more values (called copies), zero or more quality control flags, a location, and an error estimate. Regardless of the physical order of the observations in the file, they are always processed in increasing time order, using a simple linked list mechanism. This tool reads in one or more input observation sequence files, and creates a single output observation sequence file with all observations sorted into a single, monotonically increasing time ordered output file.

DART observation sequence files contain a header with the total observation count and a table of contents of observation types. The output file from this tool culls out unused observations, and only includes observation types in the table of contents which actually occur in the output file. The table of contents **does not** need to be the same across multiple files to merge them. Each file has a self-contained numbering system for observation types. However, the `obs_sequence_tool` must be compiled with a list of observation types (defined in the `obs_def` files listed in the `preprocess` namelist) which includes all defined types across all input files. See the building section below for more details on compiling the tool.

The tool can handle observation sequence files at any stage along the processing pipeline: a template file with locations but no data, input files for an assimilation which have observation values only, or output files from an assimilation which then might include the prior and posterior mean and standard deviation, and optionally the output from the forward operator from each ensemble member. In all of these cases, the format of each individual observation is the same. It has zero or more *copies*, which is where the observation value and the means, forward operators, etc are stored. Each observation also has zero or more quality control values, *qc*, which can be associated with the incoming data quality, or can be added by the DART software to indicate how the assimilation processed this observation. Each of the copies and qc entries has an single associated character label at the start of the observation sequence file which describes what each entry is, called the *metadata*.

For multiple observation sequence files to be merged they must have the same number of *copies* and *qc* values, and all associated *metadata* must be identical. To merge multiple files where the numbers do not match exactly, the tool can be used on the individual files to rename, subset, and reorder the *copies* and/or *qc* first, and then the resulting files are mergeable. To merge multiple files where the metadata strings do not match, but the data copy or qc values are indeed the same things, there are options to rename the metadata strings. **This option should be used with care. If the copies or qc values in different files are not really the same, the tool will go ahead and merge them but the resulting file will be very wrong.**

The tool offers an additional option for specifying a list of input files. The user creates an ASCII file by any desired method (e.g. ls > file, editor), with one filename per line. The names on each line in the file should not have any delimiters, e.g. no single or double quotes at the start or end of the filename. They specify this file with the `filename_seq_list` namelist item, and the tool opens the list file and processes each input file in turn. The namelist item `num_input_files` is now DEPRECATED and is ignored. The number of input files is computed from either the explicit list in `filename_seq`, or the contents of the `filename_seq_list` file.

Time is stored inside of DART as a day number and number of seconds, which is the same no matter which calendar is being used. But many real-world observations use the Gregorian calendar for converting between number of days and an actual date. If the `gregorian_cal` namelist item is set to `.TRUE.` then any times will be printed out to the log file will be both in day/seconds and calendar date. If the observation times are not using the Gregorian calendar, then set this value to `.FALSE.` and only days/seconds will be printed.

The most common use of this tool is to process a set of input files into a single output file, or to take one input file and extract a subset of observations into a smaller file. The examples section below outlines several common scenerios.

The tool now also allows the number of copies to be changed, but only to select subsets or reorder them. It is not yet possible to merge copies or QCs from observations in different files into a single observation with more copies.

Observations can also be selected by a given range of quality control values or data values.

Observations can be restricted to a given bounding box, either in latitude and longitude (in the horizontal only), or if the observations have 1D locations, then a single value for min_box and max_box can be specified to restrict the observations to a subset of the space.

6.106.5 Faq

Can i merge files where the observation types are different?

Yes. The numbering in the table of contents at the top of each file is only local to that file. All processing of types is done with the string name, not the numbers. Neither the set of obs types, nor the observation numbers need to match across files.

I get an error about unknown observation types

Look at the `&preprocess_nml` namelist in the input.nml file in the directory where your tool was built. It must have all the observation types you need to handle listed in the `input_files` item.

Can i list more files than necessary in my input file list?

Sure. It will take slightly longer to run, in that the tool must open the file and check the times and observation types. But it is not an error to list files where no observations will be copied to the output file. It is a common task to list a set of observation files and then set the first and last observation times, run the tool to select a shorter time period, then change the first and last times and run again with the same list of files.

6.106.6 Building

Most `$DART/models/*/work` directories will build the tool along with other executable programs. It is also possible to build the tool in the `$DART/observations/utilities` directory. The `preprocess` program must be built and run first, to define what set of observation types will be supported. See the *PROGRAM preprocess* for more details on how to define the list and run it. The combined list of all observation types which will be encountered over all input files must be in the preprocess input list. The other important choice when building the tool is to include a compatible locations module. For the low-order models, the `oned` module should be used; for real-world observations, the `threed_sphere` module should be used.

6.106.7 Modules used

```
types_mod
utilities_mod
time_manager_mod
obs_def_mod
obs_sequence_mod
```

6.106.8 Files

- `input.nml`
- The input files specified in the `filename_seq` namelist variable, or inside the file named in `filename_seq_list`.
- The output file specified in the `filename_out` namelist variable.

6.106.9 References

- none

6.107 PROGRAM `integrate_model`

6.107.1 Overview

Generic main program which can be compiled with a model-specific `model_mod.f90` file. The model must provide an `adv_1step()` subroutine which advances one copy of the model forward in time.

The executable built by this program can be used by the serial program `perfect_model_obs`, or either the serial or parallel version of the `filter` program. This program is called by the default script in the template directory called `advance_model.csh`, and is selected by setting the corresponding `"async = "` namelist setting to 2.

This program only advances a single ensemble member per execution and is expected to be run as a serial program. It can be compiled with the MPI wrappers and called with mpirun with more than 1 task, however, it will only call the model advance subroutine from a single task (task 0). This can be useful in testing various scripting options using simpler and smaller models in preparation for running a larger parallel model.

6.107.2 Namelist

There is no namelist for this program.

6.107.3 Modules used

```
types_mod
time_manager_mod
utilities_mod
assim_model_mod
obs_model_mod
ensemble_manager_mod
mpi_utilities_mod
```

6.107.4 Files

- inputfile (temp_ic)
- outputfile (temp_ud)

6.107.5 References

- none

6.108 PROGRAM `obs_diag` (for 1D observations)

6.108.1 Overview/usage

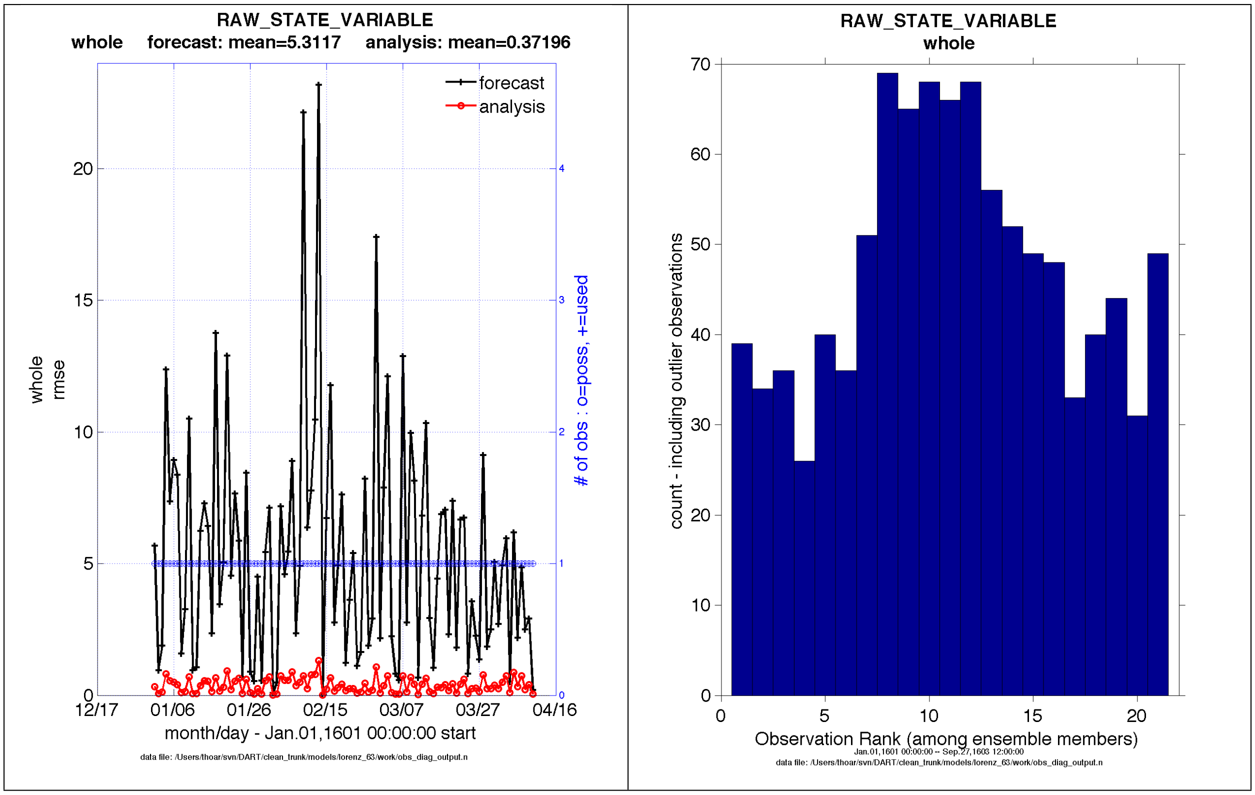
Main program for observation-space diagnostics for the models with 1D locations. 18 quantities are calculated for each region for each temporal bin specified by user input. The result of the code is a netCDF file that contains the 18 quantities of the prior (aka 'guess') and posterior (aka 'analysis') estimates as a function of time and region as well as all the metadata to create meaningful figures. **The 1D version of ``obs_diag`` has defaults that automatically set the first and last bin center based on the first and last observation time in the set of observations being processed.** This is different behavior than the 3D versions.

Each `obs_seq.final` file contains an observation sequence that has multiple 'copies' of the observation. One copy is the actual observation, another copy is the prior ensemble mean estimate of the observation, one is the spread of the prior ensemble estimate, one may be the prior estimate from ensemble member 1, . . . etc. The only observations for the 1D models are generally the result of a 'perfect model' experiment, so there is an additional copy called the 'truth' - the noise-free expected observation given the true model state. Since this copy does not, in general, exist for the high-order models, all comparisons are made with the copy labelled 'observation'. There is also a namelist variable (`use_zero_error_obs`) to compare against the 'truth' instead; the observation error variance is then automatically set to zero.

Each ensemble member applies a forward observation operator to the state to compute the "expected" value of an observation. Please note: the forward observation operator is applied **AFTER** any prior inflation has taken place! Similarly, the forward observation operator is applied AFTER any posterior inflation. This has always been the case. For a detailed look at the relationship between the observation operators and inflation, please look at the Detailed Program Execution Flow section of *PROGRAM filter*.

Given multiple estimates of the observation, several quantities can be calculated. It is possible to compute the expected observations from the state vector before assimilating (the "guess", "forecast", or "prior") or after the assimilation (the "analysis", or "posterior").

Even with `input.nml:filter_nml:num_output_obs_members` set to `0`; the full [prior,posterior] ensemble mean and [prior,posterior] ensemble spread are preserved in the `obs_seq.final` file. Consequently, the ensemble means and spreads are used to calculate the diagnostics. If the `input.nml:filter_nml:num_output_obs_members` is set to `80` (for example); the first 80 ensemble members prior and posterior "expected" values of the observation are also included. In this case, the `obs_seq.final` file contains enough information to calculate a rank histograms, verify forecasts, etc. The ensemble means are still used for many other calculations.



Since this program is fundamentally interested in the response as a function of region, there are three versions of this program; one for each of the `oned, threed_sphere, or threed_cartesian` location modules (`location_mod.f90`). It did not make sense to ask the `lorenz_96` model what part of North America you'd like to investigate or how you would like to bin in the vertical. The low-order models write out similar netCDF files and the Matlab scripts have been updated accordingly. The oned observations have locations conceptualized as being on a unit circle, so only the namelist input variables pertaining to longitude are used.

`obs_diag` is designed to explore the effect of the assimilation in two ways; 1) as a function of time for a particular variable (this is the figure on the left), and sometimes 2) in terms of a rank histogram - "Where does the actual observation rank relative to the rest of the ensemble?" (figure on the right). The figures were created by Matlab® scripts that query the `obs_diag_output.nc` file: *DART/diagnostics/matlab/*plot_evolution.m and plot_rank_histogram.m. Both of these takes as input a file name and a 'quantity' to plot ('rmse','spread','totalspread', . . .) and exhaustively plots the quantity (for every variable, every region) in a single matlab figure window - and creates a series of .ps files with multiple pages for each of the figures. The directory gets cluttered with them.

The observation sequence files contain only the time of the observation, nothing of the assimilation interval, etc. - so it requires user guidance to declare what sort of temporal binning for the temporal evolution plots. I do a 'bunch' of arithmetic on the namelist times to convert them to a series of temporal bin edges that are used when traversing the observation sequence. The actual algorithm is that the user input for the start date and bin width set up a sequence that ends in one of two ways . . . the last time is reached or the number of bins has been reached. **NOTE:** for the purpose of interpretability, the 1D `obs_diag` routines saves 'dates' as GREGORIAN dates despite the fact these systems have no concept of a calendar.

`obs_diag` reads `obs_seq.final` files and calculates the following quantities (in no particular order) for an arbitrary number of regions and levels. `obs_diag` creates a netCDF file called `obs_diag_output.nc`. It is necessary to query the `CopyMetaData` variable to determine the storage order (i.e. "which copy is what?") if you want to use your own plotting routines.

ncdump -f F -v CopyMetaData obs_diag_output.nc

Nposs	The number of observations available to be assimilated.
Nused	The number of observations that were assimilated.
rmse	The root-mean-squared error (the horizontal wind components are also used to calculate the vector wind velocity and its RMS error).
bias	The simple sum of forecast - observation. The bias of the horizontal wind speed (not velocity) is also computed.
spread	The standard deviation of the univariate obs. DART does not exploit the bivariate nature of U,V winds and so the spread of the horizontal wind is defined as the sum of the spreads of the U and V components.
totalspread	The total standard deviation of the estimate. We pool the ensemble variance of the observation plus the observation error variance and take the square root.
NbadDARTQC	the number of observations that had a DART QC value (> 1 for a prior, > 3 for a posterior)
observation	the mean of the observation values
ens_mean	the ensemble mean of the model estimates of the observation values
N_trusted	the number of implicitly trusted observations, regardless of DART QC
N_DARTqc_0	the number of observations that had a DART QC value of 0
N_DARTqc_1	the number of observations that had a DART QC value of 1
N_DARTqc_2	the number of observations that had a DART QC value of 2
N_DARTqc_3	the number of observations that had a DART QC value of 3
N_DARTqc_4	the number of observations that had a DART QC value of 4
N_DARTqc_5	the number of observations that had a DART QC value of 5
N_DARTqc_6	the number of observations that had a DART QC value of 6
N_DARTqc_7	the number of observations that had a DART QC value of 7
N_DARTqc_8	the number of observations that had a DART QC value of 8

The DART QC flag is intended to provide information about whether the observation was assimilated, evaluated only, whether the assimilation resulted in a 'good' observation, etc. *DART QC values lower than* **2** *indicate the prior and posteriors are OK.* DART QC values higher than **3** were **not** assimilated or evaluated. Here is the table that should explain things more fully:

DART QC flag value

meaning

0

observation assimilated

1

observation evaluated only (because of namelist settings)

2

assimilated, but the posterior forward operator failed

3

evaluated only, but the posterior forward operator failed

4

prior forward operator failed

5

not used because observation type not listed in namelist

6

rejected because incoming observation QC too large

7

rejected because of a failed outlier threshold test

8

vertical conversion failed

9+

reserved for future use

6.108.2 What is new in the Manhattan release

1. Support for DART QC = 8 (failed vertical conversion). This is provided simply to make the netCDF files as consistent as needed for plotting purposes.
2. Simplified input file specification.
3. Some of the internal variable names have been changed to make it easier to distinguish between variances and standard deviations.

6.108.3 What is new in the Lanai release

`obs_diag` has several improvements:

1. Support for 'trusted' observations. Trusted observation types may be specified in the namelist and all observations of that type will be counted in the statistics despite the DART QC code (as long as the forward observation operator succeeds). See namelist variable `trusted_obs`.
2. Support for 'true' observations (i.e. from an OSSE). If the 'truth' copy of an observation is desired for comparison (instead of the default copy) the observation error variance is set to 0.0 and the statistics are calculated relative to the 'truth' copy (as opposed to the normal 'noisy' or 'observation' copy). See namelist variable `use_zero_error_obs`.
3. discontinued the use of `rat_cri` and `input_qc_threshold` namelist variables. Their functionality was replaced by the DART QC mechanism long ago.
4. The creation of the rank histogram (if possible) is now namelist-controlled by namelist variable `create_rank_histogram`.

6.108.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_diag_nml
   obs_sequence_name      = ''
   obs_sequence_list      = ''
   bin_width_days         = -1
   bin_width_seconds      = -1
```

(continues on next page)

(continued from previous page)

```
   init_skip_days        = 0
   init_skip_seconds     = 0
   max_num_bins          = 9999
   Nregions              = 3
   lonlim1               = 0.0, 0.0, 0.5
   lonlim2               = 1.0, 0.5, 1.0
   reg_names             = 'whole', 'yin', 'yang'
   trusted_obs           = 'null'
   use_zero_error_obs    = .false.
   create_rank_histogram = .true.
   outliers_in_histogram = .true.
   verbose               = .false.
   /
```

The allowable ranges for the region boundaries are: lon [0.0, 1.0). The 1D locations are conceived as the distance around a unit sphere. An observation with a location exactly ON a region boundary cannot 'count' for both regions. The logic used to resolve this is:

> if((lon lon1) .and. (lon < lon2)) keeper = .true.

Consequently, if you want to include an observation precisely AT 1.0, (for example), you need to specify something a little larger than 1.0.

You can only specify **either** `obs_sequence_name` **or** `obs_sequence_list` – not both. One of them has to be an empty string … i.e. `''`.

Item	Type	Description
obs_sequence_name	character(len=256), dimension(100)	An array of names of observation sequence files. These may be relative or absolute filenames. If this is set, `obs_sequence_list` must be set to ' ' (empty string).
obs_sequence_list	character(len=256)	Name of an ascii text file which contains a list of one or more observation sequence files, one per line. If this is specified, `obs_sequence_name` must be set to ' '. Can be created by any method, including sending the output of the 'ls' command to a file, a text editor, or another program. If this is set, `obs_sequence_name` must be set to ' ' (empty string).
bin_width_days bin_width_seconds	integer	Specifies the width of the analysis window. All observations within a window centered at the observation time +/- bin_width_[days,seconds] is used. If both values are 0, half the separation between observation times as defined in the observation sequence file is used for the bin width (i.e. all observations used).
init_skip_days, init_skip_seconds	integer	Ignore all observations before this time. This allows one to skip the 'spinup' or stabilization period of an assimilation.
max_num_bins	integer	This provides a way to restrict the number of temporal bins. If `max_num_bins` is set to '10', only 10 timesteps will be output, provided there are that many.
Nregions	integer	The number of regions for the unit circle for which you'd like observation-space diagnostics. If 3 is not enough increase `MaxRegions` in `obs_diag.f90` and recompile.
lonlim1	real(r8) array of length(Nregions)	starting value of coordinates defining 'regions'. A value of -1 indicates the start of 'no region'.
lonlim2	real(r8) array of length(Nregions)	ending value of coordinates defining 'regions'. A value of -1 indicates the end of 'no region'.
reg_names	character(len=6), dimension(Nregions)	Array of names for each of the regions. The default example has the unit circle as a whole and divided into two equal parts, so there are only three regions.
trusted_obs	character(len=32), dimension(5)	Array of names for observation TYPES that will be included in the statistics if at all possible (i.e. the forward observation operator succeeds). For 1D observations the only choices in the code as distributed are 'RAW_STATE_VARIABLE' and/or 'RAW_STATE_1D_INTEGRAL'. (Additional 1D observation types can be added by the user.)
use_zero_error_obs	logical	if `.true.`, the observation copy used for the statistics calculations will be 'truth'. Only 'perfect' observations (from `perfect_model_obs`) have this copy. The observation error variance will be set to zero.
create_rank_histogram	logical	if `.true.` and there are actual ensemble estimates of the observations in the `obs_seq.final` (i.e. `filter_nml:num_output_obs_members` is larger than zero), a rank histogram will be created.
outliers_in_histogram	logical	if `.true.` the observations that have been rejected by the outlier threshhold mechanism will be *included* in the calculation of the rank histogram.
verbose	logical	switch controlling amount of run-time output.

6.108.5 Modules directly used

```
types_mod
obs_sequence_mod
obs_def_mod
obs_kind_mod
location_mod
time_manager_mod
utilities_mod
sort_mod
random_seq_mod
```

6.108.6 Modules indirectly used

```
assim_model_mod
cov_cutoff_mod
model_mod
null_mpi_utilities_mod
```

6.108.7 Files

- `input.nml` is used for `obs_diag_nml`
- `obs_diag_output.nc` is the netCDF output file
- `dart_log.out` list directed output from the obs_diag.
- `LargeInnov.txt` contains the distance ratio histogram – useful for estimating the distribution of the magnitudes of the innovations.

Discussion of obs_diag_output.nc

Every observation type encountered in the observation sequence file is tracked separately, and aggregated into temporal and spatial bins. There are two main efforts to this program. One is to track the temporal evolution of any of the quantities available in the netCDF file for any possible observation type:

ncdump -v CopyMetaData,ObservationTypes obs_diag_output.nc

The other is to explore the vertical profile of a particular observation kind. By default, each observation kind has a 'guess/prior' value and an 'analysis/posterior' value - which shed some insight into the innovations.

Temporal evolution

The `obs_diag_output.nc` output file has all the metadata I could think of, as well as separate variables for every observation type in the observation sequence file. Furthermore, there is a separate variable for the 'guess/prior' and 'analysis/posterior' estimate of the observation. To distinguish between the two, a suffix is appended to the variable name. An example seems appropriate:

```
...
char CopyMetaData(copy, stringlength) ;
        CopyMetaData:long_name = "quantity names" ;
...
```

(continues on next page)

(continued from previous page)

```
int rank_bins(rank_bins) ;
        rank_bins:long_name = "rank histogram bins" ;
        rank_bins:comment = "position of the observation among the sorted noisy␣
↪ensemble members" ;
float RAW_STATE_VARIABLE_guess(time, copy, region) ;
        RAW_STATE_VARIABLE_guess:_FillValue = -888888.f ;
        RAW_STATE_VARIABLE_guess:missing_value = -888888.f ;
float RAW_STATE_VARIABLE_analy(time, copy, region) ;
        RAW_STATE_VARIABLE_analy:_FillValue = -888888.f ;
        RAW_STATE_VARIABLE_analy:missing_value = -888888.f ;
...
```

Rank histograms

If it is possible to calculate a rank histogram, there will also be :

```
 ...
int RAW_STATE_VARIABLE_guess_RankHist(time, rank_bins, region) ;
 ...
```

as well as some global attributes. The attributes reflect the namelist settings and can be used by plotting routines to provide additional annotation for the histogram.

```
:DART_QCs_in_histogram = 0, 1, 2, 3, 7 ;
:outliers_in_histogram = "TRUE" ;
```

Please note:
netCDF restricts variable names to 40 characters, so '_Rank_Hist' may be truncated.

6.108.8 References

1. none

6.108.9 Private components

N/A

6.109 PROGRAM obs_diag (for observations that use the threed_cartesian location module)

6.109.1 Overview

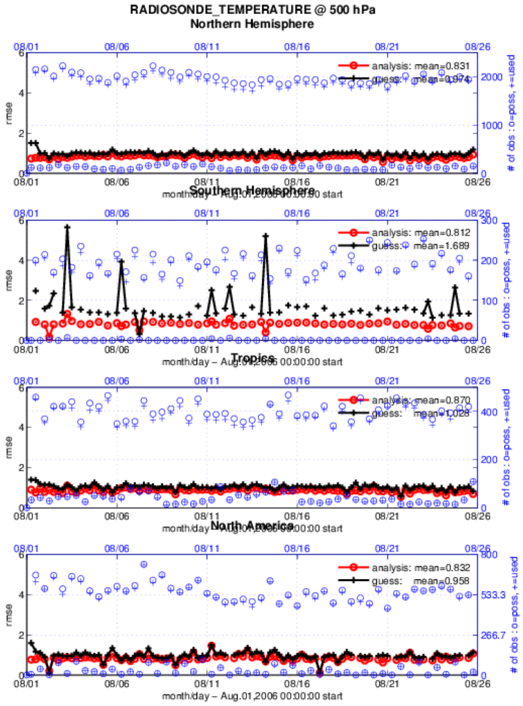
Main program for evaluating filter performance in observation space. Primarily, the prior or posterior ensemble mean (and spread) are compared to the observation and several quantities are calculated. These quantities are then saved in a netCDF file that has all the metadata to create meaningful figures.

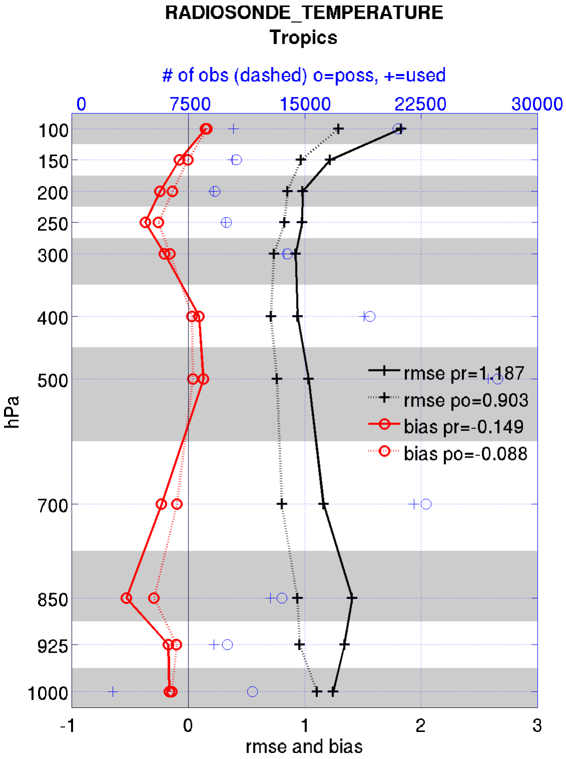
Each `obs_seq.final` file contains an observation sequence that has multiple 'copies' of the observation. One copy is the actual observation, another copy is the prior ensemble mean estimate of the observation, one is the spread of the prior ensemble estimate, one may be the prior estimate from ensemble member 1, ... etc. If the original observation sequence is the result of a 'perfect model' experiment, there is an additional copy called the 'truth' - the noise-free expected observation given the true model state. Since this copy does not, in general, exist for the high-order models, all comparisons are made with the copy labelled 'observation'. There is also a namelist variable (`use_zero_error_obs`) to compare against the 'truth' instead; the observation error variance is then automatically set to zero.

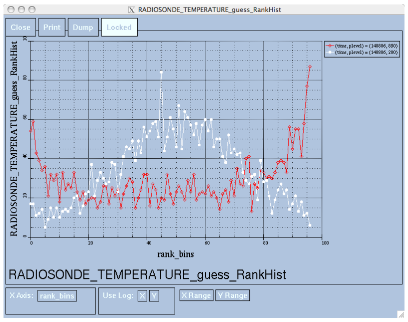
Each ensemble member applies a forward observation operator to the state to compute the "expected" value of an observation. Please note: the forward observation operator is applied **AFTER** any prior inflation has taken place! Similarly, the forward observation operator is applied AFTER any posterior inflation. This has always been the case. For a detailed look at the relationship between the observation operators and inflation, please look at the Detailed Program Execution Flow section of *PROGRAM filter*.

Given multiple estimates of the observation, several quantities can be calculated. It is possible to compute the expected observations from the state vector before assimilating (the "guess", "forecast", or "prior") or after the assimilation (the "analysis", or "posterior").

Even with `input.nml:filter_nml:num_output_obs_members` set to `0`; the full [prior,posterior] ensemble mean and [prior,posterior] ensemble spread are preserved in the `obs_seq.final` file. Consequently, the ensemble means and spreads are used to calculate the diagnostics. If the `input.nml:filter_nml:num_output_obs_members` is set to `80` (for example); the first 80 ensemble members prior and posterior "expected" values of the observation are also included. In this case, the `obs_seq.final` file contains enough information to calculate a rank histograms, verify forecasts, etc. The ensemble means are still used for many other calculations.


RADIOSONDE_TEMPERATURE @ 500 hPa
Northern Hemisphere
analysis: mean=0.831
guess: mean=0.974
Southern Hemisphere
analysis: mean=0.812
guess: mean=1.689
Tropics
analysis: mean=0.870
guess: mean=1.028
North America
analysis: mean=0.832
guess: mean=0.958
rmse
# of obs : o=poss, +=used
08/01
08/06
08/11
08/16
08/21
08/26
0
2
4
6
1000
2000
100
200
300
400
266.7
533.3
800
month/day – Aug.01,2006 00:00:00 start


RADIOSONDE_TEMPERATURE
Tropics
# of obs (dashed) o=poss, +=used
0
7500
15000
22500
30000
100
150
200
250
300
400
500
700
850
925
1000
hPa
rmse pr=1.187
rmse po=0.903
bias pr=-0.149
bias po=-0.088
-1
0
1
2
3
rmse and bias


RADIOSONDE_TEMPERATURE_guess_RankHist
Close
Print
Dump
Locked
(time, plevel) = (148886, 850)
(time, plevel) = (148886, 200)
RADIOSONDE_TEMPERATURE_guess_RankHist
0
20
40
60
80
100
rank_bins
RADIOSONDE_TEMPERATURE_guess_RankHist
X Axis:
rank_bins
Use Log:
X
Y
X Range
Y Range

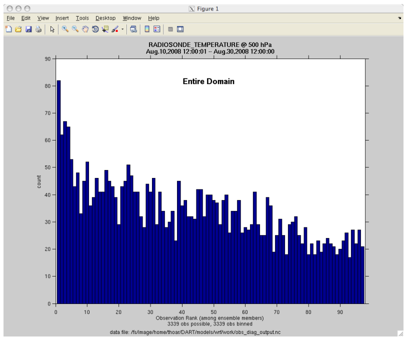


Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help
RADIOSONDE_TEMPERATURE @ 500 hPa
Aug.10,2008 12:00:01 -- Aug.30,2008 12:00:00
Entire Domain
count
0
10
20
30
40
50
60
70
80
90
Observation Rank (among ensemble members)
3339 obs possible, 3339 obs binned
data file: /fs/image/home/thoar/DART/models/wrf/work/obs_diag_output.nc

Since this program is fundamentally interested in the response as a function of region, there are three versions of this program; one for each of the `oned, threed_sphere, or threed_cartesian` location modules (`location_mod.f90`). It did not make sense to ask the `lorenz_96` model what part of North America you'd like to investigate or how you would like to bin in the vertical. The low-order models write out similar netCDF files and the Matlab scripts have been updated accordingly. The oned observations have locations conceptualized as being on a unit circle, so only the namelist input variables pertaining to longitude are used.

Identity observations (only possible from "perfect model experiments") are already explored with state-space diagnostics, so `obs_diag` simply skips them.

`obs_diag` is designed to explore the effect of the assimilation in three ways; 1) as a function of time for a particular variable and level (this is the figure on the left), 2) as a time-averaged vertical profile (figure in the middle), and sometimes 3) in terms of a rank histogram - "Where does the actual observation rank relative to the rest of the ensemble?" (figures on the right). The figures on the left and center were created by several Matlab® scripts that query the `obs_diag_output.nc` file: *DART/diagnostics/matlab/*plot_evolution.m and plot_profile.m. Both of these takes as input a file name and a 'quantity' to plot ('rmse','spread','totalspread', . . .) and exhaustively plots the quantity (for every variable, every level, every region) in a single matlab figure window - and creates a series of .ps files with multiple pages for each of the figures. The directory gets cluttered with them. The rank histogram information can easily be plotted with ncview, a free third-party piece of software or with plot_rank_histogram.m.

`obs_diag` can be configured to compare the ensemble estimates against the 'observation' copy or the 'truth' copy based on the setting of the `use_zero_error_obs` namelist variable.

The observation sequence files contain only the time of the observation, nothing of the assimilation interval, etc. - so it requires user guidance to declare what sort of temporal binning for the temporal evolution plots. I do a 'bunch' of arithmetic on the namelist times to convert them to a series of temporal bin edges that are used when traversing the observation sequence. The actual algorithm is that the user input for the start date and bin width set up a sequence that ends in one of two ways . . . the last time is reached or the number of bins has been reached.

`obs_diag` reads `obs_seq.final` files and calculates the following quantities (in no particular order) for an arbitrary number of regions and levels. `obs_diag` creates a netCDF file called `obs_diag_output.nc`. It is necessary to query the `CopyMetaData` variable to determine the storage order (i.e. "which copy is what?") if you want to use your own plotting routines.

ncdump -f F -v CopyMetaData obs_diag_output.nc

Nposs	The number of observations available to be assimilated.
Nused	The number of observations that were assimilated.
NbadUV	the number of velocity observations that had a matching component that was not assimilated;
NbadLV	the number of observations that were above or below the highest or lowest model level, respectively;
rmse	The root-mean-squared error (the horizontal wind components are also used to calculate the vector wind velocity and its RMS error).
bias	The simple sum of forecast - observation. The bias of the horizontal wind speed (not velocity) is also computed.
spread	The standard deviation of the univariate obs. DART does not exploit the bivariate nature of U,V winds and so the spread of the horizontal wind is defined as the sum of the spreads of the U and V components.
totalspread	The total standard deviation of the estimate. We pool the ensemble variance of the observation plus the observation error variance and take the square root.
NbadDARTQC	the number of observations that had a DART QC value (> 1 for a prior, > 3 for a posterior)
observation	the mean of the observation values
ens_mean	the ensemble mean of the model estimates of the observation values
N_trusted	the number of implicitly trusted observations, regardless of DART QC
N_DARTqc_0	the number of observations that had a DART QC value of 0
N_DARTqc_1	the number of observations that had a DART QC value of 1
N_DARTqc_2	the number of observations that had a DART QC value of 2
N_DARTqc_3	the number of observations that had a DART QC value of 3
N_DARTqc_4	the number of observations that had a DART QC value of 4
N_DARTqc_5	the number of observations that had a DART QC value of 5
N_DARTqc_6	the number of observations that had a DART QC value of 6
N_DARTqc_7	the number of observations that had a DART QC value of 7
N_DARTqc_8	the number of observations that had a DART QC value of 8

The temporal evolution of the above quantities for every observation type (RADIOSONDE_U_WIND_COMPONENT, AIRCRAFT_SPECIFIC_HUMIDITY, ...) is recorded in the output netCDF file - `obs_diag_output.nc`. This netCDF file can then be loaded and displayed using the Matlab® scripts in `..../DART/diagnostics/matlab`. (which may depend on functions in `..../DART/matlab`). The temporal, geographic, and vertical binning are under namelist control. Temporal averages of the above quantities are also stored in the netCDF file. Normally, it is useful to skip the 'burn-in' period - the amount of time to skip is under namelist control.

The DART QC flag is intended to provide information about whether the observation was assimilated, evaluated only, whether the assimilation resulted in a 'good' observation, etc. *DART QC values lower than**2**indicate the prior and posteriors are OK.* DART QC values higher than **3** were **not** assimilated or evaluated. Here is the table that should explain things more fully:

DART QC flag value

meaning

0

observation assimilated

1

observation evaluated only (because of namelist settings)

2

assimilated, but the posterior forward operator failed

3

evaluated only, but the posterior forward operator failed

4

prior forward operator failed

5

not used because observation type not listed in namelist

6

rejected because incoming observation QC too large

7

rejected because of a failed outlier threshold test

8

vertical conversion failed

9+

reserved for future use

6.109.2 What is new in the Manhattan release

1. Support for DART QC = 8 (failed vertical conversion).
2. Simplified input file specification.
3. Replace namelist integer variable `debug` with logical variable `verbose` to control amount of run-time output.
4. Removed `rat_cri` and `input_qc_threshold` from the namelists. They had been deprecated for quite some time.
5. Some of the internal variable names have been changed to make it easier to distinguish between variances and standard deviations.

6.109.3 What is new in the Lanai release

`obs_diag` has several improvements:

1. Improved vertical specification. Namelist variables `[h,p,m]level_edges` allow fine-grained control over the vertical binning. It is not allowed to specify both the edges and midpoints for the vertical bins.
2. Improved error-checking for input specification, particularly the vertical bins. Repeated values are squeezed out.
3. Support for 'trusted' observations. Trusted observation types may be specified in the namelist and all observations of that type will be counted in the statistics despite the DART QC code (as long as the forward observation operator succeeds). See namelist variable `trusted_obs`. For more details, see the section on Trusted observations.

4. Support for ‘true’ observations (i.e. from an OSSE). If the ‘truth’ copy of an observation is desired for comparison (instead of the default copy) the observation error variance is set to 0.0 and the statistics are calculated relative to the ‘truth’ copy (as opposed to the normal ‘noisy’ or ‘observation’ copy). See namelist variable `use_zero_error_obs`.
5. discontinued the use of `rat_cri` and `input_qc_threshold` namelist variables. Their functionality was replaced by the DART QC mechanism long ago.
6. The creation of the rank histogram (if possible) is now namelist-controlled by namelist variable `create_rank_histogram`.

6.109.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand ‘&’ and terminate with a slash ‘/’. Character strings that contain a ‘/’ must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_diag_nml
   obs_sequence_name     = ''
   obs_sequence_list     = ''
   first_bin_center      =  2003, 1, 1, 0, 0, 0
   last_bin_center       =  2003, 1, 2, 0, 0, 0
   bin_separation        =     0, 0, 0, 6, 0, 0
   bin_width             =     0, 0, 0, 6, 0, 0
   time_to_skip          =     0, 0, 0, 6, 0, 0
   max_num_bins          = 1000
   hlevel                = -888888.0
   hlevel_edges          = -888888.0
   Nregions              = 0
   xlim1                 = -888888.0
   xlim2                 = -888888.0
   ylim1                 = -888888.0
   ylim2                 = -888888.0
   reg_names             = 'null'
   trusted_obs           = 'null'
   create_rank_histogram = .true.
   outliers_in_histogram = .false.
   use_zero_error_obs    = .false.
   verbose               = .false.
   /
```

The date-time integer arrays in this namelist have the form (YYYY, MM, DY, HR, MIN, SEC).
The allowable ranges for the region boundaries are: latitude [-90.,90], longitude [0.,Inf.]

You can only specify **either** `obs_sequence_name` **or** `obs_sequence_list` – not both. One of them has to be an empty string … i.e. `''`.

Item	Type	Description
obs_sequence_name	character(len=256), dimension(100)	An array of names of observation sequence files. These may be relative or absolute filenames. If this is set, `obs_sequence_list` must be set to ' ' (empty string).
obs_sequence_list	character(len=256)	Name of an ascii text file which contains a list of one or more observation sequence files, one per line. If this is specified, `obs_sequence_name` must be set to ' '. Can be created by any method, including sending the output of the 'ls' command to a file, a text editor, or another program. If this is set, `obs_sequence_name` must be set to ' ' (empty string).
first_bin_center	integer, dimension(6)	first timeslot of the first obs_seq.final file to process. The six integers are: year, month, day, hour, hour, minute, second, in that order. `obs_diag` has improved run-time output that reports the time and date of the first and last observations in every observation sequence file. Look for the string 'First observation date' in the logfile. If the `verbose` is 'true', it is also written to the screen.
last_bin_center	integer, dimension(6)	last timeslot of interest. (reminder: the last timeslot of day 1 is hour 0 of day 2) The six integers are: year, month, day, hour, hour, minute, second, in that order. This does not need to be exact, the values from `first_bin_center` and `bin_separation` are used to populate the time array and stop on or before the time defined by `last_bin_center`. See also `max_num_bins`.
bin_separation	integer, dimension(6)	Time between bin centers. The year and month values *must* be zero.
bin_width	integer, dimension(6)	Time span around bin centers in which obs will be compared. The year and month values *must* be zero. Frequently, but not required to be, the same as the values for bin_separation. 0
time_to_skip	integer, dimension(6)	Time span at the beginning to skip when calculating vertical profiles of rms error and bias. The year and month values *must* be zero. Useful because it takes some time for the assimilation to settle down from the climatological spread at the start. `time_to_skip` is an amount of time AFTER the first edge of the first bin.
max_num_bins	integer	This provides an alternative way to declare the `last_bin_center`. If `max_num_bins` is set to '10', only 10 timesteps will be output - provided `last_bin_center` is set to some later date.
hlevel	real, dimension(50)	Same, but for observations that have height(m) or depth(m) as the vertical coordinate. An example of defining the midpoints is: `hlev el = 1000, 2000, 3000, 4000, 5000, 60 00, 7000, 8000, 9000, 10000, 11000,`
hlevel_edges	real, dimension(51)	The edges defining the height (or depth) levels for the vertical binning. You may specify either `hlevel` or `hlevel_edges`, but not both. An example of defining the edges is: `hlevel_edges = 0, 1500, 2500, 3500, 4500, 5500, 6500,`
Nregions	integer	Number of regions of the globe for which obs space diagnostics are computed separately. Must be between [1,50]. If 50 is not enough, increase `obs_diag.f90MaxRegions` and recompile.
xlim1	real, dimension(50)	western extent of each of the regions.
xlim2	real, dimension(50)	eastern extent of each of the regions.
ylim1	real, dimension(50)	southern extent of the regions.
ylim2	real, dimension(50)	northern extent of the regions.
reg_names	character(len=129), dimension(50)	Array of names for the regions to be analyzed. Will be used for plot titles.
trusted_obs	character(len=32),	list of observation types that **must** participate in the calculation of the statistics, regardless of the DART QC (provided that the forward observation operator can still be applied without

6.109.5 Other modules used

```
obs_sequence_mod
obs_kind_mod
obs_def_mod (and possibly other obs_def_xxx mods)
assim_model_mod
random_seq_mod
model_mod
location_mod
types_mod
time_manager_mod
utilities_mod
sort_mod
```

6.109.6 Files

- `input.nml` is used for `obs_diag_nml`
- `obs_diag_output.nc` is the netCDF output file
- `dart_log.out` list directed output from the obs_diag.
- `LargeInnov.txt` contains the distance ratio histogram – useful for estimating the distribution of the magnitudes of the innovations.

Obs_diag may require a model input file from which to get grid information, metadata, and links to modules providing the models expected observations. It all depends on what's needed by the `model_mod.f90`

Discussion of obs_diag_output.nc

Every observation type encountered in the observation sequence file is tracked separately, and aggregated into temporal and 3D spatial bins. There are two main efforts to this program. One is to track the temporal evolution of any of the quantities available in the netCDF file for any possible observation type:

ncdump -v CopyMetaData,ObservationTypes obs_diag_output.nc

The other is to explore the vertical profile of a particular observation kind. By default, each observation kind has a 'guess/prior' value and an 'analysis/posterior' value - which shed some insight into the innovations.

Temporal evolution

The `obs_diag_output.nc` output file has all the metadata I could think of, as well as separate variables for every observation type in the observation sequence file. Furthermore, there is a separate variable for the 'guess/prior' and 'analysis/posterior' estimate of the observation. To distinguish between the two, a suffix is appended to the variable name. An example seems appropriate:

```
...
char CopyMetaData(copy, stringlength) ;
        CopyMetaData:long_name = "quantity names" ;
char ObservationTypes(obstypes, stringlength) ;
        ObservationTypes:long_name = "DART observation types" ;
        ObservationTypes:comment = "table relating integer to observation type string
↪" ;
float RADIOSONDE_U_WIND_COMPONENT_guess(time, copy, hlevel, region) ;
        RADIOSONDE_U_WIND_COMPONENT_guess:_FillValue = -888888.f ;
```

(continues on next page)

(continued from previous page)

```
        RADIOSONDE_U_WIND_COMPONENT_guess:missing_value = -888888.f ;
float RADIOSONDE_V_WIND_COMPONENT_guess(time, copy, hlevel, region) ;
        RADIOSONDE_V_WIND_COMPONENT_guess:_FillValue = -888888.f ;
        RADIOSONDE_V_WIND_COMPONENT_guess:missing_value = -888888.f ;
...
float MARINE_SFC_ALTIMETER_guess(time, copy, surface, region) ;
        MARINE_SFC_ALTIMETER_guess:_FillValue = -888888.f ;
        MARINE_SFC_ALTIMETER_guess:missing_value = -888888.f ;
...
float RADIOSONDE_WIND_VELOCITY_guess(time, copy, hlevel, region) ;
        RADIOSONDE_WIND_VELOCITY_guess:_FillValue = -888888.f ;
        RADIOSONDE_WIND_VELOCITY_guess:missing_value = -888888.f ;
...
float RADIOSONDE_U_WIND_COMPONENT_analy(time, copy, hlevel, region) ;
        RADIOSONDE_U_WIND_COMPONENT_analy:_FillValue = -888888.f ;
        RADIOSONDE_U_WIND_COMPONENT_analy:missing_value = -888888.f ;
float RADIOSONDE_V_WIND_COMPONENT_analy(time, copy, hlevel, region) ;
        RADIOSONDE_V_WIND_COMPONENT_analy:_FillValue = -888888.f ;
        RADIOSONDE_V_WIND_COMPONENT_analy:missing_value = -888888.f ;
...
```

There are several things to note:

1. the ‘WIND_VELOCITY’ component is nowhere ‘near’ the corresponding U,V components.
2. all of the ‘guess’ variables come before the matching ‘analy’ variables.
3. surface variables (i.e. `MARINE_SFC_ALTIMETER` have a coordinate called ‘surface’ as opposed to ‘hlevel’ for the others in this example).

Vertical profiles

Believe it or not, there are another set of netCDF variables specifically for the vertical profiles, essentially duplicating the previous variables but **without the ‘time’ dimension**. These are distinguished by the suffix added to the observation kind - ‘VPguess’ and ‘VPanaly’ - ‘VP’ for Vertical Profile.

```
...
float SAT_WIND_VELOCITY_VPguess(copy, hlevel, region) ;
        SAT_WIND_VELOCITY_VPguess:_FillValue = -888888.f ;
        SAT_WIND_VELOCITY_VPguess:missing_value = -888888.f ;
...
float RADIOSONDE_U_WIND_COMPONENT_VPanaly(copy, hlevel, region) ;
        RADIOSONDE_U_WIND_COMPONENT_VPanaly:_FillValue = -888888.f ;
        RADIOSONDE_U_WIND_COMPONENT_VPanaly:missing_value = -888888.f ;
...
```

Observations flagged as ‘surface’ do not participate in the vertical profiles (Because surface variables cannot exist on any other level, there’s not much to plot!). Observations on the lowest level DO participate. There’s a difference!

Rank histograms

If it is possible to calculate a rank histogram, there will also be :

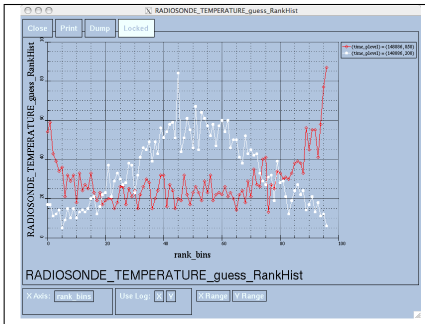
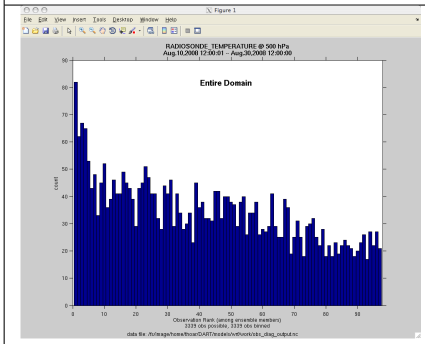
```
...
int RADIOSONDE_U_WIND_COMPONENT_guess_RankHi(time, rank_bins, hlevel, region) ;
...
int RADIOSONDE_V_WIND_COMPONENT_guess_RankHi(time, rank_bins, hlevel, region) ;
...
int MARINE_SFC_ALTIMETER_guess_RankHist(time, rank_bins, surface, region) ;
...
```

as well as some global attributes. The attributes reflect the namelist settings and can be used by plotting routines to provide additional annotation for the histogram.

```
:DART_QCs_in_histogram = 0, 1, 2, 3, 7 ;
:outliers_in_histogram = "TRUE" ;
```

Please note:

1. netCDF restricts variable names to 40 characters, so '_Rank_Hist' may be truncated.
2. It is sufficiently vague to try to calculate a rank histogram for a velocity derived from the assimilation of U,V components such that NO rank histogram is created for velocity. A run-time log message will inform as to which variables are NOT having a rank histogram variable preserved in the obs_diag_output.nc file - IFF it is possible to calculate a rank histogram in the first place.

 RADIOSONDE_TEMPERATURE_guess_RankHist Close Print Dump Locked (time, plevel) = (148886, 850) (time, plevel) = (148886, 200) RADIOSONDE_TEMPERATURE_guess_RankHist 0 20 40 60 80 100 rank_bins RADIOSONDE_TEMPERATURE_guess_RankHist X Axis: rank_bins Use Log: X Y X Range Y Range	Instructions for viewing the rank histogram with ncview.
 Figure 1 File Edit View Insert Tools Desktop Window Help RADIOSONDE_TEMPERATURE @ 500 hPa Aug.10,2008 12:00:01 -- Aug.30,2008 12:00:00 Entire Domain count 0 10 20 30 40 50 60 70 80 90 Observation Rank (among ensemble members) 3339 obs possible, 3339 obs binned data file: /fs/image/home/thoar/DART/models/wrf/work/obs_diag_output.nc	Instructions for viewing the rank histogram with Matlab.

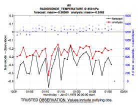
“trusted” observation types

This needs to be stated up front: obs_diag is a post-processor; it cannot influence the assimilation. One interpretation of a TRUSTED observation is that the assimilation should **always** use the observation, even if it is far from the ensemble. At present (23 Feb 2015), the filter in DART does not forcibly assimilate any one observation and selectively assimilate the others. Still, it is useful to explore the results using a set of ‘trusted type’ observations, whether they were assimilated, evaluated, or rejected by the outlier threshhold. This is the important distinction. The diagnostics can be calculated differently for each *observation type*.

The normal diagnostics calculate the metrics (rmse, bias, etc.) only for the ‘good’ observations - those that were assimilated or evaluated. The outlier_threshold essentially defines what observations are considered too far from the ensemble **prior** to be useful. These observations get a DART QC of 7 and are not assimilated. The observations with a DART QC of 7 do not contribute the the metrics being calculated. Similarly, if the forward observation operator fails, these observations cannot contribute. When the operator fails, the ‘expected’ observation value is ‘MISSING’, and there is no ensemble mean or spread.

‘Trusted type’ observation metrics are calculated using all the observations that were assimilated or evaluated **AND** the observations that were rejected by the outlier threshhold. obs_diag can post-process the DART QC and calculate the metrics appropriately for **observation types** listed in the trusted_obs namelist variable. If there are trusted observation types specified for obs_diag, the obs_diag_output.nc has global metadata to indicate that a different set of criteria were used to calculate the metrics. The individual variables also have an extra attribute. In the following output, input.nml:obs_diag_nml:trusted_obs was set: trusted_obs = 'RADIOSONDE_TEMPERATURE', 'RADIOSONDE_U_WIND_COMPONENT'

```
  ...
        float RADIOSONDE_U_WIND_COMPONENT_guess(time, copy, hlevel, region) ;
                RADIOSONDE_U_WIND_COMPONENT_guess:_FillValue = -888888.f ;
                RADIOSONDE_U_WIND_COMPONENT_guess:missing_value = -888888.f ;
                RADIOSONDE_U_WIND_COMPONENT_guess:TRUSTED = "TRUE" ;
        float RADIOSONDE_V_WIND_COMPONENT_guess(time, copy, hlevel, region) ;
                RADIOSONDE_V_WIND_COMPONENT_guess:_FillValue = -888888.f ;
                RADIOSONDE_V_WIND_COMPONENT_guess:missing_value = -888888.f ;
  ...
// global attributes:
  ...
                :trusted_obs_01 = "RADIOSONDE_TEMPERATURE" ;
                :trusted_obs_02 = "RADIOSONDE_U_WIND_COMPONENT" ;
                :obs_seq_file_001 = "cam_obs_seq.1978-01-01-00000.final" ;
                :obs_seq_file_002 = "cam_obs_seq.1978-01-02-00000.final" ;
                :obs_seq_file_003 = "cam_obs_seq.1978-01-03-00000.final" ;
  ...
                :MARINE_SFC_ALTIMETER = 7 ;
                :LAND_SFC_ALTIMETER = 8 ;
                :RADIOSONDE_U_WIND_COMPONENT--TRUSTED = 10 ;
                :RADIOSONDE_V_WIND_COMPONENT = 11 ;
                :RADIOSONDE_TEMPERATURE--TRUSTED = 14 ;
                :RADIOSONDE_SPECIFIC_HUMIDITY = 15 ;
                :AIRCRAFT_U_WIND_COMPONENT = 21 ;
  ...
```

The Matlab scripts try to ensure that the trusted observation graphics clarify that the metrics plotted are somehow ‘different’ than the normal processing stream. Some text is added to indicate that the values include the outlying observations. **IMPORTANT:** The interpretation of the number of observations ‘possible’ and ‘used’ still reflects what was used **in the assimilation!** The number of observations rejected by the outlier threshhold is not explicilty plotted. To reinforce this, the text for the observation axis on all graphics has been changed to `"o=possible, *=assimilated"`. In short, the distance between the number of observations possible and the number assimilated still reflects the number of observations rejected by the outlier threshhold and the number of failed forward observation operators.	

There is ONE ambiguous case for trusted observations. There may be instances in which the observation fails the outlier threshhold test (which is based on the prior) and the posterior forward operator fails. DART does not have a QC that explicilty covers this case. The current logic in `obs_diag` correctly handles these cases **except** when trying to use ‘trusted’ observations. There is a section of code in `obs_diag` that may be enabled if you are encountering this ambiguous case. As `obs_diag` runs, a warning message is issued and a summary count is printed if the ambiguous case is encountered. What normally happens is that if that specific observation type is trusted, the posterior values include a MISSING value in the calculation which makes them inaccurate. If the block of code is enabled, the DART QC is recast as the PRIOR forward observation operator fails. This is technically incorrect, but for the case of trusted observations, it results in only calculating statistics for trusted observations that have a useful prior and posterior. **This should not be used unless you are willing to intentionally disregard ‘trusted’ observations that were rejected by the outlier threshhold.** Since the whole point of a trusted observation is to *include* observations potentially rejected by the outlier threshhold, you see the problem. Some people like to compare the posteriors. *THAT* can be the problem.

```
if ((qc_integer == 7) .and. (abs(posterior_mean(1) - MISSING_R8) < 1.0_r8)) then
            write(string1,*)'WARNING ambiguous case for obs index ',obsindex
            string2 = 'obs failed outlier threshhold AND posterior operator failed.'
            string3 = 'Counting as a Prior QC == 7, Posterior QC == 4.'
            if (trusted) then
! COMMENT      string3 = 'WARNING changing DART QC from 7 to 4'
! COMMENT      qc_integer = 4
            endif
            call error_handler(E_MSG,'obs_diag',string1,text2=string2,text3=string3)
            num_ambiguous = num_ambiguous + 1
         endif
```

6.109.7 Usage

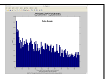
`obs_diag` is built in . . . /DART/models/*your_model*/work, in the same way as the other DART components.

Multiple observation sequence files

There are two ways to specify input files for `obs_diag`. You can either specify the name of a file containing a list of files (in `obs_sequence_list`), or you may specify a list of files via `obs_sequence_name`.

Example: observation sequence files spanning 30 days

In this example, we will be accumulating metrics for 30 days. The `obs_diag_output.nc` file will have exactly ONE timestep in it (so it won't be much use for the `plot_evolution` functions) - but the `plot_profile` functions and the `plot_rank_histogram` function will be used to explore the assimilation. By way of an example, we will NOT be using outlier observations in the rank histogram. Lets presume that all your `obs_seq.final` files are in alphabetically-nice directories:



```
/Exp1/Dir01/obs_seq.final
/Exp1/Dir02/obs_seq.final
/Exp1/Dir03/obs_seq.final
...
/Exp1/Dir99/obs_seq.final
```

The first step is to create a file containing the list of observation sequence files you want to use. This can be done with the unix command 'ls' with the -1 option (that's a number one) to put one file per line.

ls -1 /Exp1/Dir*/obs_seq.final > obs_file_list.txt

It is necessary to turn on the verbose option to check the first/last times that will be used for the histogram. Then, the namelist settings for 2008 07 31 12Z through 2008 08 30 12Z are:

```
&obs_diag_nml
   obs_sequence_name     = ''
   obs_sequence_list     = 'obs_file_list.txt'
   first_bin_center      =  2008, 8,15,12, 0, 0
   last_bin_center       =  2008, 8,15,12, 0, 0
   bin_separation        =     0, 0,30, 0, 0, 0
   bin_width             =     0, 0,30, 0, 0, 0
   time_to_skip          =     0, 0, 0, 0, 0, 0
   max_num_bins          = 1000
   Nregions              = 1
   xlim1                 = -1.0
   xlim2                 = 1000000.0
   ylim1                 = -1.0
   ylim2                 = 1000000.0
   reg_names             = 'Entire Domain'
   create_rank_histogram = .true.
   outliers_in_histogram = .false.
   verbose               = .true.
   /
```

then, simply run `obs_diag` in the usual manner - you may want to save the run-time output to a file. Here is a portion of the run-time output:

```
...
Region   1 Entire Domain                        (WESN):     0.0000    360.0000    -90.0000  ␣
↪ 90.0000
 Requesting             1  assimilation periods.

epoch       1  start day=148865, sec=43201
epoch       1 center day=148880, sec=43200
epoch       1    end day=148895, sec=43200
epoch       1  start 2008 Jul 31 12:00:01
epoch       1 center 2008 Aug 15 12:00:00
```

(continues on next page)

(continued from previous page)

```
epoch      1    end 2008 Aug 30 12:00:00
...
MARINE_SFC_HORIZONTAL_WIND_guess_RankHis has            0 "rank"able observations.
SAT_HORIZONTAL_WIND_guess_RankHist       has            0 "rank"able observations.
...
```

Discussion: It should be pretty clear that there is exactly 1 assimilation period, it may surprise you that the start is 1 second past 12Z. This is deliberate and reflects the DART convention of starting intervals 1 second after the end of the previous interval. The times in the netCDF variables reflect the defined start/stop of the period, regardless of the time of the first/last observation.

Please note that none of the 'horizontal_wind' variables will have a rank histogram, so they are not written to the netCDF file. ANY variable that does not have a rank histogram with some observations will NOT have a rank histogram variable in the netCDF file.

Now that you have the `obs_diag_output.nc`, you can explore it with plot_profile.m, plot_bias_xxx_profile.m, or plot_rmse_xxx_profile.m, and look at the rank histograms with ncview or `plot_rank_histogram.m`.

6.109.8 References

1. none

6.109.9 Private components

N/A

6.110 PROGRAM `obs_diag` (for observations that use the threed_sphere location module)

6.110.1 Overview

Main program for evaluating filter performance in observation space. Primarily, the prior or posterior ensemble mean (and spread) are compared to the observation and several quantities are calculated. These quantities are then saved in a netCDF file that has all the metadata to create meaningful figures.

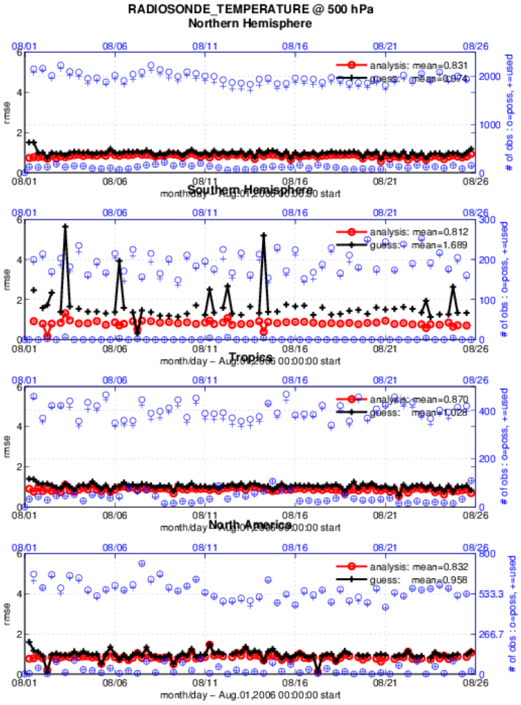
Each `obs_seq.final` file contains an observation sequence that has multiple 'copies' of the observation. One copy is the actual observation, another copy is the prior ensemble mean estimate of the observation, one is the spread of the prior ensemble estimate, one may be the prior estimate from ensemble member 1, ... etc. If the original observation sequence is the result of a 'perfect model' experiment, there is an additional copy called the 'truth' - the noise-free expected observation given the true model state. Since this copy does not, in general, exist for the high-order models, all comparisons are made with the copy labelled 'observation'. There is also a namelist variable (`use_zero_error_obs`) to compare against the 'truth' instead; the observation error variance is then automatically set to zero.

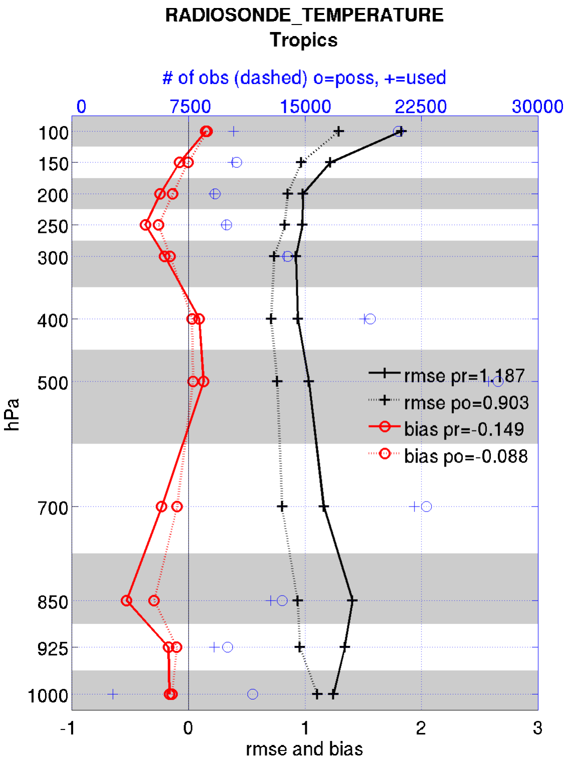
Each ensemble member applies a forward observation operator to the state to compute the "expected" value of an observation. Please note: the forward observation operator is applied **AFTER** any prior inflation has taken place! Similarly, the forward observation operator is applied AFTER any posterior inflation. This has always been the case.

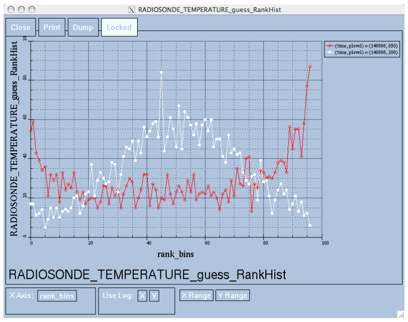
For a detailed look at the relationship between the observation operators and inflation, please look at the Detailed Program Execution Flow section of *PROGRAM filter*.

Given multiple estimates of the observation, several quantities can be calculated. It is possible to compute the expected observations from the state vector before assimilating (the "guess", "forecast", or "prior") or after the assimilation (the "analysis", or "posterior").

Even with `input.nml:filter_nml:num_output_obs_members` set to `0`; the full [prior,posterior] ensemble mean and [prior,posterior] ensemble spread are preserved in the `obs_seq.final` file. Consequently, the ensemble means and spreads are used to calculate the diagnostics. If the `input.nml:filter_nml:num_output_obs_members` is set to `80` (for example); the first 80 ensemble members prior and posterior "expected" values of the observation are also included. In this case, the `obs_seq.final` file contains enough information to calculate a rank histograms, verify forecasts, etc. The ensemble means are still used for many other calculations.






RADIOSONDE_TEMPERATURE_guess_RankHist
Close
Print
Dump
Locked
RADIOSONDE_TEMPERATURE_guess_RankHist
rank_bins
RADIOSONDE_TEMPERATURE_guess_RankHist
X Axis:
rank_bins
Use Log:
X
Y
X Range
Y Range

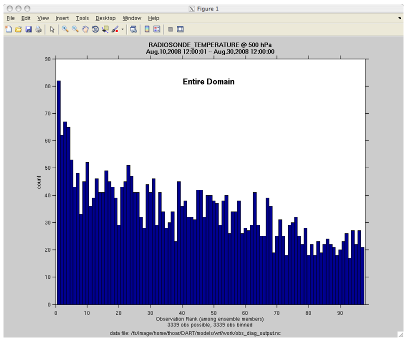


Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help
RADIOSONDE_TEMPERATURE @ 500 hPa
Aug.10,2008 12:00:01 - Aug.30,2008 12:00:00
Entire Domain
count
Observation Rank (among ensemble members)
3339 obs possible, 3339 obs binned

Since this program is fundamentally interested in the response as a function of region, there are three versions of this program; one for each of the `oned, threed_sphere, or threed_cartesian` location modules (`location_mod.f90`). It did not make sense to ask the `lorenz_96` model what part of North America you'd like to investigate or how you would like to bin in the vertical. The low-order models write out similar netCDF files and the Matlab scripts have been updated accordingly. The oned observations have locations conceptualized as being on a unit circle, so only the namelist input variables pertaining to longitude are used.

Identity observations (only possible from "perfect model experiments") are already explored with state-space diagnostics, so obs_diag simply skips them.

obs_diag is designed to explore the effect of the assimilation in three ways; 1) as a function of time for a particular variable and level (this is the figure on the left), 2) as a time-averaged vertical profile (figure in the middle), and sometimes 3) in terms of a rank histogram - "Where does the actual observation rank relative to the rest of the ensemble?" (figures on the right). The figures on the left and center were created by several Matlab® scripts that query the obs_diag_output.nc file: *DART/diagnostics/matlab/*plot_evolution.m and plot_profile.m. Both of these takes as input a file name and a 'quantity' to plot ('rmse','spread','totalspread', . . .) and exhaustively plots the quantity (for every variable, every level, every region) in a single matlab figure window - and creates a series of .ps files with multiple pages for each of the figures. The directory gets cluttered with them. The rank histogram information can easily be plotted with ncview, a free third-party piece of software or with plot_rank_histogram.m.

obs_diag can be configured to compare the ensemble estimates against the 'observation' copy or the 'truth' copy based on the setting of the use_zero_error_obs namelist variable.

The observation sequence files contain only the time of the observation, nothing of the assimilation interval, etc. - so it requires user guidance to declare what sort of temporal binning for the temporal evolution plots. I do a 'bunch' of arithmetic on the namelist times to convert them to a series of temporal bin edges that are used when traversing the observation sequence. The actual algorithm is that the user input for the start date and bin width set up a sequence that ends in one of two ways . . . the last time is reached or the number of bins has been reached.

obs_diag reads obs_seq.final files and calculates the following quantities (in no particular order) for an arbitrary number of regions and levels. obs_diag creates a netCDF file called obs_diag_output.nc. It is necessary to query the CopyMetaData variable to determine the storage order (i.e. "which copy is what?") if you want to use your own plotting routines.

ncdump -f F -v CopyMetaData obs_diag_output.nc

Nposs	The number of observations available to be assimilated.
Nused	The number of observations that were assimilated.
NbadUV	the number of velocity observations that had a matching component that was not assimilated;
NbadLV	the number of observations that were above or below the highest or lowest model level, respectively;
rmse	The root-mean-squared error (the horizontal wind components are also used to calculate the vector wind velocity and its RMS error).
bias	The simple sum of forecast - observation. The bias of the horizontal wind speed (not velocity) is also computed.
spread	The standard deviation of the univariate obs. DART does not exploit the bivariate nature of U,V winds and so the spread of the horizontal wind is defined as the sum of the spreads of the U and V components.
totalspread	The total standard deviation of the estimate. We pool the ensemble variance of the observation plus the observation error variance and take the square root.
NbadDARTQC	the number of observations that had a DART QC value (> 1 for a prior, > 3 for a posterior)
observation	the mean of the observation values
ens_mean	the ensemble mean of the model estimates of the observation values
N_trusted	the number of implicitly trusted observations, regardless of DART QC
N_DARTqc_0	the number of observations that had a DART QC value of 0
N_DARTqc_1	the number of observations that had a DART QC value of 1
N_DARTqc_2	the number of observations that had a DART QC value of 2
N_DARTqc_3	the number of observations that had a DART QC value of 3
N_DARTqc_4	the number of observations that had a DART QC value of 4
N_DARTqc_5	the number of observations that had a DART QC value of 5
N_DARTqc_6	the number of observations that had a DART QC value of 6
N_DARTqc_7	the number of observations that had a DART QC value of 7
N_DARTqc_8	the number of observations that had a DART QC value of 8

The temporal evolution of the above quantities for every observation type (RADIOSONDE_U_WIND_COMPONENT, AIRCRAFT_SPECIFIC_HUMIDITY, …) is recorded in the output netCDF file - `obs_diag_output.nc`. This netCDF file can then be loaded and displayed using the Matlab® scripts in `..../DART/diagnostics/matlab`. (which may depend on functions in `..../DART/matlab`). The temporal, geographic, and vertical binning are under namelist control. Temporal averages of the above quantities are also stored in the netCDF file. Normally, it is useful to skip the 'burn-in' period - the amount of time to skip is under namelist control.

The DART QC flag is intended to provide information about whether the observation was assimilated, evaluated only, whether the assimilation resulted in a 'good' observation, etc. *DART QC values lower than***2***indicate the prior and posteriors are OK.* DART QC values higher than **3** were **not** assimilated or evaluated. Here is the table that should explain things more fully:

DART QC flag value

meaning

0

observation assimilated

1

observation evaluated only (because of namelist settings)

2

assimilated, but the posterior forward operator failed

3

evaluated only, but the posterior forward operator failed

4

prior forward operator failed

5

not used because observation type not listed in namelist

6

rejected because incoming observation QC too large

7

rejected because of a failed outlier threshold test

8

vertical conversion failed

9+

reserved for future use

6.110.2 What is new in the Manhattan release

1. Support for DART QC = 8 (failed vertical conversion).
2. Simplified input file specification.
3. Removed `rat_cri` and `input_qc_threshold` from the namelists. They had been deprecated for quite some time.
4. Some of the internal variable names have been changed to make it easier to distinguish between variances and standard deviations.

6.110.3 What is new in the Lanai release

`obs_diag` has several improvements:

1. Improved vertical specification. Namelist variables `[h,p,m]level_edges` allow fine-grained control over the vertical binning. It is not allowed to specify both the edges and midpoints for the vertical bins.
2. Improved error-checking for input specification, particularly the vertical bins. Repeated values are squeezed out.
3. Support for 'trusted' observations. Trusted observation types may be specified in the namelist and all observations of that type will be counted in the statistics despite the DART QC code (as long as the forward observation operator succeeds). See namelist variable `trusted_obs`. For more details, see the section on Trusted observations.
4. Support for 'true' observations (i.e. from an OSSE). If the 'truth' copy of an observation is desired for comparison (instead of the default copy) the observation error variance is set to 0.0 and the statistics are calculated relative to the 'truth' copy (as opposed to the normal 'noisy' or 'observation' copy). See namelist variable `use_zero_error_obs`.
5. discontinued the use of `rat_cri` and `input_qc_threshold` namelist variables. Their functionality was replaced by the DART QC mechanism long ago.

6. The creation of the rank histogram (if possible) is now namelist-controlled by namelist variable `create_rank_histogram`.

6.110.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_diag_nml
   obs_sequence_name     = ''
   obs_sequence_list     = ''
   first_bin_center      =  2003, 1, 1, 0, 0, 0
   last_bin_center       =  2003, 1, 2, 0, 0, 0
   bin_separation        =     0, 0, 0, 6, 0, 0
   bin_width             =     0, 0, 0, 6, 0, 0
   time_to_skip          =     0, 0, 1, 0, 0, 0
   max_num_bins          = 1000
   plevel                = -888888.0
   hlevel                = -888888.0
   mlevel                = -888888
   plevel_edges          = -888888.0
   hlevel_edges          = -888888.0
   mlevel_edges          = -888888
   Nregions              = 0
   lonlim1               = -888888.0
   lonlim2               = -888888.0
   latlim1               = -888888.0
   latlim2               = -888888.0
   reg_names             = 'null'
   trusted_obs           = 'null'
   create_rank_histogram = .true.
   outliers_in_histogram = .false.
   use_zero_error_obs    = .false.
   verbose               = .false.
   /
```

The date-time integer arrays in this namelist have the form (YYYY, MM, DY, HR, MIN, SEC).
The allowable ranges for the region boundaries are: latitude [-90.,90], longitude [0.,Inf.]

You can only specify **either** `obs_sequence_name` **or** `obs_sequence_list` – not both. One of them has to be an empty string … i.e. `''`.

Item	Type	Description
obs_sequence_name	character(len=256), dimension(100)	An array of names of observation sequence files. These may be relative or absolute filenames. If this is set, `obs_sequence_list` must be set to ' ' (empty string).
obs_sequence_list	character(len=256)	Name of an ascii text file which contains a list of one or more observation sequence files, one per line. If this is specified, `obs_sequence_name` must be set to ' '. Can be created by any method, including sending the output of the 'ls' command to a file, a text editor, or another program. If this is set, `obs_sequence_name` must be set to ' ' (empty string).
first_bin_center	integer, dimension(6)	first timeslot of the first obs_seq.final file to process. The six integers are: year, month, day, hour, hour, minute, second, in that order. `obs_diag` has improved run-time output that reports the time and date of the first and last observations in every observation sequence file. Look for the string 'First observation date' in the logfile. If the `verbose` is 'true', it is also written to the screen.
last_bin_center	integer, dimension(6)	last timeslot of interest. (reminder: the last timeslot of day 1 is hour 0 of day 2) The six integers are: year, month, day, hour, hour, minute, second, in that order. This does not need to be exact, the values from `first_bin_center` and `bin_separation` are used to populate the time array and stop on or before the time defined by `last_bin_center`. See also `max_num_bins`.
bin_separation	integer, dimension(6)	Time between bin centers. The year and month values *must* be zero.
bin_width	integer, dimension(6)	Time span around bin centers in which obs will be compared. The year and month values *must* be zero. Frequently, but not required to be, the same as the values for bin_separation. 0
time_to_skip	integer, dimension(6)	Time span at the beginning to skip when calculating vertical profiles of rms error and bias. The year and month values *must* be zero. Useful because it takes some time for the assimilation to settle down from the climatological spread at the start. `time_to_skip` is an amount of time AFTER the first edge of the first bin.
max_num_bins	integer	This provides an alternative way to declare the `last_bin_center`. If `max_num_bins` is set to '10', only 10 timesteps will be output - provided `last_bin_center` is set to some later date.
plevel	real, dimension(50)	The midpoints defining the pressure levels for the vertical binning. There is no specification of bin width - a continuum is used. If a single midpoint value is entered, the bin edges are +/- 10% of the midpoint value. If you'd like to change that see the routine *Rmidpoints2edges()*. You may specify either `plevel` or `plevel_edges`, but not both.
plevel_edges	real, dimension(51)	The edges defining the pressure levels for the vertical binning. You may specify either `plevel` or `plevel_edges`, but not both.
hlevel	real, dimension(50)	Same, but for observations that have height(m) or depth(m) as the vertical coordinate.
hlevel_edges	real, dimension(51)	The edges defining the height (or depth) levels for the vertical binning. You may specify either `hlevel` or `hlevel_edges`, but not both.
mlevel	real, dimension(50)	Same, but for observations that have model level as the vertical coordinate.
mlevel_edges	real, dimension(51)	The edges defining the model levels for the vertical binning. You may specify either `mlevel` or `mlevel_edges`, but not both.
Nregions	integer	Number of regions of the globe for which obs space diagnostics are computed separately. Must be between [1,50]. If 50 is not enough, increase `obs_diag.f90MaxRegions` and recompile.
lonlim1	real, dimension(50)	Westernmost longitudes of each of the regions.
lonlim2	real, dimen-	Easternmost longitudes of each of the regions. *If any of these values is***less than***the westernmost values, it defines a region that spans the prime meridian.* e.g. a specification of `lonlim1`

6.110.5 Other modules used

```
obs_sequence_mod
obs_kind_mod
obs_def_mod (and possibly other obs_def_xxx mods)
assim_model_mod
random_seq_mod
model_mod
location_mod
types_mod
time_manager_mod
utilities_mod
sort_mod
```

6.110.6 Files

- `input.nml` is used for `obs_diag_nml`
- `obs_diag_output.nc` is the netCDF output file
- `dart_log.out` list directed output from the obs_diag.
- `LargeInnov.txt` contains the distance ratio histogram – useful for estimating the distribution of the magnitudes of the innovations.

Obs_diag may require a model input file from which to get grid information, metadata, and links to modules providing the models expected observations. It all depends on what's needed by the `model_mod.f90`

Discussion of obs_diag_output.nc

Every observation type encountered in the observation sequence file is tracked separately, and aggregated into temporal and 3D spatial bins. There are two main efforts to this program. One is to track the temporal evolution of any of the quantities available in the netCDF file for any possible observation type:

ncdump -v CopyMetaData,ObservationTypes obs_diag_output.nc

The other is to explore the vertical profile of a particular observation kind. By default, each observation kind has a 'guess/prior' value and an 'analysis/posterior' value - which shed some insight into the innovations.

Temporal evolution

The `obs_diag_output.nc` output file has all the metadata I could think of, as well as separate variables for every observation type in the observation sequence file. Furthermore, there is a separate variable for the 'guess/prior' and 'analysis/posterior' estimate of the observation. To distinguish between the two, a suffix is appended to the variable name. An example seems appropriate:

```
...
char CopyMetaData(copy, stringlength) ;
        CopyMetaData:long_name = "quantity names" ;
char ObservationTypes(obstypes, stringlength) ;
        ObservationTypes:long_name = "DART observation types" ;
        ObservationTypes:comment = "table relating integer to observation type string
↪" ;
float RADIOSONDE_U_WIND_COMPONENT_guess(time, copy, plevel, region) ;
        RADIOSONDE_U_WIND_COMPONENT_guess:_FillValue = -888888.f ;
```

(continues on next page)

(continued from previous page)

```
        RADIOSONDE_U_WIND_COMPONENT_guess:missing_value = -888888.f ;
float RADIOSONDE_V_WIND_COMPONENT_guess(time, copy, plevel, region) ;
        RADIOSONDE_V_WIND_COMPONENT_guess:_FillValue = -888888.f ;
        RADIOSONDE_V_WIND_COMPONENT_guess:missing_value = -888888.f ;
...
float MARINE_SFC_ALTIMETER_guess(time, copy, surface, region) ;
        MARINE_SFC_ALTIMETER_guess:_FillValue = -888888.f ;
        MARINE_SFC_ALTIMETER_guess:missing_value = -888888.f ;
...
float RADIOSONDE_WIND_VELOCITY_guess(time, copy, plevel, region) ;
        RADIOSONDE_WIND_VELOCITY_guess:_FillValue = -888888.f ;
        RADIOSONDE_WIND_VELOCITY_guess:missing_value = -888888.f ;
...
float RADIOSONDE_U_WIND_COMPONENT_analy(time, copy, plevel, region) ;
        RADIOSONDE_U_WIND_COMPONENT_analy:_FillValue = -888888.f ;
        RADIOSONDE_U_WIND_COMPONENT_analy:missing_value = -888888.f ;
float RADIOSONDE_V_WIND_COMPONENT_analy(time, copy, plevel, region) ;
        RADIOSONDE_V_WIND_COMPONENT_analy:_FillValue = -888888.f ;
        RADIOSONDE_V_WIND_COMPONENT_analy:missing_value = -888888.f ;
...
```

There are several things to note:

1. the 'WIND_VELOCITY' component is nowhere 'near' the corresponding U,V components.
2. all of the 'guess' variables come before the matching 'analy' variables.
3. surface variables (i.e. MARINE_SFC_ALTIMETER have a coordinate called 'surface' as opposed to 'plevel' for the others in this example).

Vertical profiles

Believe it or not, there are another set of netCDF variables specifically for the vertical profiles, essentially duplicating the previous variables but **without the 'time' dimension**. These are distinguished by the suffix added to the observation kind - 'VPguess' and 'VPanaly' - 'VP' for Vertical Profile.

```
...
float SAT_WIND_VELOCITY_VPguess(copy, plevel, region) ;
        SAT_WIND_VELOCITY_VPguess:_FillValue = -888888.f ;
        SAT_WIND_VELOCITY_VPguess:missing_value = -888888.f ;
...
float RADIOSONDE_U_WIND_COMPONENT_VPanaly(copy, plevel, region) ;
        RADIOSONDE_U_WIND_COMPONENT_VPanaly:_FillValue = -888888.f ;
        RADIOSONDE_U_WIND_COMPONENT_VPanaly:missing_value = -888888.f ;
...
```

Observations flagged as 'surface' do not participate in the vertical profiles (Because surface variables cannot exist on any other level, there's not much to plot!). Observations on the lowest level DO participate. There's a difference!

Rank histograms

If it is possible to calculate a rank histogram, there will also be :

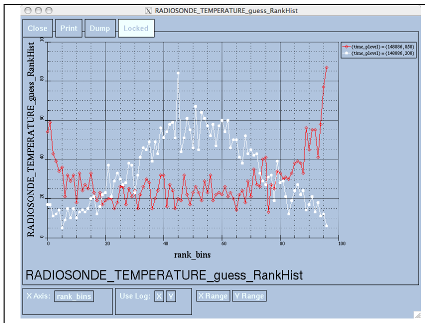
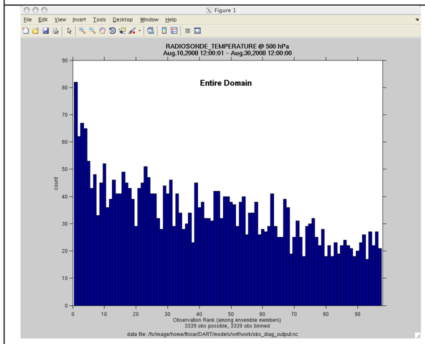
```
...
int RADIOSONDE_U_WIND_COMPONENT_guess_RankHi(time, rank_bins, plevel, region) ;
...
int RADIOSONDE_V_WIND_COMPONENT_guess_RankHi(time, rank_bins, plevel, region) ;
...
int MARINE_SFC_ALTIMETER_guess_RankHist(time, rank_bins, surface, region) ;
...
```

as well as some global attributes. The attributes reflect the namelist settings and can be used by plotting routines to provide additional annotation for the histogram.

```
:DART_QCs_in_histogram = 0, 1, 2, 3, 7 ;
:outliers_in_histogram = "TRUE" ;
```

Please note:

1. netCDF restricts variable names to 40 characters, so '_Rank_Hist' may be truncated.
2. It is sufficiently vague to try to calculate a rank histogram for a velocity derived from the assimilation of U,V components such that NO rank histogram is created for velocity. A run-time log message will inform as to which variables are NOT having a rank histogram variable preserved in the `obs_diag_output.nc` file - IFF it is possible to calculate a rank histogram in the first place.

	Instructions for viewing the rank histogram with ncview.
	Instructions for viewing the rank histogram with Matlab.

RADIOSONDE_TEMPERATURE_guess_RankHist
Close
Print
Dump
Locked
RADIOSONDE_TEMPERATURE_guess_RankHist
(time, plevel) = (148886, 850)
(time, plevel) = (148886, 200)
0
20
40
60
80
100
rank_bins
RADIOSONDE_TEMPERATURE_guess_RankHist
X Axis:
rank_bins
Use Log:
X
Y
X Range
Y Range

Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help
RADIOSONDE_TEMPERATURE @ 500 hPa
Aug.10,2008 12:00:01 -- Aug.30,2008 12:00:00
Entire Domain
count
0
10
20
30
40
50
60
70
80
90
Observation Rank (among ensemble members)
3339 obs possible, 3339 obs binned
data file: /fs/image/home/thoar/DART/models/wrf/work/obs_diag_output.nc

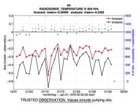
“trusted” observation types

This needs to be stated up front: `obs_diag` is a post-processor; it cannot influence the assimilation. One interpretation of a TRUSTED observation is that the assimilation should **always** use the observation, even if it is far from the ensemble. At present (23 Feb 2015), the filter in DART does not forcibly assimilate any one observation and selectively assimilate the others. Still, it is useful to explore the results using a set of ‘trusted type’ observations, whether they were assimilated, evaluated, or rejected by the outlier threshhold. This is the important distinction. The diagnostics can be calculated differently for each *observation type*.

The normal diagnostics calculate the metrics (rmse, bias, etc.) only for the ‘good’ observations - those that were assimilated or evaluated. The `outlier_threshold` essentially defines what observations are considered too far from the ensemble **prior** to be useful. These observations get a DART QC of 7 and are not assimilated. The observations with a DART QC of 7 do not contribute the the metrics being calculated. Similarly, if the forward observation operator fails, these observations cannot contribute. When the operator fails, the ‘expected’ observation value is ‘MISSING’, and there is no ensemble mean or spread.

‘Trusted type’ observation metrics are calculated using all the observations that were assimilated or evaluated **AND** the observations that were rejected by the outlier threshhold. `obs_diag` can post-process the DART QC and calculate the metrics appropriately for **observation types** listed in the `trusted_obs` namelist variable. If there are trusted observation types specified for `obs_diag`, the `obs_diag_output.nc` has global metadata to indicate that a different set of criteria were used to calculate the metrics. The individual variables also have an extra attribute. In the following output, `input.nml:obs_diag_nml:trusted_obs` was set: `trusted_obs = 'RADIOSONDE_TEMPERATURE', 'RADIOSONDE_U_WIND_COMPONENT'`

```
  ...
        float RADIOSONDE_U_WIND_COMPONENT_guess(time, copy, plevel, region) ;
                RADIOSONDE_U_WIND_COMPONENT_guess:_FillValue = -888888.f ;
                RADIOSONDE_U_WIND_COMPONENT_guess:missing_value = -888888.f ;
                RADIOSONDE_U_WIND_COMPONENT_guess:TRUSTED = "TRUE" ;
        float RADIOSONDE_V_WIND_COMPONENT_guess(time, copy, plevel, region) ;
                RADIOSONDE_V_WIND_COMPONENT_guess:_FillValue = -888888.f ;
                RADIOSONDE_V_WIND_COMPONENT_guess:missing_value = -888888.f ;
  ...
// global attributes:
  ...
                :trusted_obs_01 = "RADIOSONDE_TEMPERATURE" ;
                :trusted_obs_02 = "RADIOSONDE_U_WIND_COMPONENT" ;
                :obs_seq_file_001 = "cam_obs_seq.1978-01-01-00000.final" ;
                :obs_seq_file_002 = "cam_obs_seq.1978-01-02-00000.final" ;
                :obs_seq_file_003 = "cam_obs_seq.1978-01-03-00000.final" ;
  ...
                :MARINE_SFC_ALTIMETER = 7 ;
                :LAND_SFC_ALTIMETER = 8 ;
                :RADIOSONDE_U_WIND_COMPONENT--TRUSTED = 10 ;
                :RADIOSONDE_V_WIND_COMPONENT = 11 ;
                :RADIOSONDE_TEMPERATURE--TRUSTED = 14 ;
                :RADIOSONDE_SPECIFIC_HUMIDITY = 15 ;
                :AIRCRAFT_U_WIND_COMPONENT = 21 ;
  ...
```

The Matlab scripts try to ensure that the trusted observation graphics clarify that the metrics plotted are somehow 'different' than the normal processing stream. Some text is added to indicate that the values include the outlying observations. **IMPORTANT:** The interpretation of the number of observations 'possible' and 'used' still reflects what was used **in the assimilation!** The number of observations rejected by the outlier threshhold is not explicilty plotted. To reinforce this, the text for the observation axis on all graphics has been changed to `"o=possible, *=assimilated"`. In short, the distance between the number of observations possible and the number assimilated still reflects the number of observations rejected by the outlier threshhold and the number of failed forward observation operators.	

There is ONE ambiguous case for trusted observations. There may be instances in which the observation fails the outlier threshhold test (which is based on the prior) and the posterior forward operator fails. DART does not have a QC that explicilty covers this case. The current logic in `obs_diag` correctly handles these cases **except** when trying to use 'trusted' observations. There is a section of code in `obs_diag` that may be enabled if you are encountering this ambiguous case. As `obs_diag` runs, a warning message is issued and a summary count is printed if the ambiguous case is encountered. What normally happens is that if that specific observation type is trusted, the posterior values include a MISSING value in the calculation which makes them inaccurate. If the block of code is enabled, the DART QC is recast as the PRIOR forward observation operator fails. This is technically incorrect, but for the case of trusted observations, it results in only calculating statistics for trusted observations that have a useful prior and posterior. **This should not be used unless you are willing to intentionally disregard 'trusted' observations that were rejected by the outlier threshhold.** Since the whole point of a trusted observation is to *include* observations potentially rejected by the outlier threshhold, you see the problem. Some people like to compare the posteriors. *THAT* can be the problem.

```
if ((qc_integer == 7) .and. (abs(posterior_mean(1) - MISSING_R8) < 1.0_r8)) then
            write(string1,*)'WARNING ambiguous case for obs index ',obsindex
            string2 = 'obs failed outlier threshhold AND posterior operator failed.'
            string3 = 'Counting as a Prior QC == 7, Posterior QC == 4.'
            if (trusted) then
! COMMENT      string3 = 'WARNING changing DART QC from 7 to 4'
! COMMENT      qc_integer = 4
            endif
            call error_handler(E_MSG,'obs_diag',string1,text2=string2,text3=string3)
            num_ambiguous = num_ambiguous + 1
         endif
```

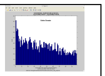
6.110.7 Usage

`obs_diag` is built in .../DART/models/*your_model*/work, in the same way as the other DART components.

Multiple observation sequence files

There are two ways to specify input files for `obs_diag`. You can either specify the name of a file containing a list of files (in `obs_sequence_list`), or you may specify a list of files via `obs_sequence_name`.

Example: observation sequence files spanning 30 days

In this example, we will be accumulating metrics for 30 days over the entire globe. The obs_diag_output.nc file will have exactly ONE timestep in it (so it won't be much use for the plot_evolution functions) - but the plot_profile functions and the plot_rank_histogram function will be used to explore the assimilation. By way of an example, we will NOT be using outlier observations in the rank histogram. Lets presume that all your obs_seq.final files are in alphabetically-nice directories:	

```
/Exp1/Dir01/obs_seq.final
/Exp1/Dir02/obs_seq.final
/Exp1/Dir03/obs_seq.final
...
/Exp1/Dir99/obs_seq.final
```

The first step is to create a file containing the list of observation sequence files you want to use. This can be done with the unix command 'ls' with the -1 option (that's a number one) to put one file per line.

ls -1 /Exp1/Dir*/obs_seq.final > obs_file_list.txt

It is necessary to turn on the verbose option to check the first/last times that will be used for the histogram. Then, the namelist settings for 2008 07 31 12Z through 2008 08 30 12Z are:

```
&obs_diag_nml
   obs_sequence_name      = ''
   obs_sequence_list      = 'obs_file_list.txt'
   first_bin_center       =  2008, 8,15,12, 0, 0
   last_bin_center        =  2008, 8,15,12, 0, 0
   bin_separation         =     0, 0,30, 0, 0, 0
   bin_width              =     0, 0,30, 0, 0, 0
   time_to_skip           =     0, 0, 0, 0, 0, 0
   max_num_bins           = 1000
   Nregions               = 1
   lonlim1                =   0.0
   lonlim2                = 360.0
   latlim1                = -90.0
   latlim2                =  90.0
   reg_names              = 'Entire Domain'
   create_rank_histogram = .true.
   outliers_in_histogram = .false.
   verbose                = .true.
   /
```

then, simply run obs_diag in the usual manner - you may want to save the run-time output to a file. Here is a portion of the run-time output:

```
...
Region   1 Entire Domain                    (WESN):     0.0000    360.0000    -90.0000  ␣
↪ 90.0000
 Requesting            1  assimilation periods.

epoch       1  start day=148865, sec=43201
epoch       1 center day=148880, sec=43200
epoch       1    end day=148895, sec=43200
epoch       1  start 2008 Jul 31 12:00:01
```

(continues on next page)

(continued from previous page)

```
epoch      1 center 2008 Aug 15 12:00:00
epoch      1    end 2008 Aug 30 12:00:00
...
MARINE_SFC_HORIZONTAL_WIND_guess_RankHis has            0 "rank"able observations.
SAT_HORIZONTAL_WIND_guess_RankHist       has            0 "rank"able observations.
...
```

Discussion: It should be pretty clear that there is exactly 1 assimilation period, it may surprise you that the start is 1 second past 12Z. This is deliberate and reflects the DART convention of starting intervals 1 second after the end of the previous interval. The times in the netCDF variables reflect the defined start/stop of the period, regardless of the time of the first/last observation.

Please note that none of the 'horizontal_wind' variables will have a rank histogram, so they are not written to the netCDF file. ANY variable that does not have a rank histogram with some observations will NOT have a rank histogram variable in the netCDF file.

Now that you have the `obs_diag_output.nc`, you can explore it with plot_profile.m, plot_bias_xxx_profile.m, or plot_rmse_xxx_profile.m, and look at the rank histograms with ncview or `plot_rank_histogram.m`.

6.110.8 References

1. none

6.110.9 Private components

N/A

6.111 PROGRAM `fill_inflation_restart`

6.111.1 Overview

Utility program to create inflation restart files with constant values.

These files can be used as input for the first step of a multi-step assimilation when adaptive inflation is being used. This allows the namelist items `inf_initial_from_restart` and `inf_sd_initial_from_restart` in the `&filter_nml` namelist to be `.TRUE.` for all steps of the assimilation including the very first one. (These items control whether inflation values are read from an input file or read from constants in the namelist.)

Adaptive inflation restart files are written at the end of a `filter` run and are needed as input for the next timestep. This program creates files that can be used for the initial run of filter when no inflation restart files have been created by filter but are required to be read as input.

This program reads the inflation values to use from the `&fill_inflation_restart_nml` namelist for setting the prior inflation mean and standard deviation, and/or the posterior inflation mean and standard deviation. It does not use the inflation values in the `&filter` namelist.

This program uses the information from the model_mod code to determine the number of items in the state vector. It must be compiled with the right model's model_mod, and if the items in the state vector are selectable by namelist options, the namelist when running this program must match exactly the namelist used during the assimilation run.

It creates files with names consistent with the input names expected by filter:

```
input_priorinf_mean.nc
input_priorinf_sd.nc
input_postinf_mean.nc
input_postinf_sd.nc
```

An older (and deprecated) alternative to running `fill_inflation_restart` is to create inflation netcdf files by using one of the NCO utilities like “`ncap2`” on a copy of another restart file to set the initial inflation mean, and another for the initial inflation standard deviation. Inflation mean and sd values look exactly like restart values, arranged by variable type like T, U, V, etc.

Depending on your version of the NCO utilities, you can use `ncap2` to set the T,U and V inf values using one of two syntaxes:

```
ncap2 -s 'T=1.0;U=1.0;V=1.0' wrfinput_d01 input_priorinf_mean.nc
ncap2 -s 'T=0.6;U=0.6;V=0.6' wrfinput_d01 input_priorinf_sd.nc
-or-
ncap2 -s 'T(:,:,:)=1.0;U(:,:,:)=1.0;V(:,:,:)=1.0' wrfinput_d01 input_priorinf_mean.nc
ncap2 -s 'T(:,:,:)=0.6;U(:,:,:)=0.6;V(:,:,:)=0.6' wrfinput_d01 input_priorinf_sd.nc
```

Some versions of the NCO utilities change the full 3D arrays into a single scalar. If that’s your result (check your output with `ncdump -h`) use the alternate syntax or a more recent version of the NCO tools.

6.111.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand ‘&’ and terminate with a slash ‘/’. Character strings that contain a ‘/’ must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&fill_inflation_restart_nml

   write_prior_inf    = .FALSE.
   prior_inf_mean     = -88888.8888
   prior_inf_sd       = -88888.8888

   write_post_inf     = .FALSE.
   post_inf_mean      = -88888.8888
   post_inf_sd        = -88888.8888

   single_file        = .FALSE.
   input_state_files  = ''
   verbose            = .FALSE.
/
```

The namelist controls which files are created and what values are written to the restart files.

Item	Type	Description
write_prior_inf	logical	Setting this to .TRUE. writes both the prior inflation mean and standard deviation files: `input_priorinf_mean.nc`, `input_priorinf_sd.nc`.
prior_inf_mean	real(r8)	Prior inflation mean value.
prior_inf_sd	real(r8)	Prior inflation standard deviation value.
write_post_inf	logical	Setting this to .TRUE. writes both the posterior inflation mean and standard deviation files `input_postinf_mean.nc`, `input_postinf_sd.nc`.
post_inf_mean	real(r8)	Posterior inflation mean value.
post_inf_sd	real(r8)	Posterior inflation standard deviation value.
single_file	logical	Currently not supported, but would be used in the case where you have a single restart file that contains all of the ensemble members. Must be .false.
input_state_files	character(:)	List one per domain, to be used as a template for the output inflation files.
verbose	logical	Setting this to .TRUE. gives more output, and is generally used for debugging

Here is an example of a typical namelist for `fill_inflation_restart` :

```
&fill_inflation_restart_nml

   write_prior_inf    = .TRUE.
   prior_inf_mean     = 1.01
   prior_inf_sd       = 0.6

   write_post_inf     = .FALSE.
   post_inf_mean      = 1.0
   post_inf_sd        = 0.6

   single_file        = .FALSE.
   input_state_files  = ''
   verbose            = .FALSE.
/
```

6.111.3 Files

Creates:

```
input_priorinf_mean.nc
input_priorinf_sd.nc
input_postinf_mean.nc
input_postinf_sd.nc
```

based on the template file from the specific model this code is compiled for.

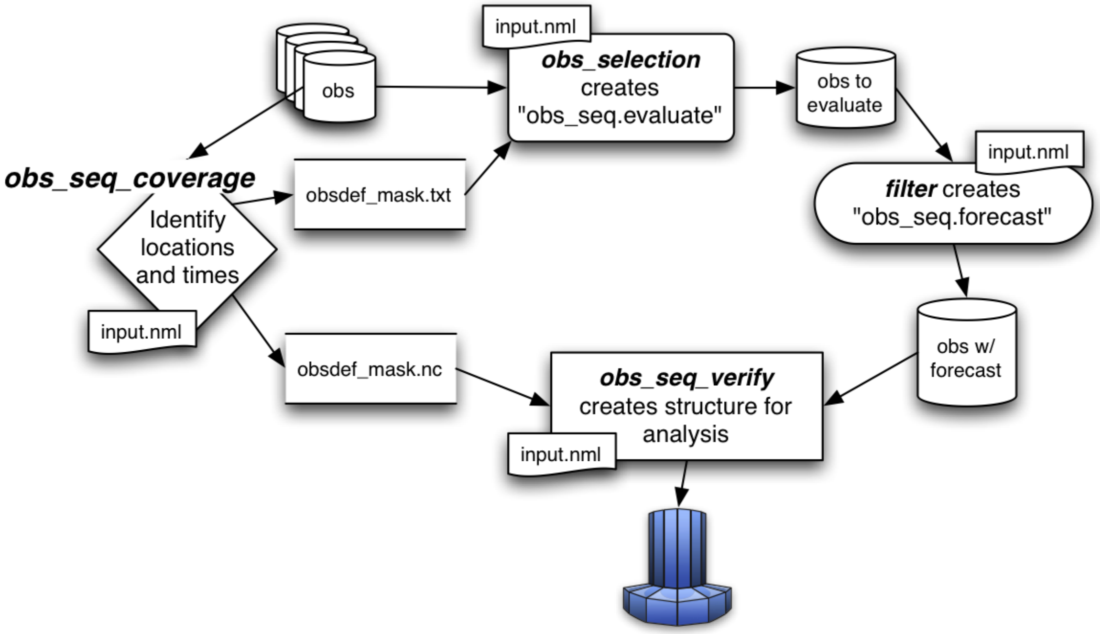
6.111.4 References

- none

6.112 program `obs_seq_coverage`

6.112.1 Overview

`obs_seq_coverage` queries a set of observation sequence files to determine which observation locations report frequently enough to be useful for a verification study. The big picture is to be able to pare down a large set of observations into a compact observation sequence file to run through *PROGRAM filter* with all of the intended observation types flagged as *evaluate_only*. DART's forward operators then get applied and all the forecasts are preserved in a standard `obs_seq.final` file - perhaps more appropriately called `obs_seq.forecast`! Paring down the input observation sequence file cuts down on the unnecessary application of the forward operator to create observation copies that will not be used anyway . . .



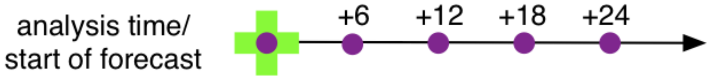
`obs_seq_coverage` results in two output files:

- `obsdef_mask.txt` contains the list of observation definitions (but not the observations themselves) that are desired. The observation definitions include the locations and times for each of the desired observation types. This file is read by *program obs_selection* and combined with the raw observation sequence files to create the observation sequence file appropriate for use in a forecast.
- `obsdef_mask.nc` contains information needed to be able to plot the times and locations of the observations in a manner to help explore the design of the verification locations/network. `obsdef_mask.nc` is *required* by *program obs_seq_verify*, the program that reorders the observations into a structure that makes it easy to calculate statistics like ROC, etc.

The following section explains the strategy and requirements for determining what observations will be used to verify a forecast. Since it is 'standard practice' to make several forecasts to build statistical strength, it is important to use the SAME set of observation locations for all the forecasts that will be verified together. To make the discussion easier, let's define the *verification network* as the set of locations and times for a particular observation type.

The entire discussion about finding locations that are repeatedly observed through time boils down to the simple statement that if the observation is within about 500cm of a previous observation, they are treated as co-located observations. For some very high resolution applications, this may be insufficient, but there it is. For observations at pressure levels, see the Word about vertical levels.

The only complicated part of determining the verification network is the temporal component. The initial time (usually an *analysis time* from a previous assimilation), the *verification interval*, and the *forecast length* completely specify the temporal aspect of a forecast. The following example has a verification interval of 6 hours and a forecast length of 24 hours. We adopt the convention of also including the initial conditions (a "nowcast") in the "forecast", so there are 5 times of interest - which we will call *verification times* and are represented by ●. The candidate observation sequence files are scanned to select all the observations that are **closest** to the verification times. The difference in time between the "nowcast" and the "forecast" is the *forecast lead*.

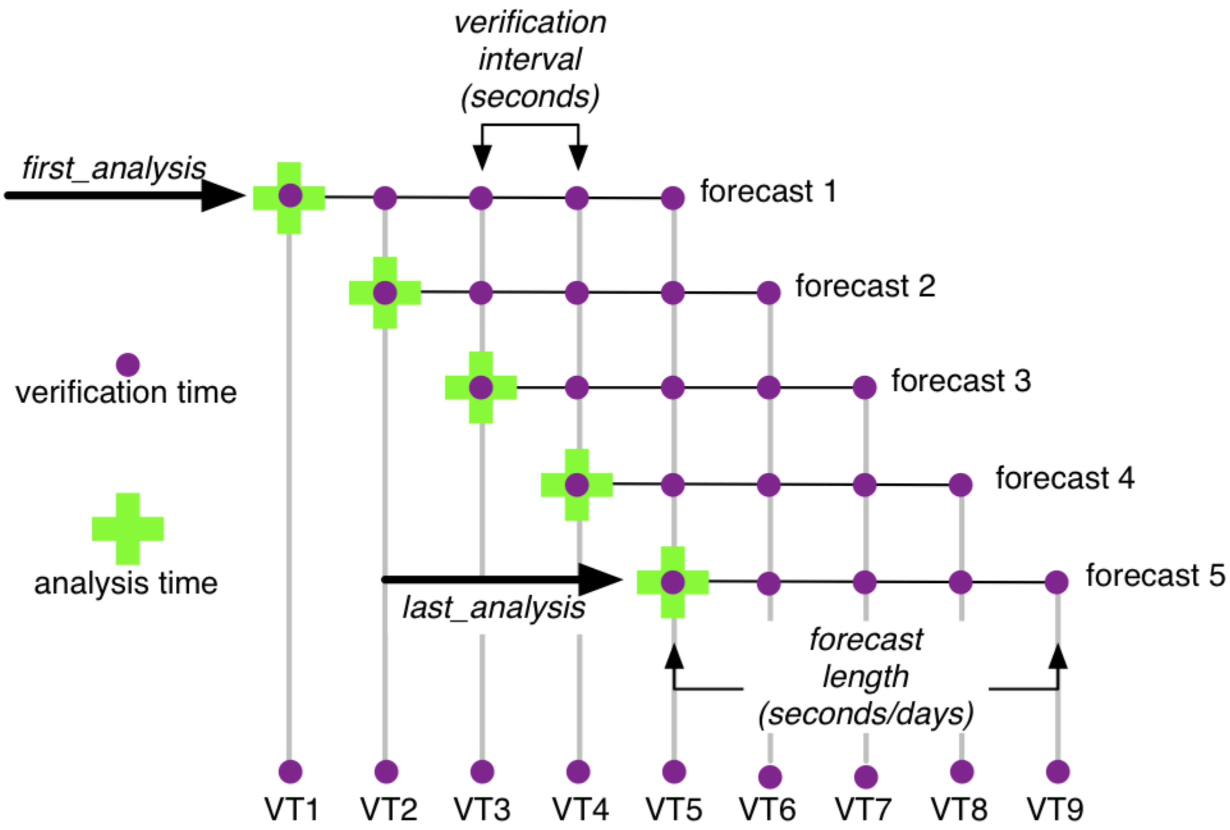


So - that is simple enough if there is only one forecast, but this is rarely the case. Let's say we have a second forecast. Ideally, we'd like to verify at exactly the same locations and forecast leads - otherwise we're not really comparing the same things. If the second verification network happens to be at locations that are easy to predict, we're comparing apples and oranges. The *fair* way to proceed is to determine the verification network that is the same for all forecasts. This generally results in a pretty small set of observations - a problem we will deal with later.

The diagram below illustrates the logic behind determining the list of verification times for a pretty common scenario: a 24-hour forecast with a forecast lead of 6 hours, repeated the next day. The *first_analysis* is at VT1 - let's call it 00Z day 1. We need to have observations available at:

VT1 (00Z day1), VT2 (06Z day1), VT3 (12Z day1), VT4 (18Z day1), and VT5 (24Z day1 / 00Z day2). The *last_analysis* starts at VT5 00Z day 2 and must verify at

VT5 (00Z day2), VT6 (06Z day2), VT7 (12Z day2), VT8 (18Z day2), and VT9 (24Z day2 / 00Z day3).



Note that, if you wanted to, you could launch forecasts at VT2, VT3, and VT4 without adding extra constraints on the verification network. `obs_seq_coverage` simply provides these possible forecasts "for free", there is no assumption about **needing** them. We will use the variable *verification_times* to describe the complete set of times for all possible forecasts. In our example above, there are 5 possible forecasts, each forecast consisting of 5 verification times (the analysis time and the 4 forecast lead times). As such, there are 9 unique verification times.

Note that no attempt is made at checking the QC value of the candidate observations. One of the common problems is that the region definition does not mesh particularly well with the model domain and the DART forward operator fails because it would have to extrapolate (which is not allowed). Without checking the QC value, this can mean there are a lot of 'false positives'; observations that seemingly could be used to validate, but are actually just outside the model domain. I'm working on that . . . .

The USAGE section has more on the actual use of `obs_seq_coverage`.

6.112.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_seq_coverage_nml
   obs_sequences       = ''
   obs_sequence_list   = ''
   obs_of_interest     = ''
   textfile_out        = 'obsdef_mask.txt'
   netcdf_out          = 'obsdef_mask.nc'
   calendar            = 'Gregorian'
   first_analysis      =  2003, 1, 1, 0, 0, 0
```

(continues on next page)

(continued from previous page)

```
   last_analysis     =  2003, 1, 2, 0, 0, 0
   forecast_length_days          = 1
   forecast_length_seconds       = 0
   verification_interval_seconds = 21600
   temporal_coverage_percent     = 100.0
   lonlim1                       =  -888888.0
   lonlim2                       =  -888888.0
   latlim1                       =  -888888.0
   latlim2                       =  -888888.0
   verbose                       = .false.
   debug                         = .false.
  /
```

Note that -888888.0 is not a useful number. To use the defaults delete these lines from the namelist, or set them to 0.0, 360.0 and -90.0, 90.0.

The date-time integer arrays in this namelist have the form (YYYY, MM, DD, HR, MIN, SEC).

The allowable ranges for the region boundaries are: latitude [-90.,90], longitude [0.,Inf.]

You can specify **either** *obs_sequences* **or** *obs_sequence_list* – not both. One of them has to be an empty string … i.e. ‘’.

Item	Type	Description
obs_sequences	character(len=256)	Name of the observation sequence file(s). This may be a relative or absolute filename. If the filename contains a '/', the filename is considered to be comprised of everything to the right, and a directory structure to the left. The directory structure is then queried to see if it can be incremented to handle a sequence of observation files. The default behavior of `obs_seq_coverage` is to look for additional files to include until the files are exhausted or an `obs_seq.final` file is found that contains observations beyond the timeframe of interest. e.g. 'obsdir_001/obs_seq.final' will cause `obs_seq_coverage` to look for 'obsdir_002/obs_seq.final', and so on. If this is set, *obs_sequence_list* must be set to ' '.
obs_sequence_list	character(len=256)	Name of an ascii text file which contains a list of one or more observation sequence files, one per line. If this is specified, *obs_sequences* must be set to ' '. Can be created by any method, including sending the output of the 'ls' command to a file, a text editor, or another program.
obs_of_interest	character(len=32), dimension(:)	These are the observation types that will be verified. It is an array of character strings that must match the standard DART observation types. Simply add as many or as few observation types as you need. Could be 'METAR_U_10_METER_WIND', 'METAR_V_10_METER_WIND',..., for example.
textfile_out	character(len=256)	The name of the file that will contain the observation definitions of the verfication observations. Only the metadata from the observations (location, time, obs_type) are preserved in this file. They are in no particular order. *program obs_selection* will use this file as a 'mask' to extract the real observations from the candidate observation sequence files.
netcdf_out	character(len=256)	The name of the file that will contain the observation definitions of the unique locations that match **any** of the verification times. This file is used in conjunction with *program obs_seq_verify* to reorder the `obs_seq.forecast` into a structure that will facilitate calculat-

For example:

```
&obs_seq_coverage_nml
   obs_sequences     = ''
   obs_sequence_list = 'obs_coverage_list.txt'
   obs_of_interest   = 'METAR_U_10_METER_WIND',
                       'METAR_V_10_METER_WIND'
   textfile_out      = 'obsdef_mask.txt'
   netcdf_out        = 'obsdef_mask.nc'
   calendar          = 'Gregorian'
   first_analysis    =  2003, 1, 1, 0, 0, 0
   last_analysis     =  2003, 1, 2, 0, 0, 0
   forecast_length_days           = 1
   forecast_length_seconds        = 0
   verification_interval_seconds = 21600
   temporal_coverage_percent      = 100.0
   lonlim1     =    0.0
   lonlim2     =  360.0
   latlim1     =  -90.0
   latlim2     =   90.0
   verbose     = .false.
   /
```

6.112.3 Other modules used

```
assim_model_mod
types_mod
location_mod
model_mod
null_mpi_utilities_mod
obs_def_mod
obs_kind_mod
obs_sequence_mod
random_seq_mod
time_manager_mod
utilities_mod
```

6.112.4 Files

- `input.nml` is used for *obs_seq_coverage_nml*
- A text file containing the metadata for the observations to be used for forecast evaluation is created. This file is subsequently required by *program obs_selection* to subset the set of input observation sequence files into a single observation sequence file (`obs_seq.evaluate`) for the forecast step. (`obsdef_mask.txt` is the default name)
- A netCDF file containing the metadata for a much larger set of observations that may be used is created. This file is subsequently required by *program obs_seq_coverage* to define the desired times and locations for the verification. (`obsdef_mask.nc` is the default name)

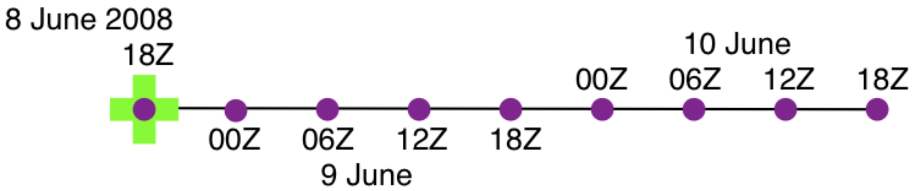
6.112.5 Usage

`obs_seq_coverage` is built in . . . /DART/models/*your_model*/work, in the same way as the other DART components.

There is no requirement on the reporting time/frequence of the candidate voxels. Once the verification times have been defined, the observation **closest in time** to the verification time is selected, the others are ignored. Only observations within half the verification interval are eligible to be considered "close".

A word about vertical levels. If the desired observation type has UNDEFINED or SURFACE for the vertical coordinate system, there is no concern about trying to match the vertical. If the desired observation types use PRESSURE; the following 14 levels are used as the standard levels: 1000, 925, 850, 700, 500, 400, 300, 250, 200, 150, 100, 70, 50, 10 (all hPa). **No other vertical coordinate system is supported.**

Example: a single 48-hour forecast that is evaluated every 6 hours



In this example, we are generating an `obsdef_mask.txt` file for a single forecast. All the required input observation sequence filenames will be contained in a file referenced by the *obs_sequence_list* variable. We'll also restrict the observations to a specific rectangular (in Lat/Lon) region at a particular level. It is convenient to turn on the verbose option the first time to get a feel for the logic. Here are the namelist settings if you want to verify the METAR_U_10_METER_WIND and METAR_V_10_METER_WIND observations over the entire globe every 6 hours for 2 days starting 18Z 8 Jun 2008:

```
&obs_seq_coverage_nml
   obs_sequences       = ''
   obs_sequence_list   = 'obs_file_list.txt'
   obs_of_interest     = 'METAR_U_10_METER_WIND',
                         'METAR_V_10_METER_WIND'
   textfile_out        = 'obsdef_mask.txt'
   netcdf_out          = 'obsdef_mask.nc'
   calendar            = 'Gregorian'
   first_analysis      =  2008, 6, 8, 18, 0, 0
   last_analysis       =  2008, 6, 8, 18, 0, 0
   forecast_length_days           = 2
   forecast_length_seconds        = 0
   verification_interval_seconds = 21600
   temporal_coverage_percent      = 100.0
   lonlim1             =    0.0
   lonlim2             =  360.0
   latlim1             =  -90.0
   latlim2             =   90.0
   verbose             = .true.
   /
```

The first step is to create a file containing the list of observation sequence files you want to use. This can be done with the unix command 'ls' with the -1 option (that's a number one) to put one file per line, particularly if the files are organized in a nice fashion. If your observation sequence are organized like this:

```
/Exp1/Dir20080101/obs_seq.final
/Exp1/Dir20080102/obs_seq.final
/Exp1/Dir20080103/obs_seq.final
...
/Exp1/Dir20081231/obs_seq.final
```

then

ls -1 /Exp1/Dir*/obs_seq.final > obs_file_list.txt

creates the desired file. Then, simply run `obs_seq_coverage` - you may want to save the run-time output to a file. It is convenient to turn on the verbose option the first time. Here is a portion of the run-time output:

```
[thoar@mirage2 work]$ ./obs_seq_coverage | & tee my.log
 Starting program obs_seq_coverage
 Initializing the utilities module.
 Trying to log to unit           10
 Trying to open file dart_log.out

 --------------------------------------
 Starting ... at YYYY MM DD HH MM SS =
                 2011  2 22 13 15  2
 Program obs_seq_coverage
 --------------------------------------

 set_nml_output Echo NML values to log file only
 Trying to open namelist log dart_log.nml
 location_mod: Ignoring vertical when computing distances; horizontal only
 ------------------------------------------------------


 -------------- ASSIMILATE_THESE_OBS_TYPES --------------
 RADIOSONDE_TEMPERATURE
 RADIOSONDE_U_WIND_COMPONENT
 RADIOSONDE_V_WIND_COMPONENT
 SAT_U_WIND_COMPONENT
 SAT_V_WIND_COMPONENT
 -------------- EVALUATE_THESE_OBS_TYPES --------------
 RADIOSONDE_SPECIFIC_HUMIDITY
 ------------------------------------------------------

 METAR_U_10_METER_WIND is type           36
 METAR_V_10_METER_WIND is type           37

 There are            9  verification times per forecast.
 There are            1  supported forecasts.
 There are            9  total times we need observations.

 At least           9  observations times are required at:
 verification #            1 at 2008 Jun 08 18:00:00
 verification #            2 at 2008 Jun 09 00:00:00
 verification #            3 at 2008 Jun 09 06:00:00
 verification #            4 at 2008 Jun 09 12:00:00
 verification #            5 at 2008 Jun 09 18:00:00
 verification #            6 at 2008 Jun 10 00:00:00
 verification #            7 at 2008 Jun 10 06:00:00
 verification #            8 at 2008 Jun 10 12:00:00
 verification #            9 at 2008 Jun 10 18:00:00
```

(continues on next page)

(continued from previous page)

```
 obs_seq_coverage  opening obs_seq.final.2008060818
 QC index           1  NCEP QC index
 QC index           2  DART quality control

First observation time day=148812, sec=64380
First observation date 2008 Jun 08 17:53:00
 Processing obs        10000  of        84691
 Processing obs        20000  of        84691
 Processing obs        30000  of        84691
 Processing obs        40000  of        84691
 Processing obs        50000  of        84691
 Processing obs        60000  of        84691
 Processing obs        70000  of        84691
 Processing obs        80000  of        84691
 obs_seq_coverage  doneDONEdoneDONE does not exist. Finishing up.

 There were          442  voxels matching the input criterion.
...
```

Discussion

Note that the values of ASSIMILATE_THESE_OBS_TYPES and EVALUATE_THESE_OBS_TYPES are completely irrelevant - since we're not actually doing an assimilation. The **BIG** difference between the two output files is that obsdef_mask.txt contains the metadata for just the matching observations while obsdef_mask.nc contains the metadata for all candidate locations as well as a lot of information about the desired verification times. It is possible to explore obsdef_mask.nc to review the selection criteria to include observations/"voxels" that do not perfectly match the original selection criteria.

Now that you have the obsdef_mask.nc, you can explore it with ncdump.

```
netcdf obsdef_mask {
dimensions:
        voxel = UNLIMITED ; // (512 currently)
        time = 9 ;
        analysisT = 1 ;
        forecast_lead = 9 ;
        nlevels = 14 ;
        linelen = 256 ;
        nlines = 446 ;
        stringlength = 32 ;
        location = 3 ;
variables:
        int voxel(voxel) ;
                voxel:long_name = "desired voxel flag" ;
                voxel:description = "1 == good voxel" ;
        double time(time) ;
                time:long_name = "verification time" ;
                time:units = "days since 1601-1-1" ;
                time:calendar = "GREGORIAN" ;
        double analysisT(analysisT) ;
                analysisT:long_name = "analysis (start) time of each forecast" ;
                analysisT:units = "days since 1601-1-1" ;
```

(continues on next page)

(continued from previous page)

```
                analysisT:calendar = "GREGORIAN" ;
        int forecast_lead(forecast_lead) ;
                forecast_lead:long_name = "current forecast length" ;
                forecast_lead:units = "seconds" ;
        double verification_times(analysisT, forecast_lead) ;
                verification_times:long_name = "verification times during each␣
↪forecast run" ;
                verification_times:units = "days since 1601-1-1" ;
                verification_times:calendar = "GREGORIAN" ;
                verification_times:rows = "each forecast" ;
                verification_times:cols = "each verification time" ;
        float mandatory_level(nlevels) ;
                mandatory_level:long_name = "mandatory pressure levels" ;
                mandatory_level:units = "Pa" ;
        char namelist(nlines, linelen) ;
                namelist:long_name = "input.nml contents" ;
        char obs_type(voxel, stringlength) ;
                obs_type:long_name = "observation type string at this voxel" ;
        double location(voxel, location) ;
                location:description = "location coordinates" ;
                location:location_type = "loc3Dsphere" ;
                location:long_name = "threed sphere locations: lon, lat, vertical" ;
                location:storage_order = "Lon Lat Vertical" ;
                location:units = "degrees degrees which_vert" ;
        int which_vert(voxel) ;
                which_vert:long_name = "vertical coordinate system code" ;
                which_vert:VERTISUNDEF = -2 ;
                which_vert:VERTISSURFACE = -1 ;
                which_vert:VERTISLEVEL = 1 ;
                which_vert:VERTISPRESSURE = 2 ;
                which_vert:VERTISHEIGHT = 3 ;
                which_vert:VERTISSCALEHEIGHT = 4 ;
        int ntimes(voxel) ;
                ntimes:long_name = "number of observation times at this voxel" ;
        double first_time(voxel) ;
                first_time:long_name = "first valid observation time at this voxel" ;
                first_time:units = "days since 1601-1-1" ;
                first_time:calendar = "GREGORIAN" ;
        double last_time(voxel) ;
                last_time:long_name = "last valid observation time at this voxel" ;
                last_time:units = "days since 1601-1-1" ;
                last_time:calendar = "GREGORIAN" ;
        double ReportTime(voxel, time) ;
                ReportTime:long_name = "time of observation" ;
                ReportTime:units = "days since 1601-1-1" ;
                ReportTime:calendar = "GREGORIAN" ;
                ReportTime:missing_value = 0. ;
                ReportTime:_FillValue = 0. ;

// global attributes:
                :creation_date = "YYYY MM DD HH MM SS = 2011 03 01 09 28 40" ;
                :obs_seq_coverage_source = "$URL$" ;
                :obs_seq_coverage_revision = "$Revision$" ;
                :obs_seq_coverage_revdate = "$Date$" ;
                :min_steps_required = 9 ;
                :forecast_length_days = 2 ;
                :forecast_length_seconds = 0 ;
```

(continues on next page)

(continued from previous page)

```
                :verification_interval_seconds = 21600 ;
                :obs_of_interest_001 = "METAR_U_10_METER_WIND" ;
                :obs_of_interest_002 = "METAR_V_10_METER_WIND" ;
                :obs_seq_file_001 = "obs_seq.final.2008060818" ;
data:

 time = 148812.75, 148813, 148813.25, 148813.5, 148813.75, 148814, 148814.25,
    148814.5, 148814.75 ;

 forecast_lead = 0, 21600, 43200, 64800, 86400, 108000, 129600, 151200, 172800 ;
}
```

The first thing to note is that there are more voxels (512) than reported during the run-time output (442). Typically, there will be many more voxels in the netCDF file than will meet the selection criteria - but this is just an example. Some of the voxels in the netCDF file do not meet the selection criteria - meaning they do not have observations at all 9 required times. Furthermore, there are 512 locations for ALL of the desired observation types. In keeping with the DART philosophy of scalar observations, each observation type gets a separate voxel. There are **not** 512 METAR_U_10_METER_WIND observations and 512 METAR_V_10_METER_WIND observations. There are N METAR_U_10_METER_WIND observations and M METAR_V_10_METER_WIND observations where N+M = 512. And only 442 of them have observations at all the times required for the verification. Dump the *obs_type* variable to see what voxel has what observation type.

The *voxel* variable is fundamentally a flag that indicates if the station has all of the desired verification times. Combine that information with the *obs_type* and *location* to determine where your verifications of any particular observation type will take place.

Now that you have the `obsdef_mask.txt`, you can run *program obs_selection* to subset the observation sequence files into one compact file to use in your ensemble forecast.

6.112.6 References

- none - but this seems like a good place to start: The Centre for Australian Weather and Climate Research - Forecast Verification Issues, Methods and FAQ

6.113 PROGRAM `advance_time`

6.113.1 Overview

Provides a shell-scripting-friendly way to increment and decrement calendar dates and times. The code uses the standard DART time manager for all time calculations.

A date, an increment or decrement, and an optional output formatting flag are read from standard input. Increments can be days, hours, minutes, or seconds. The accuracy is to the second. The resulting output time string is echoed to standard output. For example:

```
echo 2007073012 12 | advance_time
```

will output the string 2007073100. It uses the Gregorian calendar and will roll over month and year boundaries, both going forward and backwards in time. See the Usage section below for more examples of use.

The program is general purpose, but based on a time program distributed with the WRF model. This is the reason there are a few WRF specific options, for example the '-w' flag outputs a date string in a WRF-specific format, useful for creating WRF filenames.

The program does require that an 'input.nml' namelist file exist in the current directory, and at least a &utilities_nml namelist (which can be empty) exists.

6.113.2 Usage

Interface identical to `advance_cymdh`, except for reading the arg line from standard input, to be more portable since iargc() is nonstandard across different fortran implementations.

- default numeric increment is hours
- has accuracy down to second
- can use day/hour/minute/second (with/without +/- sign) to advance time
- can digest various input date format if it still has the right order (ie. cc yy mm dd hh nn ss)
- can digest flexible time increment
- can output in wrf date format (ccyy-mm-dd_hh:nn:ss)
- can specify output date format
- can output Julian day
- can output Gregorian days and seconds (since year 1601)

Some examples:

```
advance 12 h:
  echo 20070730      12         | advance_time

back 1 day 2 hours 30 minutes and 30 seconds:
  echo 2007073012   -1d2h30m30s | advance_time

back 3 hours 30 minutes less 1 second:
  echo 2007073012    1s-3h30m   | advance_time

advance 2 days and 1 second, output in wrf date format :
  echo 200707301200  2d1s -w    | advance_time
  echo 2007-07-30_12:00:00 2d1s -w  | advance_time
  echo 200707301200  2d1s -f ccyy-mm-dd_hh:nn:ss | advance_time

advance 120 h, and print year and Julian day:
  echo 2007073006    120 -j     | advance_time

advance 120 h, print year, Julian day, hour, minute and second:
  echo 2007073006    120 -J     | advance_time

print Gregorian day and second (since year 1601):
  echo 2007073006    0 -g       | advance_time
```

6.113.3 Modules used

```
utilities_mod
time_manager_mod
parse_args_mod
```

6.113.4 Namelist

No namelist is currently defined for `advance_time`.

6.113.5 Files

- input.nml

6.114 program `model_mod_check`

6.114.1 Overview

`model_mod_check` tests some of the more fundamental routines in any `model_mod`. This is intended to be used when adding a new model to DART - test the pieces as they are written. As such, this program is meant to be hacked up and customized to your own purpose. Right now, it reads in model netCDF file(s) - one per domain/nest/whatever - and writes out files, queries the metdata, etc. It also exercises `static_init_model()`, which is the first routine to get right . . .

6.114.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_mod_check
   num_ens                = 1
   single_file            = .FALSE.
   input_state_files      = 'null'
   output_state_files     = 'null'
   all_metadata_file      = 'metadata.txt'

   test1thru              = 7
   run_tests              = -1

   x_ind                  = -1
   loc_of_interest        = -1.0, -1.0, -1.0
   quantity_of_interest   = 'NONE'

   interp_test_dlon       = 10.0
   interp_test_dlat       = 10.0
   interp_test_dvert      = 10.0

   interp_test_lonrange   = 0.0, 120.0
   interp_test_latrange   = 0.0, 120.0
```

(continues on next page)

(continued from previous page)

```
interp_test_vertrange = 0.0, 100.0

interp_test_dx        = -888888.0
interp_test_dy        = -888888.0
interp_test_dz        = -888888.0

interp_test_xrange    = -888888.0, -888888.0
interp_test_yrange    = -888888.0, -888888.0
interp_test_zrange    = -888888.0, -888888.0

interp_test_vertcoord = 'VERTISHEIGHT'
verbose               = .FALSE.
/
```

Item	Type	Description
num_ens	integer	Provided for future use. Must be 1. Ultimately, The number of ensemble members you would like to read in for testing.
single_file	logical	If .TRUE. all members are stored in a single restart file.
input_state_files(:)	character(len=256)	The name(s) of the NetCDF file(s) containing the model states, one per domain. If num_ens > 1 and not single_file, specify a filename for each ensemble member (num_ens). If you have both multiple ensemble members in separate files AND multiple domains, specify all the ensemble member filenames for domain 1, then all the ensemble member filenames for domain 2, etc.
output_state_files(:)	character(len=256)	The name(s) of the output NetCDF file(s) for testing IO, one per domain. If num_ens > 1 and not single_file, specify a filename for each ensemble member (num_ens). If you have both multiple ensemble members in separate files AND multiple domains, specify all the ensemble member filenames for domain 1, then all the ensemble member filenames for domain 2, etc.
all_metadata_file	character(len=256)	Test 6 produces an exhaustive list of metadata for EVERY element in the DART state vector. The metadata get written to this file name.
x_ind	integer(i8)	An integer index into the DART state vector. This will be used to test the metadata routines. Answers questions about location, what variable type is stored there, etc.
loc_of_interest	real(r8), dimension(3)	The lat/lon/level for a **particular** location. Used in Test 4, the single-point interpolation test. Indirectly tests the routine to find the closest gridpoint.
quantity_of_interest	character(len=32)	Specifies the QUANTITY of the model state to use in Tests 4, 5, and 7.
interp_test_dlon	real(r8)	The distance (measured in degrees) on the longitude interpolation grid. Ignored if interpolating with cartesian coordinates. Used in Test 5.
interp_test_dlat	real(r8)	The distance (measured in degrees) on the latitude interpolation grid. Ignored if interpolating with cartesian coordinates. Used in Test 5.
interp_test_dvert	real(r8)	The distance (measured in interp_vertcoord) on the vertical interpolation grid. Ignored if interpolating with cartesian coordinates. Used in Test 5.

A more typical namelist for a single ensemble member for a model with an outer grid and a single nested grid is shown below.

```
&model_mod_check_nml
   input_state_files     = 'dart_vector1.nc','dart_vector2.nc'
   output_state_files    = 'check_me1.nc', 'check_me2.nc'
   all_metadata_file     = 'metadata.txt'
   verbose               = .TRUE.
   test1thru             = 5
   run_tests             = -1
   loc_of_interest       = 243.72386169, 52.78578186, 10.0
   x_ind                 = 12666739
   quantity_of_interest  = 'QTY_POTENTIAL_TEMPERATURE'
   interp_test_lonrange  = 144.0, 326.0
   interp_test_dlon      = 1.0
   interp_test_latrange  = -5.0, 80.0
   interp_test_dlat      = 1.0
   interp_test_vertrange = 100.0, 11000.0
   interp_test_dvert     = 200.0
   interp_test_vertcoord = 'VERTISHEIGHT'
  /
```

6.114.3 Other modules used

```
assimilation_code/location/threed_sphere/location_mod.f90
assimilation_code/location/utilities/default_location_mod.f90
assimilation_code/location/utilities/location_io_mod.f90
assimilation_code/modules/assimilation/adaptive_inflate_mod.f90
assimilation_code/modules/assimilation/assim_model_mod.f90
assimilation_code/modules/assimilation/assim_tools_mod.f90
assimilation_code/modules/assimilation/cov_cutoff_mod.f90
assimilation_code/modules/assimilation/filter_mod.f90
assimilation_code/modules/assimilation/obs_model_mod.f90
assimilation_code/modules/assimilation/quality_control_mod.f90
assimilation_code/modules/assimilation/reg_factor_mod.f90
assimilation_code/modules/assimilation/sampling_error_correction_mod.f90
assimilation_code/modules/assimilation/smoother_mod.f90
assimilation_code/modules/io/dart_time_io_mod.f90
assimilation_code/modules/io/direct_netcdf_mod.f90
assimilation_code/modules/io/io_filenames_mod.f90
assimilation_code/modules/io/state_structure_mod.f90
assimilation_code/modules/io/state_vector_io_mod.f90
assimilation_code/modules/observations/forward_operator_mod.f90
assimilation_code/modules/observations/obs_kind_mod.f90
assimilation_code/modules/observations/obs_sequence_mod.f90
assimilation_code/modules/utilities/distributed_state_mod.f90
assimilation_code/modules/utilities/ensemble_manager_mod.f90
assimilation_code/modules/utilities/netcdf_utilities_mod.f90
assimilation_code/modules/utilities/null_mpi_utilities_mod.f90
assimilation_code/modules/utilities/null_win_mod.f90
assimilation_code/modules/utilities/obs_impact_mod.f90
assimilation_code/modules/utilities/options_mod.f90
assimilation_code/modules/utilities/parse_args_mod.f90
assimilation_code/modules/utilities/random_seq_mod.f90
assimilation_code/modules/utilities/sort_mod.f90
assimilation_code/modules/utilities/time_manager_mod.f90
```

(continues on next page)

(continued from previous page)

```
assimilation_code/modules/utilities/types_mod.f90
assimilation_code/modules/utilities/utilities_mod.f90
assimilation_code/programs/model_mod_check/model_mod_check.f90
models/your_model_here/model_mod.f90
models/model_mod_tools/test_interpolate_threed_sphere.f90
models/model_mod_tools/model_check_utilities_mod.f90
models/utilities/default_model_mod.f90
observations/forward_operators/obs_def_mod.f90
observations/forward_operators/obs_def_utilities_mod.f90
```

Items highlighted may change based on which model is being tested.

6.114.4 Files

- `input.nml` is used for `model_mod_check_nml`
- The `"input_state_files"` can either be a single file containing multiple restart files, or a single NetCDF restart file. One file per domain.
- The `"output_state_files"` is the output netCDF files from Test 2. Check the attributes, values, etc.
- `check_me_interptest.nc` and `check_me_interptest.m` are the result of Test 5.
- `"all_metadata_file"` is the run-time output of Test 6.

6.114.5 Usage

Normal circumstances indicate that you are trying to put a new model into DART, so to be able to build and run `model_mod_check`, you will need to create a `path_names_model_mod_check` file with the following contents:

```
assimilation_code/location/threed_sphere/location_mod.f90
assimilation_code/location/utilities/default_location_mod.f90
assimilation_code/location/utilities/location_io_mod.f90
assimilation_code/modules/assimilation/adaptive_inflate_mod.f90
assimilation_code/modules/assimilation/assim_model_mod.f90
assimilation_code/modules/assimilation/assim_tools_mod.f90
assimilation_code/modules/assimilation/cov_cutoff_mod.f90
assimilation_code/modules/assimilation/filter_mod.f90
assimilation_code/modules/assimilation/obs_model_mod.f90
assimilation_code/modules/assimilation/quality_control_mod.f90
assimilation_code/modules/assimilation/reg_factor_mod.f90
assimilation_code/modules/assimilation/sampling_error_correction_mod.f90
assimilation_code/modules/assimilation/smoother_mod.f90
assimilation_code/modules/io/dart_time_io_mod.f90
assimilation_code/modules/io/direct_netcdf_mod.f90
assimilation_code/modules/io/io_filenames_mod.f90
assimilation_code/modules/io/state_structure_mod.f90
assimilation_code/modules/io/state_vector_io_mod.f90
assimilation_code/modules/observations/forward_operator_mod.f90
assimilation_code/modules/observations/obs_kind_mod.f90
assimilation_code/modules/observations/obs_sequence_mod.f90
assimilation_code/modules/utilities/distributed_state_mod.f90
assimilation_code/modules/utilities/ensemble_manager_mod.f90
assimilation_code/modules/utilities/netcdf_utilities_mod.f90
```

(continues on next page)

(continued from previous page)

```
assimilation_code/modules/utilities/null_mpi_utilities_mod.f90
assimilation_code/modules/utilities/null_win_mod.f90
assimilation_code/modules/utilities/obs_impact_mod.f90
assimilation_code/modules/utilities/options_mod.f90
assimilation_code/modules/utilities/parse_args_mod.f90
assimilation_code/modules/utilities/random_seq_mod.f90
assimilation_code/modules/utilities/sort_mod.f90
assimilation_code/modules/utilities/time_manager_mod.f90
assimilation_code/modules/utilities/types_mod.f90
assimilation_code/modules/utilities/utilities_mod.f90
assimilation_code/programs/model_mod_check/model_mod_check.f90
models/your_model_here/model_mod.f90
models/model_mod_tools/test_interpolate_threed_sphere.f90
models/utilities/default_model_mod.f90
observations/forward_operators/obs_def_mod.f90
observations/forward_operators/obs_def_utilities_mod.f90
```

as well as a `mkmf_model_mod_check` script. You should be able to look at any other `mkmf_xxxx` script and figure out what to change. Once they exist:

```
[~/DART/models/yourmodel/work] % csh mkmf_model_mod_check
[~/DART/models/yourmodel/work] % make
[~/DART/models/yourmodel/work] % ./model_mod_check
```

Unlike other DART components, you are expected to modify `model_mod_check.f90` to suit your needs as you develop your `model_mod`. The code is roughly divided into the following categories:

1. Check the geometry information,
2. Read/write a restart file,
3. Check the construction of the state vector … i.e. the metadata,
4. Interpolate at a single point,
5. Interpolate for a range of points.

Test 0. mandatory

The first test in `model_mod_check` reads the namelist and runs `static_init_model` - which generally sets the geometry of the grid, the number of state variables and their shape, etc. Virtually everything requires knowledge of the grid and state vector, so this block cannot be skipped.

Test 1. checking the geometry information

The first test in `model_mod_check` exercises a basic required interface `get_model_size()`. This also generates a report on the geometry of the grid, the number of state variables and their shape, etc. as well as the total number of elements in the DART state vector.

Test 2. read/writing a restart file

This directly reads and write state variables from the model netCDF file. This is a nice sanity check to make sure that the DART state vector is being read in properly.

Test 3. check the construction of the state vector

It is critical to return the correct metadata for any given index into the DART state vector. This code block tests the two most common features of the metadata. As a bonus, this routine is also quite useful to determine EXACTLY where to place your first test observation. If you test precisely at a grid location, you should be able to really get a handle on debugging your `model_interpolate()` routine.

Test 4. test interpolation on a single point

This tests your model's interpolation routine on a single point and returns the interpolated value. This requires that Test 2 works - it needs a valid model state with data. Test 2 is automatically run if this test is selected.

Test 5. test interpolation on a range of values

This tests your model's interpolation routine on a range of values returns the interpolated grid in `check_me_interptest.nc` and `check_me_interptest.m` which can be read in Matlab and used to visualize the result. This requires that Test 2 works - it needs a valid model state with data. Test 2 is automatically run if this test is selected.

Test 6. exhaustively test the construction of the state vector

This can be a long test, depending on the size of your state vector. This returns the same data as in Test 3 - but *for every element* in the state vector. The metadata are written to a file specified by `all_metadata_file` and `check_me_interptest.m` which can be read in Matlab and used to visualize the result.

Test 7. find the closest gridpoint to a test location

This is a good test to verify that *get_state_meta_data()* and the grid information are correct. Typically, one would put in a location that is actually **on** the grid and see if the correct gridpoint index is returned. Repeat the test with slightly different locations until the next gridpoint is closer. Repeat . . .

6.114.6 References

- none

6.115 PROGRAM `closest_member_tool`

6.115.1 Overview

Utility program to compare the ensemble mean to an ensemble of restart files, which can now be run in parallel. The program prints out a sorted order of which members are 'closest' to the mean, where the method used to determine 'close' is selectable by namelist option. It also creates a file with a single number or character string in it, for ease in scripting, which identifies the closest member.

The ensemble mean is computed from the input ensemble. The difference is computed point by point across the ensemble members. There is an option to restrict the computation to just a subset of the entire state vector by listing one or more generic quantities. In this case, only state vector items matching one of these quantities will contribute to the total difference value.

Available methods are:

1 - simple absolute difference: The absolute value of the difference between each item in the mean vector and the corresponding item in each ensemble member, accumulated over the entire state vector.

2 - normalized absolute difference: The absolute value of the difference between each item in the mean vector and the corresponding item in each ensemble member normalized by the mean value, accumulated over the entire state vector.

3 - simple RMS difference: The square root of the accumulated sum of the square of the difference between each item in the mean vector and the corresponding item in each ensemble member.

4 - normalized RMS difference: The square root of the accumulated sum of the square of the normalized difference between each item in the mean vector and the corresponding item in each ensemble member.

This program could be used to select one or more ensemble members to run a free model forecast forward in time after the assimilation is finished. Each member is an equally likely representation of the model state. Using the ensemble mean may not be the best choice since the mean may not have self-consistent fine-scale structures in the data.

In addition to printing out data about all members to both the console and to the dart log file, this program creates a single output file containing information about the closest member. If the input restart data is in a single file, the output file 'closest_restart' contains a single number which is the ensemble member number. If the input restart data is in separate files, the output file contains the full filename of the closest member, e.g. 'filter_restart.0004' if member 4 is closest. For scripting the contents of this file can be used to copy the corresponding member data and convert it to the model input format for a free forecast, for example.

6.115.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&closest_member_tool_nml
   input_restart_files     = ''
   input_restart_file_list = ''
   output_file_name        = 'closest_restart'
   ens_size                = 1
   difference_method       = 4
   use_only_qtys           = ''
   single_restart_file_in  = .false.
  /
```

Item	Type	Description
input_restart_files	character(len=256),dimension(ens_size x num_domains)	An array of filenames each containing a list DART restart data.
input_restart_file_list	character(len=256),dimension(num_domains)	A file containing a list of filenames for DART restart data, one for each domain.
output_file_name	character(len=256)	This is a file containing the member number that is closest to the ensemble mean.
ens_size	integer	Total number of ensemble members.
difference_method	integer	Select which method is used to compute 'distance' from mean: • 1 = simple absolute difference • 2 = absolute difference normalized by the mean • 3 = simple RMS difference • 4 = RMS of the normalized difference
use_only_quantities	character(len=32)	If unspecified, all items in the state vector contribute to the total difference. If one or more quantities are listed here, only items in the state vector of these quantities contribute to the total difference. These are the generic quantities, such as QTY_TEMPERATURE, QTY_U_WIND_COMPONENT, QTY_DENSITY, etc. and not specific types like RADIOSONDE_TEMPERATURE. Consult the model interface code to determine which possible quantities are returned by the get_state_meta_data() routine.
single_restart_file_in	logical	**Not supported yet.** Contact dart@ucar.edu if you are interested in using this tool with files that contain all ensemble members in a single file.

Below is an example of a typical namelist for the closest_member_tool.

```
&closest_member_tool_nml
   input_restart_files     = ''
   input_restart_file_list = 'restart_list.txt'
   output_file_name        = 'closest_restart.txt'
   ens_size                = 3
   single_restart_file_in  = .false.
   difference_method       = 4
   use_only_qtys           = ''
  /
```

where restart_list.txt contains

```
cam_restart_0001.nc
cam_restart_0002.nc
cam_restart_0003.nc
```

Currently single_restart_file_in is not supported. This is typically used for simpler models that have built in model advances such as lorenz_96.

6.115.3 Files

- inputfile.####.nc (list of restarts to find closest member) -or-
- restart_list.txt (a file containing a list of restart files) and,
- input.nml

6.115.4 References

- none

6.116 PROGRAM restart_file_tool

6.116.1 Overview

This tool still exists in the "Classic" release of DART but is no longer needed in the "Manhattan" release. DART initial condition and restart files are now in NetCDF format and any standard NetCDF tool can be used to manipulate them.

6.117 PROGRAM filter

6.117.1 Overview

Main program for driving ensemble filter assimilations.

filter is a Fortran 90 program, and provides a large number of options for controlling execution behavior and parameter configuration that are driven from its namelist. See the namelist section below for more details. The number of assimilation steps to be done is controlled by the input observation sequence and by the time-stepping capabilities of the model being used in the assimilation.

This overview includes these subsections:

- Program Flow
- Filter Types
- Getting Started
- Free Model Run after Assimilation
- Evaluate a Model State against Observations

- Compare Results with and without Assimilation
- DART Quality Control Values on Output
- Description of Inflation Options
- Detailed Program Flow

See the DART web site for more documentation, including a discussion of the capabilities of the assimilation system, a diagram of the entire execution cycle, the options and features.

Program flow

The basic execution loop is:

- Read in model initial conditions, observations, set up and initialize
- Until out of observations:
  - Run multiple copies of the model to get forecasts of model state
  - Assimilate all observations in the current time window
  - Repeat
- Write out diagnostic files, restart files, final observation sequence file

The time of the observations in the input observation sequence file controls the length of execution of filter.

For large, parallel models, the execution loop is usually wrapped in an external script which does these additional steps:

- Link to an observation sequence file which contains only observation times within the next assimilation window
- Link any output inflation files from the previous step to be the input files for this step
- Run filter, which will exit after doing the assimilation without trying to advance the model
- Save the output diagnostic files for later
- Advance the N copies of the model using the model scripts or whatever method is appropriate
- Repeat until all data is assimilated

For large models filter is almost always compiled to be a parallel MPI program, and most large models are themselves a parallel program using OpenMP, MPI, or both. MPI programs usually cannot start other MPI programs, so the external script submits both the filter job and the N model advances to a batch system so all run as independent parallel jobs.

The same source code is used for all applications of filter. The code specific to the types of observations and the interface code for the computational model is configured at compile time. The top level directory has been simplified from previous versions to look like :

- `README`
- `COPYRIGHT`
- *assimilation_code*
- *build_templates*
- *diagnostics*
- *documentation*
- *models*
- *observations*

the *assimilation_code* contains all *module* and *program* source code for all of the main programs including filter. Specifically in the modules directory there is a `filter_mod.f90` which contains the source for the filter main program. Each model has a separate directory under DART/models, and under each model is a work directory where the code is compiled and can be run for testing. Generally when a full-size experiment is done the executables are copied to a different location - e.g. scratch space on a large filesystem - since the data files for 10s to 100s of copies of a model can get very large. A lightly pruned directory tree can be browsed in the main index.html.

Directories expected to be modified

DART is distributed as a toolkit/library/facility that can be used as-is with the existing models and observations, but is also designed so that users can add new models, new observation types and forward operators, and new assimilation algorithms.

The locations in the DART code tree which are intended to be modified by users are:

New Models Add a new directory in the `models` subdirectory. Copy (recursively, e.g. `cp -r`) the contents of the `template` directory and modify from there. Note that the `model_mod.f90` file in the template dir is appropriate for small models; for large geophysical models see the `full_model_mod.f90` file and also examine other model directories for ideas. See additional documentation in the *MODULE model_mod* documentation, and the DART web pages on adding new models.

New Observation Platforms To convert observations from other formats to DART format, add a new directory in the `observations/obs_converters` subdirectory and populate it with converter code.

New Observation Types and Forward Operators Define a new type (a measurement from an observing platform) via a file in the `observations/forward_operators` subdirectory. If the forward operator is more complicated than directly interpolating a field in the model state, this is where the code for that goes. See additional documentation in the *MODULE obs_def_mod* documentation, and the DART web pages on adding new types. Adding a new type may require adding a new `generic kind`, which is documented in *MODULE obs_kind_mod*.

New Assimilation Algorithms If you want to try out a different filter type modify the filter code in the `assim_tools_mod.f90` file. See the *MODULE assim_tools_mod* documentation.

Detailed program execution flow

The Manhattan release of DART includes state space output expanded from the previous two stages (Prior and Posterior) to up to four (input, preassim, postassim, and output). This makes it possible to examine the states with and without either kind of inflation, as described below. In addition, the state space vectors are each written to a separate NetCDF file: `${stage}_mean.nc, ${stage}_sd.nc, ${stage}_member_####.nc` . The detailed execution flow inside the filter program is:

- Read in observations.
- Read in state vectors from model netcdf restart files.
- Initialize inflation fields, possibly reading netcdf restart files.
- If requested, initialize and write to “input” netcdf diagnostic files.
- Trim off any observations if start/stop times specified.
- Begin main assimilation loop:
    - Check model time vs observation times:
        - If current assimilation window is earlier than model time, error.
        - If current assimilation window includes model time, begin assimilating.

    - If current assimilation window is later than model time, advance model:
      - Write out current state vectors for all ensemble members.
      - Advance the model by subroutine call or by shell script:
      - Tell the model to run up to the requested time.
      - Read in new state vectors from netcdf files for all ensemble members.
  - Apply prior inflation if requested.
  - Compute ensemble of prior observation values with forward operators.
  - If requested, compute and write the "preassim" netcdf diagnostic files. This is AFTER any prior inflation has been applied.
  - Compute prior observation space diagnostics.
  - Assimilate all observations in this window:
    - Get all obs locations and kinds.
    - Get all state vector locations and kinds.
    - For each observation:
      - Compute the observation increments.
      - Find all other obs and states within localization radius.
      - Compute the covariance between obs and state variables.
      - Apply increments to state variables weighted by correlation values.
      - Apply increments to any remaining unassimilated observations.
      - Loop until all observations in window processed.
  - If requested, compute and write the "postassim" netcdf diagnostic files (members, mean, spread). This is BEFORE any posterior inflation has been applied.
  - Apply posterior inflation if requested.
  - Compute ensemble of posterior observation values with forward operators.
  - Compute posterior observation space diagnostics.
  - If requested, compute and write out the "output" netcdf diagnostic files (members, mean, spread). This is AFTER any posterior inflation has been applied.
  - Loop until all observations in input file processed.
- Close diagnostic files.
- Write out final observation sequence file.
- Write out inflation restart files if requested.
- Write out final state vectors to model restart files if requested.
- Release memory for state vector and observation ensemble members.

6.117.2 Namelist

See the filter namelist page for a detailed description of all `&filter_nml` variables. This namelist is read from the file `input.nml`.

6.117.3 Modules used

```
mpi_utilities_mod
filter_mod
```

Note that filter_mod.f90 uses many more modules.

6.117.4 Files

See Detailed Program Flow for a short description of DART's new 'stages'. In addition, the Manhattan release simplifies some namelists by replacing many user-settable file names with hardwired filenames. Files can then be renamed in the run scripts to suit the user's needs.

- input ensemble member states; from *&filter_nml :: input_state_files* or *input_state_file_list*
- output ensemble member states; to *&filter_nml :: output_state_files* or *output_state_file_list*
- input observation sequence file; from `&filter_nml :: obs_sequence_in_name`
- output observation sequence file; from `&filter_nml :: obs_sequence_out_name`
- output state space diagnostics files; `${stage}_mean.nc, ${stage}_sd.nc,` where stage = {input,preassim,postassim,output}
- input state space inflation data (if enabled); from `input_{prior,post}inf_{mean,sd}.nc.`
- output state space inflation data (if enabled); to `${stage}_{prior,post}inf_{mean,sd}.nc.`, where stage "input"
- input.nml, to read &filter_nml

6.117.5 References

- Anderson, J. L., 2001: An Ensemble Adjustment Kalman Filter for Data Assimilation. Mon. Wea. Rev., 129, 2884-2903. doi: 10.1175/1520-0493(2001)129<2884:AEAKFF>2.0.CO;2
- Anderson, J. L., 2003: A Local Least Squares Framework for Ensemble Filtering. Mon. Wea. Rev., 131, 634-642. doi: 10.1175/1520-0493(2003)131<0634:ALLSFF>2.0.CO;2
- Anderson, J. L., 2007: An adaptive covariance inflation error correction algorithm for ensemble filters. Tellus A, 59, 210-224. doi: 10.1111/j.1600-0870.2006.00216.x
- Anderson, J. L., 2007: Exploring the need for localization in ensemble data assimilation using a hierarchical ensemble filter. Physica D, 230, 99-111. doi:10.1016/j.physd.2006.02.011
- Anderson, J., Collins, N., 2007: Scalable Implementations of Ensemble Filter Algorithms for Data Assimilation. Journal of Atmospheric and Oceanic Technology, 24, 1452-1463. doi: 10.1175/JTECH2049.1
- Anderson, J. L., 2009: Spatially and temporally varying adaptive covariance inflation for ensemble filters. Tellus A, 61, 72-83. doi: 10.1111/j.1600-0870.2008.00361.x

- Anderson, J., T. Hoar, K. Raeder, H. Liu, N. Collins, R. Torn, and A. Arellano, 2009: The Data Assimilation Research Testbed: A Community Facility. Bull. Amer. Meteor. Soc., 90, 1283-1296. doi: 10.1175/2009BAMS2618.1
- Anderson, J. L., 2010: A Non-Gaussian Ensemble Filter Update for Data Assimilation. Mon. Wea. Rev., 139, 4186-4198. doi: 10.1175/2010MWR3253.1
- Anderson, J. L., 2011: Localization and Sampling Error Correction in Ensemble Kalman Filter Data Assimilation. Submitted for publication, Jan 2011. Contact author.

6.118 program `obs_keep_a_few`

6.118.1 Overview

This program creates an output observation sequence (obs_seq) file that is shorter than the input obs_seq file. There are two ways to restrict the number of observations copied to the output: the total number of observations regardless of observation type, or up to N observations of each type. Observations in an obs_seq file are processed in time order so the observations with the earliest timestamps will be copied.

Set either limit to -1 to disable it. If both the maximum count per type and maximum total count are given the copying stops when the first limit is reached.

If you want to subset an obs_seq file starting at a later time see the *program obs_sequence_tool* for subsetting by time and then use this tool on the output. That tool also allows you to subset by obs type, location, data value, and a variety of other options.

The `obs_keep_a_few` program only subsets by numbers of observations. It is expected to be useful when prototyping experiments so the run time is short, or for debugging or testing. Setting a limit per type ensures you have up to N of each type of observation present in the output file.

Identity observations are all considered to be the same identity "observation type" by this tool.

6.118.2 Other modules used

```
types_mod
utilities_mod
location_mod
obs_def_mod
obs_kind_mod
time_manager_mod
obs_sequence_mod
```

6.118.3 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_keep_a_few_nml
   filename_in         = ''
   filename_out        = ''
   max_count_per_type = 10
   max_total_count     = -1
```

(continues on next page)

(continued from previous page)

```
   print_only          = .false.
   calendar            = 'Gregorian'
   /
```

Item	Type	Description
filename_in	character(len=256)	Name of the observation sequence file to read.
filename_out	character(len=256)	Name of the observation sequence file to create. An existing file will be overwritten.
max_count_per_type	integer	The first N observations of each different type will be copied to the output file. Observation sequence files are processed in time order so these will be the ones with the earliest time stamps relative to other observations of this same type. Set to -1 to disable this limit.
max_total_count	integer	If greater than 0, sets the upper limit on the total number of observations to be copied to the output file regardless of type. The program quits when either this limit is reached or when there are N of each different obs type in the output. Set to -1 to disable.
print_only	logical	If true, does all the work and prints out what the output file would have in it (timestamps and counts of each obs type) but doesn't create the output file.
calendar	character(len=256)	Name of the DART calendar type to use. Generally 'Gregorian' or 'No calendar'. See the DART time manager for more options. Only controls the formatting of how the times in the output summary messages are displayed.

6.118.4 Files

- filename_in is read.
- filename_out is written.

6.118.5 References

- none

6.119 program `create_obs_sequence`

6.119.1 Overview

This program creates an observation sequence file using values read from standard input. It is typically used to create synthetic observations, or shorter sequences of observations (although there is no limit on the number of observations). For creating observation sequence files directly from large, real-world observation datasets, see the observations directory.

This program can be run interactively (input from a terminal), or input files can be created with a text editor, perl or matlab script, or any other convenient method, and then run with standard input redirected from this file. The latter method is most commonly used to create larger observation sequence files for perfect model applications.

The program can create complete observation sequences ready to be assimilated, or it can create observations with only partial data which is later filled in by another program. Each observation needs to have a type, location, time, expected error, and optionally a data value and/or a quality control indicator. For perfect model applications, it is usually convenient to define 0 quality control fields and 0 copies of the data for each observation. The output of create_obs_sequence can be read by *program perfect_model_obs* which will then create a synthetic (perfect_model) observation sequence complete with two copies of the data for each observation: the observed value and the 'true' value.

Another common approach for perfect model applications is to use create_obs_sequence to define a set of observation locations and types, and where observations will be repeatedly sampled in time. When running create_obs_sequence, specify a single observation for each different location and type, with 0 copies of data and giving all the observations the same time. Then the program *program create_fixed_network_seq* can read the output of create_obs_sequence and create an observation sequence file that will contain the set of input observations at a number of different times. This models a fixed observation station, observing the system at some frequency in time.

This program can also create what are called "identity observations". These are observations located directly at one of the state variables, so that computing the value requires no model interpolation but simply returns the actual state variable value. To specify these types of observations, the convention is to put in the negative index number for the offset of that state variable in the state vector. By specifying the index both the observation kind and location are defined by the kind and location of that state variable.

The types of observations which can be created by this program is controlled by the observation types built into the source files created by the *PROGRAM preprocess* program. The preprocess namelist sets the available observation types, and must be run each time it is changed, and then the create_obs_sequence program must be recompiled to incorporate the updated source files.

6.119.2 Other modules used

```
utilities_mod
obs_sequence_mod
assim_model_mod
```

6.119.3 Namelist

This program does not use a namelist. All user input is prompted for at the command line.

6.119.4 Files

- A file containing the output sequence is created. (`set_def.out` is the recommended name)

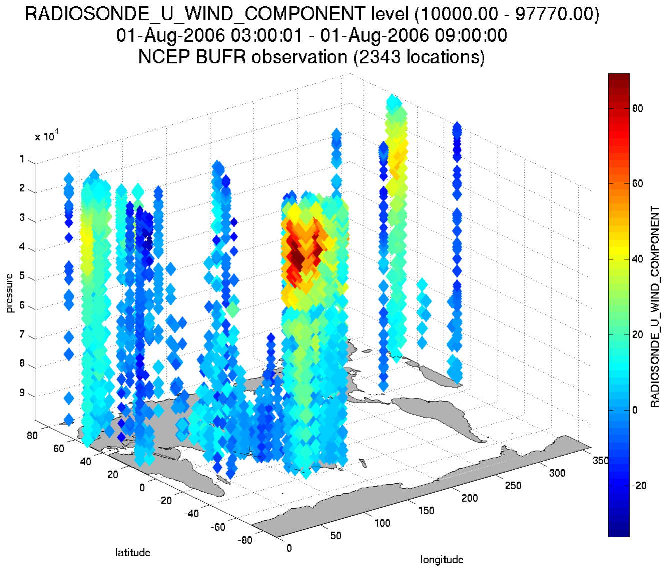
6.119.5 References

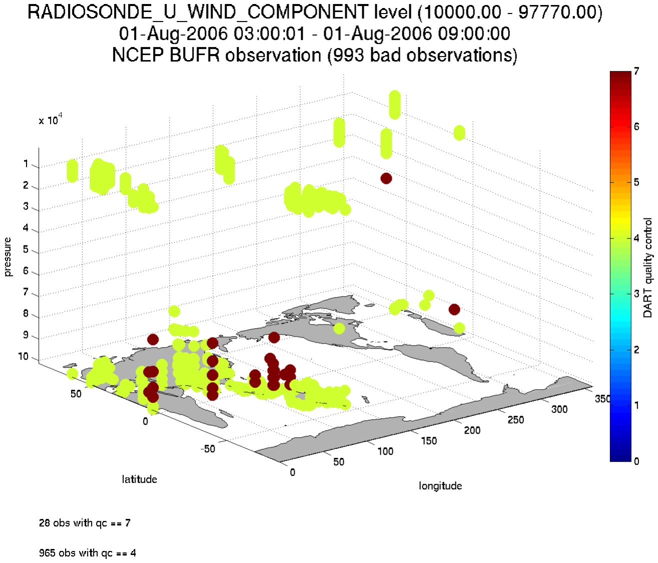
- none

6.120 PROGRAM `obs_seq_to_netcdf`

6.120.1 Overview

`obs_seq_to_netcdf` is a routine to extract the observation components from observation sequence files and write out netCDF files that can be easily digested by other applications. This routine will allow you to plot the spatial distribution of the observations and be able to discern which observations were assimilated or rejected, for example. Here are some graphics from `DART/diagnostics/matlab/plot_obs_netcdf.m`.





The intent is that user input is queried and a series of output files - one per assimilation cycle - will contain the observations for that cycle. It is hoped this will be useful for experiment design or, perhaps, debugging. This routine is also the first to use the new `schedule_mod` module which will ultimately control the temporal aspects of the assimilations (i.e. the assimilation schedule).

There is also a facility for exploring the spatial distributions of quantities like bias between the ensemble mean and the observations: `DART/diagnostics/matlab/plot_obs_netcdf_diffs.m`.

Required namelist interfaces `&obs_seq_to_netcdf` and `&schedule_nml` are read from file `input.nml`.

What's on the horizon ..

obs_seq_to_netcdf is a step toward encoding our observations in netCDF files.

The dependence on the ``threed_sphere/location_mod.f90`` has been removed. This program will work with any ``location_mod.f90``. Also, this program no longer tries to construct 'wind' observations from horizontal components since the program really should be faithful to preserving exactly what is in the input file. i.e. We're not making stuff up.

There are several Matlab scripts that understand how to read and plot observation data in netcdf format. See the link_obs.m script that creates several linked figures with the ability to 'brush' data in one view and have those selected data (and attributes) get highlighted in the other views.

6.120.2 Namelist

This namelist is read from the file input.nml. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_seq_to_netcdf_nml
   obs_sequence_name = 'obs_seq.final',
   obs_sequence_list = '',
   append_to_netcdf  = .false.,
   lonlim1     =    0.0,
   lonlim2     =  360.0,
   latlim1     =  -90.0,
   latlim2     =   90.0,
   verbose     = .false.
/
```

The allowable ranges for the region boundaries are: latitude [-90.,90], longitude [0.,360.] ... but it is possible to specify a region that spans the dateline by specifying the lonlim2 to be less than lonlim1.

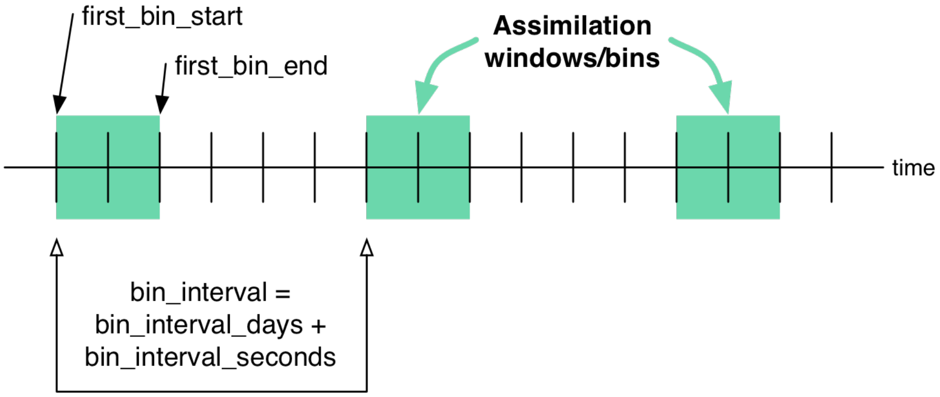
You can only specify **either** obs_sequence_name **or** obs_sequence_list – not both.

Item	Type	Description
obs_sequence_name	character(len=256)	Name of an observation sequence file(s). This may be a relative or absolute filename. If the filename contains a '/', the filename is considered to be comprised of everything to the right, and a directory structure to the left. The directory structure is then queried to see if it can be incremented to handle a sequence of observation files. The default behavior of `obs_seq_to_netcdf` is to look for additional files to include until the files are exhausted or an `obs_seq.final` file is found that contains observations beyond the timeframe of interest. e.g. 'obsdir_001/obs_seq.final' will cause `obs_seq_to_netcdf` to look for 'obsdir_002/obs_seq.final', and so on. If this is specified, 'obs_sequence_list' must be set to ' '.
obs_sequence_list	character(len=256)	Name of an ascii text file which contains a list of one or more observation sequence files, one per line. If this is specified, 'obs_sequence_name' must be set to ' '. Can be created by any method, including sending the output of the 'ls' command to a file, a text editor, or another program.
append_to_netcdf	logical	This gives control over whether to overwrite or append to an existing netcdf output file. It is envisioned that you may want to combine multiple observation sequence files into one netcdf file (i.e. `append_to_netcdf=.true.`) to explore the effects on data coverage, etc. The default behavior is to create a new `obs_epoch_xxx.nc` file with every execution.
lonlim1	real	Westernmost longitude of the region in degrees.
lonlim2	real	Easternmost longitude of the region in degrees. *If this value is* **less than** *the westernmost value, it defines a region that spans the prime meridian.* It is perfectly acceptable to specify lonlim1 = 330 , lonlim2 = 50 to identify a region like "Africa".
latlim1	real	Southernmost latitude of the region in degrees.
latlim2	real	Northernmost latitude of the region in degrees.
verbose	logical	Print extra info about the obs_seq_to_netcdf run.

The schedule namelist

The default values specify one giant 'bin'.

If the `print_table` variable is 'true' a summary of the assimilation schedule will be written to the screen.



```
&schedule_nml
   calendar        = 'Gregorian',
   first_bin_start =  1601,  1,  1,  0,  0,  0,
   first_bin_end   =  2999,  1,  1,  0,  0,  0,
   last_bin_end    =  2999,  1,  1,  0,  0,  0,
   bin_interval_days    = 1000000,
   bin_interval_seconds = 0,
```

(continues on next page)

(continued from previous page)

```
   max_num_bins          = 1000,
   print_table           = .true.
  /
```

Item	Type	Description
calendar	character(len=32)	Type of calendar to use to interpret dates. May be any type supported by the `time_manager_mod`. The string is case-insensitive.
first_bin_start	integer, dimension(6)	the first time of the first assimilation period. The six integers are: year, month, day, hour, hour, minute, second – in that order.
first_bin_end	integer, dimension(6)	the end of the first assimilation period. The six integers are: year, month, day, hour, hour, minute, second – in that order.
last_bin_end	integer, dimension(6)	the approximate end of the last assimilation period. The six integers are: year, month, day, hour, hour, minute, second – in that order. This does not need to be exact, the values from `last_bin_end`, `bin_interval_days`, and `bin_interval_seconds` are used to derive the assimilation schedule. The assimilation periods are repeated and will stop on or before the time defined by `last_bin_end`. See also `max_num_bins`.
bin_interval_days, bin_interval_seconds	integer	Collectively, `bin_interval_days` and `bin_interval_seconds` define the time between the start of successive assimilation windows. It is not possible to define a bin_interval such that there are overlapping bins (i.e. you can't use the same observations more than once).
max_num_bins	integer	An alternate way to specify the maximum number of assimilation periods. The assimilation bin is repeated by the bin_interval until one of two things happens: either the last time of interest is encountered (defined by `last_bin_end`) or the maximum number of assimilation periods has been reached (defined by `max_num_bins`).
print_table	logical	Prints the assimilation schedule.

Example

The following example illustrates the fact the `last_bin_end` does not have to be a 'perfect' bin end - and it gives you an idea of an assimilation schedule table. Note that the user input defines the last bin to end at 09 Z, but the last bin in the table ends at 06 Z.

```
&schedule_nml
   calendar        = 'Gregorian',
   first_bin_start =  2006, 8, 1, 0, 0, 0 ,
   first_bin_end   =  2006, 8, 1, 6, 0, 0 ,
```

(continues on next page)

(continued from previous page)

```
   last_bin_end    =  2006, 8, 2, 9, 0, 0 ,
   bin_interval_days     = 0,
   bin_interval_seconds = 21600,
   max_num_bins          = 1000,
   print_table           = .true.
   /
```

This is the 'table' part of the run-time output:

```
Requesting  5  assimilation periods.

epoch      1  start day=148135, sec=1
epoch      1    end day=148135, sec=21600
epoch      1  start 2006 Aug 01 00:00:01
epoch      1    end 2006 Aug 01 06:00:00

epoch      2  start day=148135, sec=21601
epoch      2    end day=148135, sec=43200
epoch      2  start 2006 Aug 01 06:00:01
epoch      2    end 2006 Aug 01 12:00:00

epoch      3  start day=148135, sec=43201
epoch      3    end day=148135, sec=64800
epoch      3  start 2006 Aug 01 12:00:01
epoch      3    end 2006 Aug 01 18:00:00

epoch      4  start day=148135, sec=64801
epoch      4    end day=148136, sec=0
epoch      4  start 2006 Aug 01 18:00:01
epoch      4    end 2006 Aug 02 00:00:00

epoch      5  start day=148136, sec=1
epoch      5    end day=148136, sec=21600
epoch      5  start 2006 Aug 02 00:00:01
epoch      5    end 2006 Aug 02 06:00:00
```

Notice that the leading edge of an assimilation window/bin/epoch/period is actually 1 second **after** the specified start time. This is consistent with the way DART has always worked. If you specify assimilation windows that fully occupy the temporal continuum, there has to be some decision at the edges. An observation precisely ON the edge should only participate in one assimilation window. Historically, DART has always taken observations precisely on an edge to be part of the subsequent assimilation cycle. The smallest amount of time representable to DART is 1 second, so the smallest possible delta is added to one of the assimilation edges.

6.120.3 Other modules used

```
location_mod
netcdf
obs_def_mod
obs_kind_mod
obs_sequence_mod
schedule_mod
time_manager_mod
typeSizes
types_mod
utilities_mod
```

Naturally, the program must be compiled with support for the observation types contained in the observation sequence files, so `preprocess` must be run to build appropriate `obs_def_mod` and `obs_kind_mod` modules - which may need specific `obs_def_?????.f90` files.

6.120.4 Files

Run-time

- `input.nml` is used for `obs_seq_to_netcdf_nml` and `schedule_nml`.
- `obs_epoch_xxx.nc` is a netCDF output file for assimilation period 'xxx'. Each observation copy is preserved - as are any/all QC values/copies.
- `dart_log.out` list directed output from the obs_seq_to_netcdf.

Related Matlab functions

- `diagnostics/matlab/read_obs_netcdf.m` reads the netcdf files and returns a structure with easy-to-plot components. More on that in the 'Usage' section below.
- `diagnostics/matlab/plot_obs_netcdf.m` may be used to explore the spatial distribution of observations and their values. More on that in the 'Usage' section below.
- `diagnostics/matlab/plot_obs_netcdf_diffs.m` will take the difference between any two observation copies and plot the spatial distribution and value of the difference. Useful for exploring the bias between 'observation' and 'prior ensemble mean', for example. Again, more on that in the 'Usage' section below.

Discussion of obs_epoch_xxx.nc structure

This might be a good time to review the basic observation sequence file structure. The only thing missing in the netcdf files is the 'shared' metadata for observations (e.g. GPS occultations). The observation locations, values, qc flags, error variances, etc., are all preserved in the netCDF files. The intent is to provide everything you need to make sensible plots of the observations. Some important aspects are highlighted.

```
[shad] % ncdump -v QCMetaData,CopyMetaData,ObsTypesMetaData obs_epoch_001.nc
netcdf obs_epoch_001 {
dimensions:
        linelen = 129 ;
        nlines = 104 ;
        stringlength = 32 ;
        copy = 7 ;
        qc_copy = 2 ;
        location = 3 ;
        ObsTypes = 58 ;
        ObsIndex = UNLIMITED ; // (4752 currently)
variables:
        int copy(copy) ;
                copy:explanation = "see CopyMetaData" ;
        int qc_copy(qc_copy) ;
                qc_copy:explanation = "see QCMetaData" ;
        int ObsTypes(ObsTypes) ;
                ObsTypes:explanation = "see ObsTypesMetaData" ;
        char ObsTypesMetaData(ObsTypes, stringlength) ;
                ObsTypesMetaData:long_name = "DART observation types" ;
                ObsTypesMetaData:comment = "table relating integer to observation␣
↪type string" ;
```

(continues on next page)

(continued from previous page)

```
        char QCMetaData(qc_copy, stringlength) ;
                QCMetaData:long_name = "quantity names" ;
        char CopyMetaData(copy, stringlength) ;
                CopyMetaData:long_name = "quantity names" ;
        char namelist(nlines, linelen) ;
                namelist:long_name = "input.nml contents" ;
        int ObsIndex(ObsIndex) ;
                ObsIndex:long_name = "observation index" ;
                ObsIndex:units = "dimensionless" ;
        double time(ObsIndex) ;
                time:long_name = "time of observation" ;
                time:units = "days since 1601-1-1" ;
                time:calendar = "GREGORIAN" ;
                time:valid_range = 1.15740740740741e-05, 0.25 ;
        int obs_type(ObsIndex) ;
                obs_type:long_name = "DART observation type" ;
                obs_type:explanation = "see ObsTypesMetaData" ;
                location:units = "deg_Lon deg_Lat vertical" ;
        double observations(ObsIndex, copy) ;
                observations:long_name = "org observation, estimates, etc." ;
                observations:explanation = "see CopyMetaData" ;
                observations:missing_value = 9.96920996838687e+36 ;
        int qc(ObsIndex, qc_copy) ;
                qc:long_name = "QC values" ;
                qc:explanation = "see QCMetaData" ;
        double location(ObsIndex, location) ;
                location:long_name = "location of observation" ;
                location:storage_order = "Lon Lat Vertical" ;
                location:units = "degrees degrees which_vert" ;
        int which_vert(ObsIndex) ;
                which_vert:long_name = "vertical coordinate system code" ;
                which_vert:VERTISUNDEF = -2 ;
                which_vert:VERTISSURFACE = -1 ;
                which_vert:VERTISLEVEL = 1 ;
                which_vert:VERTISPRESSURE = 2 ;
                which_vert:VERTISHEIGHT = 3 ;

// global attributes:
                :creation_date = "YYYY MM DD HH MM SS = 2009 05 01 16 51 18" ;
                :obs_seq_to_netcdf_source = "$url: http://subversion.ucar.edu/DAReS/DART/trunk/obs_sequence/obs_seq_to_netcdf.f90 $" ;
                :obs_seq_to_netcdf_revision = "$revision: 4272 $" ;
                :obs_seq_to_netcdf_revdate = "$date: 2010-02-12 14:26:40 -0700 (Fri, 12 Feb 2010) $" ;
                :obs_seq_file_001 = "bgrid_solo/work/01_01/obs_seq.final" ;
data:

 ObsTypesMetaData =
  "RADIOSONDE_U_WIND_COMPONENT     ",
  "RADIOSONDE_V_WIND_COMPONENT     ",
  "RADIOSONDE_SURFACE_PRESSURE     ",
  "RADIOSONDE_TEMPERATURE          ",
  "RADIOSONDE_SPECIFIC_HUMIDITY    ",
  ...
  yeah, yeah, yeah ... we're very impressed ...
  ...
  "VORTEX_PMIN                     ",
```

(continues on next page)

(continued from previous page)

```
  "VORTEX_WMAX                          " ;

 QCMetaData =
  "Quality Control                      ",
  "DART quality control                 " ;

 CopyMetaData =
  "observations                         ",
  "truth                                ",
  "prior ensemble mean                  ",
  "posterior ensemble mean              ",
  "prior ensemble spread                ",
  "posterior ensemble spread            ",
  "observation error variance           " ;
}
```

So, first off, the UNLIMITED dimension is not 'time'. It's simply the number of observations - a coordinate variable called `ObsIndex`. The `observations` variable is a 2D array - each column is a 'copy' of the observation. The interpretation of the column is found in the `CopyMetaData` variable. Same thing goes for the `qc` variable - each column is defined by the `QCMetaData` variable.

The `Obs_Type` variable is crucial. Each observation has an integer code to define the specific . . . DART observation type. In our example - lets assume that observation number 10 (i.e. ObsIndex == 10) has an `obs_type` of 3 [i.e. obs_type(10) = 3]. Since `ObsTypesMetaData(3) == "RADIOSONDE_SURFACE_PRESSURE"`, we know that any/all quantities where ObsIndex == 10 pertain to a radiosonde surface pressure observation.

6.120.5 Usage

Obs_seq_to_netcdf

`obs_seq_to_netcdf` is built and run in `/DART/observations/utilities/threed_sphere` or `/DART/observations/utilities/oned` or in the same way as the other DART components. That directory is intentionally designed to hold components that are model-insensitive. Essentially, we avoid having to populate every `model` directory with identical `mkmf_obs_seq_to_netcdf` and `path_names_obs_seq_to_netcdf` files. After the program has been run, `/DART/observations/utilities/threed_sphere/plot_obs_netcdf.m` can be run to plot the observations. Be aware that the `ObsTypesMetaData` list is all known observation types and not only the observation types in the netCDF file.

Example

```
&schedule_nml
   calendar        = 'Gregorian',
   first_bin_start =  2006, 8, 1, 3, 0, 0 ,
   first_bin_end   =  2006, 8, 1, 9, 0, 0 ,
   last_bin_end    =  2006, 8, 3, 3, 0, 0 ,
   bin_interval_days    = 0,
   bin_interval_seconds = 21600,
   max_num_bins         = 1000,
   print_table          = .true.
```

(continues on next page)

(continued from previous page)

```
   /

&obs_seq_to_netcdf_nml
   obs_sequence_name = '',
   obs_sequence_list = 'olist',
   append_to_netcdf  = .false.,
   lonlim1    =    0.0,
   lonlim2    =  360.0,
   latlim1    =  -80.0,
   latlim2    =   80.0,
   verbose    = .false.
   /
```

> *cat olist* /users/thoar/temp/obs_0001/obs_seq.final /users/thoar/temp/obs_0002/obs_seq.final /users/thoar/temp/obs_0003/obs_seq.final

Here is the pruned run-time output. Note that multiple input observation sequence files are queried and the routine ends (in this case) when the first observation time in a file is beyond the last time of interest.

```
 --------------------------------------
 Starting ... at YYYY MM DD HH MM SS =
                 2009  5 15  9  0 23
 Program obs_seq_to_netcdf
 --------------------------------------

 Requesting            8  assimilation periods.

epoch      1  start day=148135, sec=10801
epoch      1    end day=148135, sec=32400
epoch      1  start 2006 Aug 01 03:00:01
epoch      1    end 2006 Aug 01 09:00:00

epoch      2  start day=148135, sec=32401
epoch      2    end day=148135, sec=54000
epoch      2  start 2006 Aug 01 09:00:01
epoch      2    end 2006 Aug 01 15:00:00

epoch      3  start day=148135, sec=54001
epoch      3    end day=148135, sec=75600
epoch      3  start 2006 Aug 01 15:00:01
epoch      3    end 2006 Aug 01 21:00:00

epoch      4  start day=148135, sec=75601
epoch      4    end day=148136, sec=10800
epoch      4  start 2006 Aug 01 21:00:01
epoch      4    end 2006 Aug 02 03:00:00

epoch      5  start day=148136, sec=10801
epoch      5    end day=148136, sec=32400
epoch      5  start 2006 Aug 02 03:00:01
epoch      5    end 2006 Aug 02 09:00:00

epoch      6  start day=148136, sec=32401
epoch      6    end day=148136, sec=54000
epoch      6  start 2006 Aug 02 09:00:01
epoch      6    end 2006 Aug 02 15:00:00
```

(continues on next page)

(continued from previous page)

```
epoch      7  start day=148136, sec=54001
epoch      7    end day=148136, sec=75600
epoch      7  start 2006 Aug 02 15:00:01
epoch      7    end 2006 Aug 02 21:00:00

epoch      8  start day=148136, sec=75601
epoch      8    end day=148137, sec=10800
epoch      8  start 2006 Aug 02 21:00:01
epoch      8    end 2006 Aug 03 03:00:00

 obs_seq_to_netcdf  opening /users/thoar/temp/obs_0001/obs_seq.final

 num_obs_in_epoch (           1 ) =       103223
 InitNetCDF  obs_epoch_001.nc is fortran unit            5
 num_obs_in_epoch (           2 ) =       186523
 InitNetCDF  obs_epoch_002.nc is fortran unit            5
 num_obs_in_epoch (           3 ) =       110395
 InitNetCDF  obs_epoch_003.nc is fortran unit            5
 num_obs_in_epoch (           4 ) =       191957
 InitNetCDF  obs_epoch_004.nc is fortran unit            5

 obs_seq_to_netcdf  opening /users/thoar/temp/obs_0002/obs_seq.final

 num_obs_in_epoch (           5 ) =        90683
 InitNetCDF  obs_epoch_005.nc is fortran unit            5
 num_obs_in_epoch (           6 ) =       186316
 InitNetCDF  obs_epoch_006.nc is fortran unit            5
 num_obs_in_epoch (           7 ) =       109465
 InitNetCDF  obs_epoch_007.nc is fortran unit            5
 num_obs_in_epoch (           8 ) =       197441
 InitNetCDF  obs_epoch_008.nc is fortran unit            5

 obs_seq_to_netcdf  opening /users/thoar/temp/obs_0003/obs_seq.final

 --------------------------------------
 Finished ... at YYYY MM DD HH MM SS =
                 2009  5 15  9  2 56
 $url: http://subversion.ucar.edu/DAReS/DART/trunk/obs_sequence/obs_seq_to_netcdf.f90
↪$
 $revision: 4272 $
 $date: 2010-02-12 14:26:40 -0700 (Fri, 12 Feb 2010) $
 --------------------------------------
```

Matlab helper functions

DART uses the snctools set of functions. Our m-file `DART/diagnostics/matlab/read_obs_netcdf` uses the `snctools` toolbox.

You will need the 'normal' `DART/matlab` functions available to Matlab, so be sure your MATLABPATH is set such that you have access to `get_copy_index` as well as `nc_varget` and . . .

This generally means your MATLABPATH should look something like:

```
addpath('replace_this_with_the_real_path_to/DART/matlab')
addpath('replace_this_with_the_real_path_to/DART/diagnostics/matlab')
```

(continues on next page)

(continued from previous page)

```
addpath('some_other_netcdf_install_dir/mexnc','-BEGIN')
addpath('some_other_netcdf_install_dir/snctools')
```

On my systems, I've bundled those last 2 commands into a function called `ncstartup.m` which is run every time I start Matlab (because it is in my `~/matlab/startup.m`)

As is standard practice, the instructions for using the Matlab scripts `plot_obs_netcdf` and `plot_obs_netcdf_diffs` are available by using the Matlab 'help' facility (i.e. *help plot_obs_netcdf*). A quick discussion of them here still seems appropriate. If you run the following Matlab commands with an `obs_sequence_001.nc` file you cannot possibly have:

```
>> help plot_obs_netcdf
fname         = 'obs_sequence_001.nc';
ObsTypeString = 'RADIOSONDE_U_WIND_COMPONENT';
region        = [0 360 -90 90 -Inf Inf];
CopyString    = 'NCEP BUFR observation';
QCString      = 'DART quality control';
maxQC         = 2;
verbose       = 1;

obs = plot_obs_netcdf(fname, ObsTypeString, region, CopyString, QCString, maxQC,␣
↪verbose);

>> fname         = 'obs_sequence_001.nc';
>> ObsTypeString = 'RADIOSONDE_U_WIND_COMPONENT';
>> region        = [0 360 -90 90 -Inf Inf];
>> CopyString    = 'NCEP BUFR observation';
>> QCString      = 'DART quality control';
>> maxQC         = 2;
>> verbose       = 1;
>> obs = plot_obs_netcdf(fname, ObsTypeString, region, CopyString, QCString, maxQC,␣
↪verbose);

N =  3336 RADIOSONDE_U_WIND_COMPONENT  obs (type   1) between levels 550.00 and␣
↪101400.00
N =  3336 RADIOSONDE_V_WIND_COMPONENT  obs (type   2) between levels 550.00 and␣
↪101400.00
N =    31 RADIOSONDE_SURFACE_PRESSURE  obs (type   3) between levels 0.00 and 1378.00
N =  1276 RADIOSONDE_TEMPERATURE       obs (type   4) between levels 550.00 and␣
↪101400.00
N =   691 RADIOSONDE_SPECIFIC_HUMIDITY obs (type   5) between levels 30000.00 and␣
↪101400.00
N = 11634 AIRCRAFT_U_WIND_COMPONENT    obs (type   6) between levels 17870.00 and␣
↪99510.00
N = 11634 AIRCRAFT_V_WIND_COMPONENT    obs (type   7) between levels 17870.00 and␣
↪99510.00
N =  8433 AIRCRAFT_TEMPERATURE         obs (type   8) between levels 17870.00 and␣
↪76710.00
N =  6993 ACARS_U_WIND_COMPONENT       obs (type  10) between levels 17870.00 and␣
↪76680.00
N =  6993 ACARS_V_WIND_COMPONENT       obs (type  11) between levels 17870.00 and␣
↪76680.00
N =  6717 ACARS_TEMPERATURE            obs (type  12) between levels 17870.00 and␣
↪76680.00
```

(continues on next page)

(continued from previous page)

```
N = 20713 SAT_U_WIND_COMPONENT         obs (type  22) between levels 10050.00 and␣
↪99440.00
N = 20713 SAT_V_WIND_COMPONENT         obs (type  23) between levels 10050.00 and␣
↪99440.00
N =   723 GPSRO_REFRACTIVITY           obs (type  46) between levels 220.00 and 12000.
↪00
NCEP BUFR observation is copy   1
DART quality control is copy   2
Removing 993 obs with a DART quality control value greater than 2.000000
```

you get the plots at the top of this document. If you have a relatively new version of Matlab, you can dynamically rotate the 3D view … coooool. Even spiffier, if you click on the observations (try the BAD observations), Matlab reports the lat/lon/level of these observations. At least R2008b does, I haven't tried it with all the other variants.

The vertical levels are reported so you can restrict the area of interest with the 'region' variable [minlon maxlon minlat maxlat minlevel maxlevel]. Only the observations with a QC value less than or equal to 'maxQC' are plotted in 'Figure 1'. Note the values of 'QCString' and 'CopyString' must match some value of `QCMetaData` and `CopyMetaData`, respectively. If you're not so keen on a 3D plot, simply change the view to be directly 'overhead':

```
>> view(0,90)
```

And if you act today, we'll throw in a structure containing the selected data AT NO EXTRA CHARGE.

```
>> obs
obs =
            fname: 'obs_sequence_001.nc'
    ObsTypeString: 'RADIOSONDE_U_WIND_COMPONENT'
           region: [0 360 -90 90 -Inf Inf]
       CopyString: 'NCEP BUFR observation'
         QCString: 'DART quality control'
            maxQC: 2
          verbose: 1
             lons: [2343x1 double]
             lats: [2343x1 double]
                z: [2343x1 double]
              obs: [2343x1 double]
             Ztyp: [2343x1 double]
               qc: [2343x1 double]
         numbadqc: 993
           badobs: [1x1 struct]
```

If there are observations with QC values above that defined by `maxQC` there will be a `badobs` structure as a component in the `obs` structure.

6.120.6 References

1. none

6.120.7 Private components

N/A

6.121 program `obs_common_subset`

6.121.1 Overview

This specialized tool allows you to select subsets of observations from two or more observation sequence files output from `filter`. It creates a new set of output observation sequence files containing only the observations which were successfully assimilated in all experiments.

Experiments using the same input observation sequence file but with different configurations (e.g. different inflation values, different localization radii, etc) can assimilate different numbers of the available observations. In that case there will be differences in the diagnostic plots which are not directly relatable to the differences in the quality of the assimilation. If this tool is run on the `obs_seq.final` files from all the experiments and then the diagnostics are generated, only the observations which were assimilated in all experiments will contribute to the summary statistics. A more direct comparison can be made and improvements can be correctly attributed to the differences in the experimental parameters.

This tool is intended to be used when comparing the results from a group of related experiments in which **the exact same input observation sequence file** is used for all runs. The tool cannot process observation sequence files which differ in anything other than whether an observation was successfully assimilated/evaluated or not. Note that it is fine to add or remove observation types from the `assimilate_these_obs_types` or `evaluate_these_obs_types` namelist items for different experiments. The output observation sequence files will still contain an identical list of observations, with some marked with a DART QC indicating 'not assimilated because of namelist control'.

See the two experiment diagnostic plot documentation for Matlab scripts supplied with DART to directly compare the observation diagnostic output from multiple experiments (it does more than two, the script has a poor name).

This is one of a set of tools which operate on observation sequence files. For a more general purpose tool see the *program obs_sequence_tool*, and for a more flexible selection tool see the obs_selection_tool.

Creating an input filelist

One of the inputs to this tool is a list of filenames to compare. The filenames can be directly in the namelist file, or they can be in a set of separate text files. The latter may be easier when there are more than just a few files to compare.

For experiments where there are multiple job steps, and so multiple output observation sequence files per experiment, the input to this tool would then be a list of lists of filenames. Each set of names must be put into a text file with each filename on a separate line.

If each experiment was run in a different set of directories, and if a list of observation sequence filenames was made with the `ls` command:

```
> ls exp1/*/obs_seq.final > exp1list
> cat exp1list
exp1/step1/obs_seq.final
```

(continues on next page)

(continued from previous page)

```
exp1/step2/obs_seq.final
exp1/step3/obs_seq.final
exp1/step4/obs_seq.final
> ls exp2/*/obs_seq.final > exp2list
> cat exp2list
exp2/step1/obs_seq.final
exp2/step2/obs_seq.final
exp2/step3/obs_seq.final
exp2/step4/obs_seq.final
> ls exp3/*/obs_seq.final > exp3list
> cat exp2list
exp3/step1/obs_seq.final
exp3/step2/obs_seq.final
exp3/step3/obs_seq.final
exp3/step4/obs_seq.final
```

Then the namelist entries would be:

```
filename_seq = ''
filename_seq_list = 'exp1list', 'exp2list', exp3list'
num_to_compare_at_once = 3
```

6.121.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_common_subset_nml
 num_to_compare_at_once = 2,
 filename_seq           = '',
 filename_seq_list      = '',
 filename_out_suffix    = '.common' ,
 print_every            = 10000,
 dart_qc_threshold      = 3,
 calendar               = 'Gregorian',
 print_only             = .false.,
 eval_and_assim_can_match = .false.,
/
```

Item	Type	Description
num_to_compare_at_once	integer	Number of observation sequence files to compare together at a time. Most commonly the value is 2, but can be any number. If more than this number of files are listed as inputs, the tool will loop over the list N files at a time.
filename_seq	character(len=256), dimension(5000)	The array of names of the observation sequence files to process. If more than N files (where N is num_to_compare_at_once) are listed, they should be ordered so the first N files are compared together, followed by the next set of N files, etc. You can only specify one of filename_seq OR filename_seq_list, not both.
filename_seq_list	character(len=256), dimension(100)	An alternative way to specify the list of input observation sequence files. Give a list of N filenames which contain, one per line, the names of the observation sequence files to process. There should be N files specified (where N is num_to_compare_at_once), and the first observation sequence filename listed in each file will be compared together, then the second, until the lists are exhausted. You can only specify one of filename_seq OR filename_seq_list, not both.
filename_out_suffix	character(len=32)	A string to be appended to each of the input observation sequence file names to create the output filenames.
print_every	integer	To indicate progress, a count of the successfully processed observations is printed every Nth set of obs. To decrease the output volume set this to a larger number. To disable this output completely set this to -1.
dart_qc_threshold	integer	Observations with a DART QC value larger than this threshold will be discarded. Note that this is the QC value set by `filter` to indicate the outcome of trying to assimilate an observation. This is not related to the incoming data QC. For an observation which was successfully assimilated or evaluated in both the Prior and Posterior this should be set to 1. To also include observations which were successfully processed in the Prior but not the Posterior, set to 3. To ignore the magnitude of the DART QC values and keep observations only if the DART QCs match, set this to any value higher than 7.
calendar	character(len=32)	Set to the name of the calendar; only controls the printed output for the dates of the first and last observations in the file. Set this to "no_calendar" if the observations are not using any calendar.
print_only	logical	If .TRUE. do not create the output files, but print a summary of the number and types of each observation in each of the input and output files.
eval_and_assim_can_match	logical	Normally .FALSE. . If .TRUE. then observations which were either successfully evaluated OR assimilated will match and are kept.

6.121.3 Building

Most `$DART/models/*/work` directories will build the tool along with other executable programs. It is also possible to build the tool in the `$DART/observations/utilities` directory. The `preprocess` program must be built and run first, to define what set of observation types will be supported. See the *PROGRAM preprocess* for more details on how to define the list and run it. The combined list of all observation types which will be encountered over all input files must be in the preprocess input list. The other important choice when building the tool is to include a compatible locations module. For the low-order models, the `oned` module should be used; for real-world observations, the `threed_sphere` module should be used.

Generally the directories where executables are built will include a "quickbuild.csh" script which will build and run preprocess and then build the rest of the executables. The "input.nml" namelists will need to be edited to include all the required observation types first.

6.121.4 Modules used

```
types_mod
utilities_mod
time_manager_mod
obs_def_mod
obs_sequence_mod
```

6.121.5 Files

- `input.nml`
- The input files specified in the `filename_seq` or `filename_seq_list` namelist variable.
- The output files are specified by appending the string from the `filename_out_suffix` namelist item to the input filenames.

6.121.6 References

- none

6.122 MODULE ensemble_manager_mod

6.122.1 Overview

Manages storage and a number of operations for multiple copies of a vector. The most obvious use is to manage ensembles of model state vectors. In this case, the number of copies stored for each state vector element is the ensemble size plus one or more additional copies like the mean, variance, associated inflation values, etc. The ensemble_manager provides routines to compute the mean and variance of a subset of the copies, to track the time associated with the copies, and to write and read restart files. Most importantly, it provides a capability to do transposes between two storage representations of an ensemble. In one representation, each process stores all copies of a subset of the state variables while in the other, each process stores all of the state variables for a subset of copies. The ensemble manager is also used to manage ensembles of observation priors and quality control and ensembles of forward observation operator error status.

The ensemble manager interacts strongly with the multiple process capability of the Message Passing Interface (MPI) libraries. It is used to partition the data so each MPI process stores only a subset of the copies and variables, dividing the data as evenly as possible across the processes. At no time during the execution does any one process have to store the entire dataset for all ensemble members (unless running in serial mode without MPI, or if running with 1 MPI task).

The ensemble manager is set of general purpose data management routines. For run-time efficiency, the derived type information is not marked private which means other modules can directly manipulate the data arrays. However it means much care must be taken to access the most recently updated representation of the data, either the copies or variables arrays.

A set of sanity check routines have been added to track the last modified version of the data: the copies array or the vars array. Before directly reading or writing these arrays call one of the 'prepare' routines to indicate what kind of data access you are about to make. If the most recently updated data is not as expected an error message will occur.

After the direct access if the following operations detect that the data they are operating on is not the most recently updated they will print an error message. Routines inside the ensemble manager that alter the copies or vars will set the state automatically so these routines are only necessary to call if you are directly accessing the copies or vars arrays from outside the ensemble manager.

6.122.2 Namelist

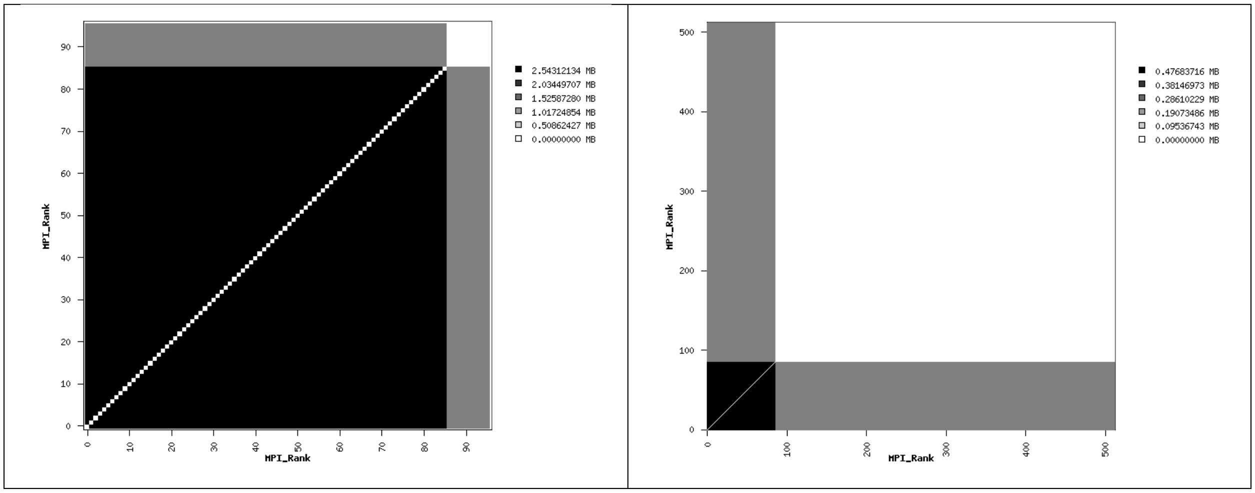
This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&ensemble_manager_nml
   layout                      = 1
   tasks_per_node              = 1
   communication_configuration = 1
   debug                       = .false.
  /
```

Item	Type	Description
layout	integer	Determines the logical process (PE) layout across MPI tasks. 1 is PE = MPI task. 2 is a round-robin layout around the nodes. Layout 2 results in a more even usage of memory across nodes. This may allow you to run with a larger state vector without hitting the memory limit of the node. It may give a slight (5%) increase in performance, but this is machine dependent. It has no effect on serial runs.
tasks_per_node	integer	The number of MPI tasks per hardware node is generally fixed when a batch job is submitted. This namelist item tells the ensemble manager what the user selected at that time. Once a program is running the code has no control to change how MPI tasks are assigned to physical CPUs. This number is used only if layout = 2, and it allows the code spread high-memory-use PEs to different hardware nodes by assigning them in a round-robin order. The job will still run if this number does not match the real "tasks_per_node" at the hardware level, but it may run out of memory if the mismatch causes multiple high-memory-use tasks to be run on the same node.
communication_configuration	integer	For most users, the default value of 1 is the best choice. However there are multiple strategies for the internal MPI communication patterns (see *Note below). Values from 1 to 4 select different options; try the various options to see if one might be faster than the others.
debug	logical	If true print debugging information.

**Note about MPI communication flags:*

The communication_configuration flags select various combinations of the internal settings for use_copy2var_send_loop and use_var2copy_rec_loop. These flags change the order of the MPI send and MPI receives in the the routines all_copies_to_all_vars and all_vars_to_all_copies. The figures below show the data transferred between tasks for an 80 member ensemble. The left figure is using 96 tasks, the right figure is using 512 tasks. As the number of tasks increases, the 'all to all' data transfer becomes a 'some to all, all to some' transfer and the order of MPI send and MPI receives becomes increasingly important. The default values give a performance advantage as the number of tasks becomes much greater than the the ensemble size. However, for small numbers of tasks, i.e. less than the ensemble size, changing the default values may improve performance.



6.122.3 Other modules used

```
types_mod
utilities_mod
assim_model_mod
time_manager_mod
random_seq_mod
mpi_utilities_mod
sort_mod
```

6.122.4 Public interfaces

use ensemble_manager_mod, only :	init_ensemble_manager
	read_ensemble_restart
	write_ensemble_restart
	get_copy
	put_copy
	broadcast_copy
	set_ensemble_time
	get_ensemble_time
	end_ensemble_manager
	duplicate_ens
	get_my_num_copies
	get_my_copies
	get_my_num_vars
	get_my_vars
	get_copy_owner_index
	get_var_owner_index
	all_vars_to_all_copies
	all_copies_to_all_vars
	compute_copy_mean
	compute_copy_mean_sd
	compute_copy_mean_var
	prepare_to_write_to_vars
	prepare_to_write_to_copies
	prepare_to_read_from_vars
	prepare_to_read_from_copies
	prepare_to_update_vars
	prepare_to_update_copies
	print_ens_handle
	map_pe_to_task

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

```
type ensemble_type
   !DIRECT ACCESS INTO STORAGE IS ALLOWED; BE CAREFUL
   integer :: num_copies
   integer :: num_vars
   integer :: my_num_copies
   integer :: my_num_vars
   integer, pointer :: my_copies(:)
   integer, pointer :: my_vars(:)
   ! Storage in next line is to be used when each PE has all copies of subset of vars
   real(r8), pointer :: copies(:, :)  ! Dimensioned (num_copies, my_num_vars)
   ! Storage on next line is used when each PE has subset of copies of all vars
   real(r8), pointer :: vars(:, :)    ! Dimensioned (num_vars, my_num_copies)
   ! Time is only related to var complete
   type(time_type), pointer :: time(:)
   integer :: distribution_type
   integer :: valid     ! copies modified last, vars modified last, both same
   integer :: id_num
   integer, allocatable :: task_to_pe_list(:) ! List of tasks
   integer, allocatable :: pe_to_task_list(:) ! List of tasks
   ! Flexible my_pe, layout_type which allows different task layouts for different␣
↪ensemble handles
   integer :: my_pe
   integer :: layout_type
end type ensemble_type
```

Provides a handle for an ensemble that manages copies of a vector. For efficiency, the type internals are not private and direct access to the storage arrays is used throughout DART.

Component	Description
num_copies	Global number of copies of the vector.
num_vars	Global number of elements (variables) in the vector.
my_num_copies	Number of copies stored by this process.
my_num_vars	Number of variables stored by this process.
my_copies	Dimensioned to size my_num_copies. Contains a list of the global indices of copies stored by this process.
my_vars	Dimensioned to size my_num_vars. Contains a list of the global indices of variables stored by this process.
copies	Dimensioned (num_copies, my_num_vars). Storage for all copies of variables stored by this process.
vars	Dimensioned (num_vars, my_num_copies). Storage for all variables of copies stored by this process.
time	Dimensioned my_num_copies. A time_type that stores time associated with a given copy of the vector.
distribution_type	Does nothing at present. Can be used for future releases to control the layout of different copies and variables in storage.
valid	Flag to track whether the copies array has the most recently updated data, the vars array is most recently modified, or if both the arrays have identical data, like after a transpose.
id_num	Internal number unique to each ensemble handle, used for debugging purposes.
task_to_pe_list	Mapping from MPI task number to logical Processing Element (PE) number. Enables different assignment of MPI tasks to PEs. If the number of MPI tasks is larger than the number of copies of the vector, when the ensemble is var complete then the first N MPI tasks have allocated 'vars' arrays and the remaining ones do not. Assigning the MPI tasks round-robin to multi-processor nodes can make the memory usage more uniform across nodes, which may allow more MPI tasks per node than the standard layout.
pe_to_task_list	Logical PE to MPI task mapping. See above for more description.
my_pe	The logical PE number for the MPI task.
layout_type	Controls the mapping type between MPI tasks and PEs. Currently type 1 is the standard layout (one-to-one mapping) and type 2 is a round-robin mapping where each node gets a task in turn before assigning a second task to each node, until all tasks are assigned.

call init_ensemble_manager(ens_handle, num_copies, num_vars [, distribution_type_in] [, layout_type])

```
type(ensemble_type), intent(out) :: ens_handle
integer,             intent(in)  :: num_copies
integer,             intent(in)  :: num_vars
integer, optional,   intent(in)  :: distribution_type_in
integer, optional,   intent(in)  :: layout_type
```

Initializes an instance of an ensemble. Storage is allocated and the size descriptions in the ensemble_type are initialized.

ens_handle	Handle for the ensemble being initialized
num_copies	Number of copies of vector.
num_vars	Number of variables in the vector.
distribution_type_in	Controls layout of storage on PEs. Currently only option 1 is supported.
layout_type	Controls layout of MPI tasks on PEs. Type 1 is the default, where MPI tasks are assigned to PEs on a one-to-one basis. Type 2 is a round-robin assignment where each node gets one task before the nodes are assigned a second task. If running with more MPI tasks than num_copies, this can result in a more uniform usage of memory across the nodes.

call read_ensemble_restart(ens_handle, start_copy, end_copy, start_from_restart, file_name [, init_time] [, force_single_file])

```
type(ensemble_type),        intent(inout) :: ens_handle
integer,                    intent(in)    :: start_copy
integer,                    intent(in)    :: end_copy
logical,                    intent(in)    :: start_from_restart
character(len=*),           intent(in)    :: file_name
type(time_type), optional, intent(in)    :: init_time
logical, optional,          intent(in)    :: force_single_file
```

Read in a set of copies of a vector from file file_name. The copies read are place into global copies start_copy:end_copy in the ens_handle. If start_from_restart is false, then only a single copy of the vector is read from the file and then it is perturbed using routines in assim_model_mod to generate the required number of copies. The read can be from a single file that contains all needed copies or from a different file for each copy. This choice is controlled by the namelist entry single_restart_file_in. However, the optional argument force_single_file forces the read to be from a single file if it is present and true. This is used for ensembles that contain the inflation values for state space inflation. If multiple files are to be read, the file names are generated by appending integers to the input file_name. If the input is a single file all reads are done sequentially by process 0 and then shipped to the PE that stores that copy. If the input is multiple files each MPI task reads the copies it stores directly and independently.

ens_handle	Handle of ensemble.
start_copy	Global index of first of continguous set of copies to be read.
end_copy	Global index of last of contiguous set of copies to be read, copies(start_copy:end_copy).
start_from_restart	If true, read all copies from file. If false, read one copy and perturb to get required number.
file_name	Name of file from which to read.
init_time	If present, set time of all copies read to this value.
force_single_file	If present and true, force the read to be from a single file which contains all copies.

call write_ensemble_restart(ens_handle, file_name, start_copy, end_copy [, force_single_file])

```
type(ensemble_type), intent(inout) :: ens_handle
character(len=*),    intent(in)    :: file_name
integer,             intent(in)    :: start_copy
integer,             intent(in)    :: end_copy
logical, optional,   intent(in)    :: force_single_file
```

Writes a set of copies of a vector to file file_name. The copies written are from global copies start_copy:end_copy in the ens_handle. The write can be to a single file or to a different file for each copy. This choice is controlled by the namelist entry single_restart_file_out. However, the optional argument force_single_file forces the write to be to a single file if it is present and true. This is used for ensembles that contain the inflation values for state space inflation. If multiple files are to be written, the file names are generated by appending integers to the input file_name. If the output is a single file all copies are shipped from the PE that stores that copy to process 0, and then written out sequentially. If the output is to multiple files each MPI task writes the copies it stores directly and independently.

`ens_handle`	Handle of ensemble.
`file_name`	Name of file from which to read.
`start_copy`	Global index of first of continguous set of copies to be written.
`end_copy`	Global index of last of contiguous set of copies to be written, copies(start_copy:end_copy).
force_single_file	If present and true, force the write to be to a single file which contains all copies.

call get_copy(receiving_pe, ens_handle, copy, vars [, mtime])

```
integer,                     intent(in)  :: receiving_pe
type(ensemble_type),         intent(in)  :: ens_handle
integer,                     intent(in)  :: copy
real(r8), dimension(:),      intent(out) :: vars
type(time_type), optional,   intent(out) :: mtime
```

Retrieves a copy of the state vector, indexed by the global index copy. The process that is to receive the copy is receiving_pe and the copy is returned in the one dimensional array vars. The time of the copy is also returned if mtime is present. This is generally used for operations, like IO, that require a single processor to do things with the entire state vector. Data is only returned in vars on the receiving PE; vars on all other PEs is unset.

`receiving_pe`	This process ends up with the requested copy of the state vector.
`ens_handle`	Handle for ensemble.
`copy`	The global index of the copy of the state vector that is to be retrieved.
`vars`	One dimensional array in which the requested copy of the state vector is returned. Data is only returned in vars on the receiving PE; vars on all other PEs is unset.
mtime	If present returns the time of the requested copy.

call put_copy(sending_pe, ens_handle, copy, vars [, mtime])

```
integer,                     intent(in)    :: sending_pe
type(ensemble_type),         intent(inout) :: ens_handle
integer,                     intent(in)    :: copy
```

(continues on next page)

(continued from previous page)

```
real(r8), dimension(:),    intent(in)    :: vars
type(time_type), optional, intent(in)    :: mtime
```

Sends a state vector, in vars, from the given process to the process storing the global index copy. The time of the copy is also sent if mtime is present. This is generally used for operations, like IO, that require a single processor to do things with the entire state vector. For instance, if a single process reads in a state vector, it can be shipped to the storing process by this subroutine. Only the data in vars on the sending PE is processed; vars on all other PEs is ignored.

sending_pe	This process sends the copy of the state vector.
ens_handle	Handle for ensemble.
copy	The global index of the copy of the state vector that is to be sent.
vars	One dimensional array in which the requested copy of the state vector is located. Only the data in vars on the sending PE is processed; vars on all other PEs is ignored.
mtime	If present send the time of the copy.

call broadcast_copy(ens_handle, copy, arraydata)

```
type(ensemble_type),    intent(in)   :: ens_handle
integer,                intent(in)   :: copy
real(r8), dimension(:), intent(out)  :: arraydata
```

Finds which PE has the global index copy and broadcasts that copy to all PEs. arraydata is an output on all PEs, even on the PE which is the owner if it is separate storage from the vars array in the ensemble handle. This is a collective routine, which means it must be called by all processes in the job.

ens_handle	Handle for ensemble.
copy	The global index of the copy of the state vector that is to be sent.
arraydata	One dimensional array into which the requested copy of the state vector will be copied on all PEs, including the sending PE.

call set_ensemble_time(ens_handle, indx, mtime)

```
type(ensemble_type), intent(inout) :: ens_handle
integer,             intent(in)    :: indx
type(time_type),     intent(in)    :: mtime
```

Set the time of a copy to the given value. indx in this case is the local copy number for a specific task. get_copy_owner_index() can be called to see if you are the owning task for a given global copy number, and to get the local index number for that copy.

ens_handle	Handle for ensemble.
indx	The local index of the copy of the state vector that is to be set.
mtime	The time to set for this copy.

call get_ensemble_time(ens_handle, indx, mtime)

```
type(ensemble_type), intent(in)   :: ens_handle
integer,             intent(in)   :: indx
type(time_type),     intent(out)  :: mtime
```

Get the time associated with a copy. `indx` in this case is the local copy number for a specific task. get_copy_owner_index() can be called to see if you are the owning task for a given global copy number, and to get the local index number for that copy.

`ens_handle`	Handle for ensemble.
`indx`	The local index of the copy to retrieve the time from.
`mtime`	The returned time value.

call end_ensemble_manager(ens_handle)

```
type(ensemble_type), intent(in)  :: ens_handle
```

Frees up storage associated with an ensemble.

`ens_handle`	Handle for an ensemble.

call duplicate_ens(ens1, ens2, duplicate_time)

```
type(ensemble_type), intent(in)    :: ens1
type(ensemble_type), intent(inout) :: ens2
logical, intent(in)                :: duplicate_time
```

Copies the contents of the vars array from ens1 into ens2. If the num_copies and num_vars are not consistent or if the distribution_type is not consistent, fails with an error. If duplicate_time is true, the times from ens1 are copied over the times of ens2. Only the vars array data is copied from the source to the destination. Transpose the data after duplication if you want to access the copies.

`ens1`	Ensemble handle of ensemble to be copies into ens2. Data from the vars array will be replicated.
`ens2`	Ensemble handle of ensemble into which ens1 vars data will be copied.
`duplicate_time`	If true, copy the times from ens1 into ens2, else leave ens2 times unchanged.

var = get_my_num_copies(ens_handle)

```
integer                              :: get_my_num_copies
type(ensemble_type), intent(in)   :: ens_handle
```

Returns number of copies stored by this process when storing all variables for a subset of copies. Same as num_copies if running with only a single process.

`var`	Returns the number of copies stored by this process when storing all variables for a subset of copies.
`ens_handle`	Handle for an ensemble.

var = get_my_num_vars(ens_handle)

```
integer                            :: get_my_num_vars
type(ensemble_type), intent(in) :: ens_handle
```

Returns number of variables stored by this process when storing all copies of a subset of variables. Same as num_vars if running with only a single process.

`var`	Returns the number of vars stored by this process when storing all copies of a subset of variables.
`ens_handle`	Handle for an ensemble.

call get_my_copies(ens_handle, copies)

```
type(ensemble_type), intent(in) :: ens_handle
integer, intent(out)            :: copies(:)
```

Returns a list of the global copy numbers stored on this process when storing subset of copies of all variables.

`ens_handle`	Handle for an ensemble.
`copies`	List of all copies stored by this process when storing subset of copies of all variables.

call get_my_vars(ens_handle, vars)

```
type(ensemble_type), intent(in) :: ens_handle
integer, intent(out)            :: vars(:)
```

Returns a list of the global variable numbers stored on this process when storing all copies of a subset of variables.

`ens_handle`	Handle for an ensemble.
`vars`	List of all variables stored on this process when storing all copies of a subset of variables.

call get_copy_owner_index(copy_number, owner, owners_index)

```
integer, intent(in)  :: copy_number
integer, intent(out) :: owner
integer, intent(out) :: owners_index
```

Given the global index of a copy number, returns the PE that stores this copy when all variables of a subset of copies are stored and the local storage index for this copy on that process.

`copy_number`	Global index of a copy from an ensemble.
`owner`	Process Element (PE) that stores this copy when each has all variables of a subset of copies.
`owners_index`	Local storage index for this copy on the owning process.

call get_var_owner_index(var_number, owner, owners_index)

```
integer, intent(in)  :: var_number
integer, intent(out) :: owner
integer, intent(out) :: owners_index
```

Given the global index of a variable in the vector, returns the PE that stores this variable when all copies of a subset of variables are stored and the local storage index for this variable on that process.

`var_number`	Global index of a variable in the vector from an ensemble.
`owner`	Process Element (PE) that stores this variable when each has all copies of subset of variables.
`owners_index`	Local storage index for this variable on the owning process.

call all_vars_to_all_copies(ens_handle, label)

```
type(ensemble_type), intent(inout)        :: ens_handle
character(len=*),    intent(in), optional :: label
```

Transposes data from a representation in which each PE has a subset of copies of all variables to one in which each has all copies of a subset of variables. In the current implementation, storage is not released so both representations are always available. However, one representation may be current while the other is out of date.

Different different numbers of copies, different lengths of the vectors, different numbers of PEs and different implementations of the MPI parallel libraries can have very different performance characteristics. The namelist item `communication_configuration` controls one of four possible combinations of the operation order during the transposes. If performance is an issue the various settings on this namelist item can be explored. See the namelist section for more details.

The transpose routines make both representations of the data equivalent until the next update to either the copies or the vars arrays, so either can be used as a data source.

ens_handle	The handle of the ensemble being transposed.
label	A character string label. If present, a timestamp with this label is printed at the start and end of the transpose.

call all_copies_to_all_vars(ens_handle, label)

```
type(ensemble_type), intent(inout)        :: ens_handle
character(len=*),    intent(in), optional :: label
```

Transposes data from a representation in which each processor has all copies of a subset of variables to one in which each has a subset of copies of all variables. In the current implementation, storage is not released so both representations are always available. However, one representation may be current while the other is out of date.

Different different numbers of copies, different lengths of the vectors, different numbers of PEs and different implementations of the MPI parallel libraries can have very different performance characteristics. The namelist item `communication_configuration` controls one of four possible combinations of the operation order during the transposes. If performance is an issue the various settings on this namelist item can be explored. See the namelist section for more details.

The transpose routines make both representations of the data equivalent until the next update to either the copies or the vars arrays, so either can be used as a data source.

ens_handle	The handle of the ensemble being transposed.
label	A character string label. If present, a timestamp with this label is printed at the start and end of the transpose.

call compute_copy_mean(ens_handle, start_copy, end_copy, mean_copy)

```
type(ensemble_type), intent(inout) :: ens_handle
integer,             intent(in)    :: start_copy
integer,             intent(in)    :: end_copy
integer,             intent(in)    :: mean_copy
```

Computes the mean of a contiguous subset of copies starting with global index start_copy and ending with global index ens_copy. Mean is written to global index mean_copy.

When this routine is called the ensemble must have all copies of a subset of the vars. It updates the copies array with the mean, so after this call the copies array data is more current and the vars data is stale.

ens_handle	Handle for an ensemble.
start_copy	Global index of first copy in mean and sd computation.
end_copy	Global index of last copy in mean and sd computation.
mean_copy	Global index of copy into which mean is written.

call compute_copy_mean_sd(ens_handle, start_copy, end_copy, mean_copy, sd_copy)

```
type(ensemble_type), intent(inout) :: ens_handle
integer,             intent(in)    :: start_copy
integer,             intent(in)    :: end_copy
integer,             intent(in)    :: mean_copy
integer,             intent(in)    :: sd_copy
```

Computes the mean and standard deviation of a contiguous subset of copies starting with global index start_copy and ending with global index ens_copy. Mean is written to index mean_copy and standard deviation to index sd_copy.

When this routine is called the ensemble must have all copies of a subset of the vars. It updates the copies arrays with the mean and sd, so after this call the copies array data is more current and the vars data is stale.

`ens_handle`	Handle for an ensemble.
`start_copy`	Global index of first copy in mean and sd computation.
`end_copy`	Global index of last copy in mean and sd computation.
`mean_copy`	Global index of copy into which mean is written.
`sd_copy`	Global index of copy into which standard deviation is written.

call compute_copy_mean_var(ens_handle, start_copy, end_copy, mean_copy, var_copy)

```
type(ensemble_type), intent(inout) :: ens_handle
integer,             intent(in)  :: start_copy
integer,             intent(in)  :: end_copy
integer,             intent(in)  :: mean_copy
integer,             intent(in)  :: var_copy
```

Computes the mean and variance of a contiguous subset of copies starting with global index start_copy and ending with global index ens_copy. Mean is written to index mean_copy and variance to index var_copy.

When this routine is called the ensemble must have all copies of a subset of the vars. It updates the copies arrays with the mean and variance, so after this call the copies array data is more current and the vars data is stale.

`ens_handle`	Handle for an ensemble.
`start_copy`	Global index of first copy in mean and sd computation.
`end_copy`	Global index of last copy in mean and sd computation.
`mean_copy`	Global index of copy into which mean is written.
`var_copy`	Global index of copy into which variance is written.

call prepare_to_update_vars(ens_handle)

```
type(ensemble_type), intent(inout)  :: ens_handle
```

Call this routine before directly accessing the `ens_handle%vars` array when the data is going to be updated, and the incoming vars array should have the most current data representation.

Internally the ensemble manager tracks which of the copies or vars arrays, or both, have the most recently updated representation of the data. For example, before a transpose (`all_vars_to_all_copies()` or `all_copies_to_all_vars()`) the code checks to be sure the source array has the most recently updated representation before it does the operation. After a transpose both representations have the same update time and are both valid.

For efficiency reasons we allow the copies and vars arrays to be accessed directly from other code without going through a routine in the ensemble manager. The "prepare" routines verify that the desired array has the most recently updated representation of the data, and if needed marks which one has been updated so the internal consistency checks have an accurate accounting of the representations.

`ens_handle`	Handle for the ensemble being accessed directly.

call prepare_to_update_copies(ens_handle)

```
type(ensemble_type), intent(inout)  :: ens_handle
```

Call this routine before directly accessing the `ens_handle%copies` array when the data is going to be updated, and the incoming copies array should have the most current data representation.

Internally the ensemble manager tracks which of the copies or vars arrays, or both, have the most recently updated representation of the data. For example, before a transpose (`all_vars_to_all_copies()` or `all_copies_to_all_vars()`) the code checks to be sure the source array has the most recently updated representation before it does the operation. After a transpose both representations have the same update time and are both valid.

For efficiency reasons we allow the copies and vars arrays to be accessed directly from other code without going through a routine in the ensemble manager. The "prepare" routines verify that the desired array has the most recently updated representation of the data, and if needed marks which one has been updated so the internal consistency checks have an accurate accounting of the representations.

`ens_handle`	Handle for the ensemble being accessed directly.

call prepare_to_read_from_vars(ens_handle)

```
type(ensemble_type), intent(inout)  :: ens_handle
```

Call this routine before directly accessing the `ens_handle%vars` array for reading only, when the incoming vars array should have the most current data representation.

Internally the ensemble manager tracks which of the copies or vars arrays, or both, have the most recently updated representation of the data. For example, before a transpose (`all_vars_to_all_copies()` or `all_copies_to_all_vars()`) the code checks to be sure the source array has the most recently updated representation before it does the operation. After a transpose both representations have the same update time and are both valid.

For efficiency reasons we allow the copies and vars arrays to be accessed directly from other code without going through a routine in the ensemble manager. The "prepare" routines verify that the desired array has the most recently updated representation of the data, and if needed marks which one has been updated so the internal consistency checks have an accurate accounting of the representations.

`ens_handle`	Handle for the ensemble being accessed directly.

call prepare_to_read_from_copies(ens_handle)

```
type(ensemble_type), intent(inout)  :: ens_handle
```

Call this routine before directly accessing the `ens_handle%copies` array for reading only, when the incoming copies array should have the most current data representation.

Internally the ensemble manager tracks which of the copies or vars arrays, or both, have the most recently updated representation of the data. For example, before a transpose (`all_vars_to_all_copies()` or `all_copies_to_all_vars()`) the code checks to be sure the source array has the most recently updated representation before it does the operation. After a transpose both representations have the same update time and are both valid.

For efficiency reasons we allow the copies and vars arrays to be accessed directly from other code without going through a routine in the ensemble manager. The "prepare" routines verify that the desired array has the most recently updated representation of the data, and if needed marks which one has been updated so the internal consistency checks have an accurate accounting of the representations.

`ens_handle`	Handle for the ensemble being accessed directly.

call prepare_to_write_to_vars(ens_handle)

```
type(ensemble_type), intent(inout)  :: ens_handle
```

Call this routine before directly accessing the `ens_handle%vars` array for writing. This routine differs from the 'update' version in that it doesn't care what the original data state is. This routine might be used in the case where an array is being filled for the first time and consistency with the data in the copies array is not an issue.

Internally the ensemble manager tracks which of the copies or vars arrays, or both, have the most recently updated representation of the data. For example, before a transpose (`all_vars_to_all_copies()` or `all_copies_to_all_vars()`) the code checks to be sure the source array has the most recently updated representation before it does the operation. After a transpose both representations have the same update time and are both valid.

For efficiency reasons we allow the copies and vars arrays to be accessed directly from other code without going through a routine in the ensemble manager. The "prepare" routines verify that the desired array has the most recently updated representation of the data, and if needed marks which one has been updated so the internal consistency checks have an accurate accounting of the representations.

`ens_handle`	Handle for the ensemble being accessed directly.

call prepare_to_write_to_copies(ens_handle)

```
type(ensemble_type), intent(inout)  :: ens_handle
```

Call this routine before directly accessing the `ens_handle%copies` array for writing. This routine differs from the 'update' version in that it doesn't care what the original data state is. This routine might be used in the case where an array is being filled for the first time and consistency with the data in the vars array is not an issue.

Internally the ensemble manager tracks which of the copies or vars arrays, or both, have the most recently updated representation of the data. For example, before a transpose (`all_vars_to_all_copies()` or `all_copies_to_all_vars()`) the code checks to be sure the source array has the most recently updated representation before it does the operation. After a transpose both representations have the same update time and are both valid.

For efficiency reasons we allow the copies and vars arrays to be accessed directly from other code without going through a routine in the ensemble manager. The "prepare" routines verify that the desired array has the most recently updated representation of the data, and if needed marks which one has been updated so the internal consistency checks have an accurate accounting of the representations.

`ens_handle`	Handle for the ensemble being accessed directly.

6.122.5 Private interfaces

	assign_tasks_to_pes
	calc_tasks_on_each_node
	create_pe_to_task_list
	get_copy_list
	get_max_num_copies
	get_max_num_vars
	get_var_list
	round_robin
	set_up_ens_distribution
	simple_layout
	sort_task_list
	timestamp_message

var = get_max_num_copies(num_copies)

```
integer                :: get_max_num_copies
integer, intent(in)    :: num_copies
```

Returns the largest number of copies that are on any pe when var complete. Depends on distribution_type with only option 1 currently implemented. Used to get size for creating storage to receive a list of the copies on a PE.

`var`	Returns the largest number of copies any an individual PE when var complete.
`num_copies`	Total number of copies in the ensemble.

var = get_max_num_vars(num_vars)

```
integer                :: get_max_num_vars
integer, intent(in)    :: num_vars
```

Returns the largest number of vars that are on any pe when copy complete. Depends on distribution_type with only option 1 currently implemented. Used to get size for creating storage to receive a list of the vars on a PE.

var	Returns the largest number of vars any an individual PE when copy complete.
num_copies	Total number of vars in an ensemble vector.

call set_up_ens_distribution(ens_handle)

```
type(ensemble_type), intent(inout) :: ens_handle
```

Figures out how to lay out the copy complete and vars complete distributions. The distribution_type identifies different options. Only distribution_type 1 is implemented. This puts every Nth var or copy on a given processor where N is the total number of processes.

ens_handle	Handle for an ensemble.

call get_var_list(num_vars, pe, var_list, pes_num_vars)

```
integer,   intent(in)      :: num_vars
integer,   intent(in)      :: pe
integer,   intent(out)     :: var_list(:)
integer,   intent(out)     :: pes_num_vars
```

Returns a list of the vars stored by process pe when copy complete and the number of these vars. var_list must be dimensioned large enough to hold all vars. Depends on distribution_type with only option 1 currently implemented.

call get_copy_list(num_copies, pe, copy_list, pes_num_copies)

```
integer,   intent(in)      :: num_copies
integer,   intent(in)      :: pe
integer,   intent(out)     :: copy_list(:)
integer,   intent(out)     :: pes_num_copies
```

Returns a list of the copies stored by process pe when var complete and the number of these copies. copy_list must be dimensioned large enough to hold all copies. Depends on distribution_type with only option 1 currently implemented.

call timestamp_message(msg [, sync] [, alltasks])

```
character(len=*), intent(in)           :: msg
logical,          intent(in), optional :: sync
logical,          intent(in), optional :: alltasks
```

Write current time and message to stdout and log file. If sync is present and true, sync mpi jobs before printing time. If alltasks is present and true, all tasks print the time. The default is only task 0 prints a timestamp.

`msg`	character string to prepend to the time info
sync	if present and true, execute an MPI_Barrier() to sync all MPI tasks before printing the time. this means the time will be the value of the slowest of the tasks to reach this point.
alltasks	if present and true, have all tasks print out a timestamp. the default is for just task 0 to print. the usual combination is either sync=true and alltasks=false, or sync=false and alltasks=true.

call print_ens_handle(ens_handle, force, label)

```
type(ensemble_type),         intent(in)  :: ens_handle
logical,           optional, intent(in)  :: force
character(len=*), optional, intent(in)  :: label
```

For debugging use, dump the contents of an ensemble handle derived type. If the `debug` namelist item is true, this will print in any case. If `debug` is false, set `force` to true to force printing. The optional string label can help provide context for the output.

`ens_handle`	The derived type to print information about.
`force`	If the `debug` namelist item is false, set this to true to enable printing.
`label`	Optional string label to print to provide context for the output.

call assign_tasks_to_pes(ens_handle, nEns_members, layout_type)

```
type(ensemble_type), intent(inout)    :: ens_handle
integer,             intent(in)       :: nEns_members
integer,             intent(inout)    :: layout_type
```

Calulate the task layout based on the tasks per node and the total number of tasks. Allows the user to spread out the ensemble members as much as possible to balance memory usage between nodes. Possible options: 1. Standard task layout - first n tasks have the ensemble members my_pe = my_task_id() 2. Round-robin on the nodes

`ens_handle`	Handle for an ensemble.

call round_robin(ens_handle)

```
type(ensemble_type), intent(inout)    :: ens_handle
```

Round-robin MPI task layout starting at the first node. Starting on the first node forces pe 0 = task 0. The smoother code assumes task 0 has an ensemble member. If you want to break the assumption that pe 0 = task 0, this routine is a good place to start. Test with the smoother.

`ens_handle`	Handle for an ensemble.

call create_pe_to_task_list(ens_handle)

```
type(ensemble_type), intent(inout)    :: ens_handle
```

Creates the `ens_handle%pe_to_task_list`. `ens_handle%task_to_pe_list` must have been assigned first, otherwise this routine will just return nonsense.

`ens_handle`	Handle for an ensemble.

call calc_tasks_on_each_node(nodes, last_node_task_number)

```
integer, intent(out)  :: last_node_task_number
integer, intent(out)  :: nodes
```

Finds the of number nodes and how many tasks are on the last node, given the number of tasks and the tasks_per_node (ptile). The total number of tasks is num_pes = task_count() The last node may have fewer tasks, for example, if ptile = 16 and the number of mpi tasks = 17

call simple_layout(ens_handle, n)

```
type(ensemble_type), intent(inout) :: ens_handle
integer,             intent(in)    :: n
```

assigns the arrays task_to_pe_list and pe_to_task list for the simple layout where my_pe = my_task_id()

ens_handle Handle for an ensemble.

n size

call sort_task_list(i, idx, n)

```
integer, intent(in)    :: n
integer, intent(inout) :: x(n)   ! array to be sorted
integer, intent(out)   :: idx(n) ! index of sorted array
```

sorts an array and returns the sorted array, and the index of the original array

n size

x(n) array to be sorted

idx(n) index of sorted array

call map_pe_to_task(ens_handle, p)

```
type(ensemble_type), intent(in) :: ens_handle
integer,             intent(in) :: p
```

Return the physical task for my_pe

ens_handle	Handle for an ensemble.
p	The MPI task corresponding to the given PE number

call map_task_to_pe(ens_handle, t)

```
type(ensemble_type), intent(in) :: ens_handle
integer,             intent(in) :: t
```

Return my_pe corresponding to the physical task

ens_handle	Handle for an ensemble.
t	Return the PE corresponding to the given MPI task number.

Files

- input.nml
- State vector restart files, either one for all copies or one per copy.
- State vector output files, either one for all copies or one per copy.

References

1. none

Private components

N/A

6.123 MODULE random_seq_mod

6.123.1 Overview

Provides access to any number of reproducible random sequences. Can sample from uniform, gaussian, two-dimensional gaussian, gamma, inverse gamma, and exponential distributions.

The current random sequence generator is a Fortran version of the GNU Library implementation of the Mersenne Twister algorithm. The original code is in the C language and the conversion to Fortran was done by the DART team.

There are test programs in the `developer_tests/random_seq` directory which show examples of calling these routines. Build and run these tests in the `test` subdirectory.

6.123.2 Other modules used

```
types_mod
utilities_mod
```

6.123.3 Public interfaces

use random_seq_mod, only :	random_seq_type
	init_random_seq
	random_uniform
	random_gaussian
	several_random_gaussians
	twod_gaussians
	random_gamma
	random_inverse_gamma
	random_exponential

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

```
type random_seq_type
   private

   integer     :: mti
   integer(i8) :: mt(624)
   real(r8)    :: lastg
   logical     :: gset

end type random_seq_type
```

This type is used to uniquely identify a sequence. Keeps the state history of the linear congruential number generator. In this implementation it is based on the Mersenne Twister from the GNU Scientific Library.

call init_random_seq(r, [, seed])

```
type(random_seq_type),           intent(inout) :: r
integer,               optional, intent(in)    :: seed
```

Initializes a random sequence for use. This must be called before any random numbers can be generated from this sequence. Any number of independent, reproducible random sequences can be generated by having multiple instances of a random_seq_type. A specified integer seed, optional, can produce a specific 'random' sequence.

r	A random sequence type to be initialized.
seed	A seed for a random sequence.

var = random_uniform(r)

```
real(r8)                             :: random_uniform
type(random_seq_type), intent(inout) :: r
```

Returns a random draw from a uniform distribution on interval [0,1].

random_uniform	A random draw from a Uniform[0,1] distribution.
r	An initialized random sequence type.

var = random_gaussian(r, mean, standard_deviation)

```
real(r8)                             :: random_gaussian
type(random_seq_type), intent(inout) :: r
real(r8),              intent(in)    :: mean
real(r8),              intent(in)    :: standard_deviation
```

Returns a random draw from a Gaussian distribution with the specified mean and standard deviation.

See this Wikipedia page for more explanation about this function.

random_gaussian	A random draw from a gaussian distribution.
r	An initialized random sequence type.
mean	Mean of the gaussian.
standard_deviation	Standard deviation of the gaussian.

call several_random_gaussians(r, mean, standard_deviation, n, rnum)

```
type(random_seq_type),  intent(inout) :: r
real(r8),               intent(in)    :: mean
real(r8),               intent(in)    :: standard_deviation
integer,                intent(in)    :: n
real(r8), dimension(:), intent(out)   :: rnum
```

Returns `n` random samples from a gaussian distribution with the specified mean and standard deviation. Array `rnum` must be at least size `n`.

`r`	An initialized random sequence type.
`mean`	Mean of the Gaussian to be sampled.
`standard_deviation`	Standard deviation of the Gaussian.
`n`	Number of samples to return
`rnum`	The random samples of the Gaussian.

call twod_gaussians(r, mean, cov, rnum)

```
type(random_seq_type),    intent(inout) :: r
real(r8), dimension(2),   intent(in)    :: mean
real(r8), dimension(2,2), intent(in)    :: cov
real(r8), dimension(2),   intent(out)   :: rnum
```

Returns a random draw from a 2D gaussian distribution with the specified mean and covariance.

The algorithm used is from Knuth, exercise 13, section 3.4.1. See this Wikipedia page for more explanation about this function.

`r`	An initialized random sequence type.
`mean`	Mean of 2D gaussian distribution.
`cov`	Covariance of 2D gaussian.
`rnum`	Returned random draw from gaussian.

var = random_gamma(r, rshape, rscale)

```
real(r8)                               :: random_gamma
type(random_seq_type), intent(inout) :: r
real(r8),              intent(in)    :: rshape
real(r8),              intent(in)    :: rscale
```

Returns a random draw from a Gamma distribution with specified `rshape` and `rscale`. Both must be positive.

Note that there are three different parameterizations in common use:

1. With shape parameter (kappa) and scale parameter (theta).
2. With shape parameter (alpha) and rate parameter (beta). Alpha is the same as kappa, and beta is an inverse scale parameter so = 1/.

3. With shape parameter (kappa) and mean parameter (mu). = /, so = /.

This form uses the first parameterization, shape () and scale (). The distribution mean is and the variance is (^{2}). This routine is based on the Gamma(a,b) generator from the GNU Scientific library. See this Wikipedia page for more explanation of the various parameterizations of this function.

random_gamma	A random draw from a gamma distribution.
r	An initialized random sequence type.
rshape	Shape parameter. Often written as either alpha or kappa.
rscale	Scale parameter. Often written as theta. If you have a rate parameter (often beta) pass in (1/rate) for scale.

var = random_inverse_gamma(r, rshape, rscale)

```
real(r8)                                :: random_inverse_gamma
type(random_seq_type), intent(inout) :: r
real(r8),              intent(in)    :: rshape
real(r8),              intent(in)    :: rscale
```

Returns a random draw from an inverse Gamma distribution with the specified `shape` and `scale`. Both must be positive. If you have 'rate' instead of 'scale' pass in (1/rate) for scale.

See this Wikipedia page for more explanation about this function.

random_inverse_gamma	A random draw from an inverse gamma distribution.
r	An initialized random sequence type.
rshape	Shape parameter. Often written as either alpha or kappa.
rscale	Scale parameter. Often written as theta. If you have a rate parameter (often beta) pass in (1/rate) for scale.

var = random_exponential(r, rate)

```
real(r8)                                :: random_exponential
type(random_seq_type), intent(inout) :: r
real(r8),              intent(in)    :: rate
```

Returns a random draw from an exponential distribution with the specified `rate`. If you have a scale parameter (which is the same as the mean, the standard deviation, and the survival parameter), specify (1/scale) for rate.

See this Wikipedia page for more explanation about this function.

random_exponential	A random draw from an exponential distribution.
r	An initialized random sequence type.
rate	Rate parameter. Often written as lambda. If you have a scale parameter pass in (1/scale) for rate.

6.123.4 Namelist

This module has no namelist input.

6.123.5 Files

- NONE

6.123.6 References

1. Knuth, Vol 2.
2. GNU Scientific Library Reference Manual

6.123.7 Private components

	init_ran
	ran_unif
	ran_gauss
	ran_gamma

call init_ran(s, seed)

```
type(random_seq_type), intent(out) :: s
integer,               intent(in)  :: seed
```

Initializes a random sequence with an integer. Any sequence initialized with the same integer will produce the same sequence of pseudo-random numbers.

s	A random sequence to be initialized
seed	An integer seed to start the sequence.

var = ran_unif(s)

```
real(r8)                           :: ran_unif
type(random_seq_type), intent(inout) :: s
```

Generate the next uniform [0, 1] random number in the sequence.

ran_unif	Next uniformly distributed [0, 1] number in sequence.
s	A random sequence.

var = ran_gauss(s)

```
real(r8)                           :: ran_gauss
type(random_seq_type), intent(inout) :: s
```

Generates a random draw from a standard gaussian.

ran_gauss	A random draw from a standard gaussian.
s	A random sequence.

var = ran_gamma(r, rshape, rscale)

```
real(r8)                           :: ran_gamma
type(random_seq_type), intent(inout) :: r
real(r8),              intent(in)    :: rshape
real(r8),              intent(in)    :: rscale
```

Generates a random draw from a Gamma distribution. See notes in the random_gamma() section about (alpha,beta) vs (kappa,theta) vs (kappa,mu) parameterizations. This is transcribed from C code in the GNU Scientific library and keeps the (shape,scale) interface.

ran_gamma	A random draw from a Gamma distribution.
r	A random sequence.
rshape	Shape parameter.
rscale	Scale parameter. (This is the inverse of a rate parameter.)

6.124 MODULE mpi_utilities_mod

6.124.1 Overview

This module provides subroutines which utilize the MPI (Message Passing Interface) parallel communications library. DART does **not** require MPI; to compile without using MPI substitute the `null_mpi_utilities_mod.f90` file for this one. That file contains the same module name and public entry points as this one but implements a serial version of all the routines. However, to be able to run most larger models with a reasonable number of ensemble members (e.g. 30-100) MPI will be needed.

The main DART executable `filter` can be compiled and run as either a serial program or a parallel program. Most work directories in the DART distribution source tree have a `quickbuild.csh` script which can take a `-mpi` or a `-nompi` flag. This flag changes the list of files to be compiled to use either the module which uses the MPI library or the one which makes no MPI calls. No source code changes are required to switch between the two options.

A parallel program generally runs faster and requires less memory per CPU than the serial code. It requires an implementation of the MPI library and run-time system to pass data between different nodes on a parallel cluster or supercomputer. There is a lot of information about MPI on the web. See here for an intro to MPI and parallel programming, and here for downloads and technical help.

Most of the larger models need to be compiled and run with MPI because of limitations on total memory accessible by a single executable. The smaller models (e.g. any of the Lorenz models) can generally be run as a serial program without needing MPI.

The MPI distributions usually include a module named `mpi` which defines the public entry points and the types and names of the routine arguments. However there are build-time options and older distributions which only supply an `mpi.h` include file. If you get a compile-time error about the mpi module being missing, edit the source code in `mpi_utilities/mpi_utilities_mod.f90` and comment out the `use mpi` line and comment in the `include 'mpi.h'` line. The 'use' line must be before the 'contains' line, while the 'include' line must be after, so do not move the existing lines. Just comment them in or out depending on which one you need to use.

To preserve backwards compatibility this code does not require a namelist. However there is a namelist defined in the source file which contains some useful run-time options. To enable it edit the source file in `mpi_utilities/mpi_utilities_mod.f90` and set `use_namelist` to .TRUE. and recompile. The code will then read the namelist described below. Messages printed to the nml output log file will confirm whether the defaults are being used or if the namelist is being read in.

6.124.2 Namelist

The source code defines a namelist, but for backwards compatibility it is not read in unless the source code in `mpi_utilities/mpi_utilities_mod.f90` is edited, the module global variable `use_namelist` is changed from .FALSE. to .TRUE., and then all executables are recompiled.

If enabled, this namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&mpi_utilities_nml
    reverse_task_layout        = .false.
    all_tasks_print            = .false.
    verbose                    = .false.
    async2_verbose             = .false.
    async4_verbose             = .false.
    shell_name                 = ''
    separate_node_sync         = .false.
```

(continues on next page)

(continued from previous page)

```
   create_local_comm          = .true.
   make_copy_before_sendrecv  = .false.
  /
```

Item	Type	Description
reverse_task_layout	logical	The synchronizing mechanism between the job script and the parallel filter in async=4 mode relies on the script and task 0 running on the same node (in the same memory space if the nodes have multiple processors). Some MPI implementations (OpenMPI being the most commonly used one) lay the tasks out so that the last task is on the same node as the script. If the async 4 model advance never starts but there are no error messages, try setting this to .TRUE. before running. See also the 'async4_verbose' flag below.
all_tasks_print	logical	In the parallel filter, informational messages only print from task 0 to avoid N copies of the same messages. Error messages and warnings print no matter which task they occur in. If this variable is set to true, even messages will print from all tasks.
verbose	logical	USE WITH CAUTION! This flag enables debugging print messages for every MPI call - sends, receives, barriers - and is very, very verbose. In most cases the size of the output file will exceed the filesystem limits or will cause the executable to run so slowly that it will not be useful. However in small testcases this can be useful to trace problems.
async2_verbose	logical	Print out messages about the handshaking between filter and the advance model scripts when running in async=2 mode. Not anywhere as verbose as the flag above; in most cases the output volume is reasonable.
async4_verbose	logical	Print out messages about the handshaking between filter and the run script when running in async=4 mode. Not anywhere as verbose as the flag above; in most cases the output volume is reasonable.
shell_name	character(len=129)	If running on compute nodes which do not have the expected default shell for async=2 or async=4 mode, specify the full pathname of the shell to execute the script. Not normally needed on most systems we run on. (However, at least one type of Cray system has this need.)
separate_node_sync	logical	Not supported yet. Will use files to handshake between the filter executable and the run script in async=4 mode when the launch script is not running on any of the same nodes as the filter tasks.
create_local_comm	logical	The DART MPI routines normally create a separate local MPI communicator instead of using MPI_COMM_WORLD. This keeps DART communications separate from any other user code. To use the default world communicator set this to .FALSE. . Normal use should leave this true.
make_copy_before_sendrecv	logical	Workaround for old MPI bug. Should be .false.

6.124.3 Other modules used

```
types_mod
utilities_mod
time_manager_mod
mpi  (or mpif.h if mpi module not available)
```

6.124.4 Public interfaces

use mpi_utilities_mod, only :	initialize_mpi_utilities
	finalize_mpi_utilities
	task_count
	my_task_id
	task_sync
	block_task
	restart_task
	array_broadcast
	send_to
	receive_from
	iam_task0
	broadcast_send
	broadcast_recv
	shell_execute
	sleep_seconds
	sum_across_tasks
	get_dart_mpi_comm
	exit_all

call initialize_mpi_utilities([progname] [, alternatename])

```
character(len=*), intent(in), optional :: progname
character(len=*), intent(in), optional :: alternatename
```

Initializes the MPI library, creates a private communicator, stores the total number of tasks and the local task number for later use, and registers this module. This routine calls `initialize_utilities()` internally before returning, so the calling program need only call this one routine to initialize the DART internals.

On some implementations of MPI (in particular some variants of MPICH) it is best to initialize MPI before any I/O is done from any of the parallel tasks, so this routine should be called as close to the process startup as possible.

It is not an error to try to initialize the MPI library more than once. It is still necessary to call this routine even if the application itself has already initialized the MPI library. Thise routine creates a private communicator so internal communications are shielded from any other communication called outside the DART libraries.

It is an error to call any of the other routines in this file before calling this routine.

`progname`	If given, written to the log file to document which program is being started.
`alternatename`	If given, use this name as the log file instead of the default `dart_log.out`.

call finalize_mpi_utilities([callfinalize] [, async])

```
logical, intent(in), optional  :: callfinalize
integer, intent(in), optional  :: async
```

Frees the local communicator, and shuts down the MPI library unless `callfinalize` is specified and is `.FALSE.`. On some hardware platforms it is problematic to try to call print or write from the parallel tasks after finalize has been executed, so this should only be called immediately before the process is ready to exit. This routine does an `MPI_Barrier()` call before calling `MPI_Finalize()` to ensure all tasks are finished writing.

If the application itself is using MPI the `callfinalize` argument can be used to defer closing the MPI library until the application does it itself. This routine does close the DART log file and releases the local communicator even if not calling MPI_Finalize, so no other DART routines which might generate output can be used after calling this routine.

It is an error to call any of the other routines in this file after calling this routine.

`callfinalize`	If false, do not call the `MPI_Finalize()` routine.
`async`	If the model advance mode (selected by the async namelist value in the filter_nml section) requires any synchronization or actions at shutdown, this is done. Currently async=4 requires an additional set of actions at shutdown time.

var = task_count()

```
integer         :: task_count
```

Returns the total number of MPI tasks this job was started with. Note that MPI task numbers start at 0, but this is a count. So a 4-task job will return 4 here, but the actual task numbers will be from 0 to 3.

var	Total number of MPI tasks in this job.

var = my_task_id()

```
integer         :: my_task_id
```

Returns the local MPI task number. This is one of the routines in which all tasks can make the same function call but each returns a different value. The return can be useful in creating unique filenames or otherwise distinguishing resources which are not shared amongst tasks. MPI task numbers start at 0, so valid task id numbers for a 4-task job will be 0 to 3.

var	My unique MPI task id number.

call task_sync()

Synchronize tasks. This call does not return until all tasks have called this routine. This ensures all tasks have reached the same place in the code before proceeding. All tasks must make this call or the program will hang.

call send_to(dest_id, srcarray [, time])

```
integer,                     intent(in) :: dest_id
real(r8), dimension(:),      intent(in) :: srcarray
type(time_type), optional, intent(in) :: time
```

Use the MPI library to send a copy of an array of data from one task to another task. The sending task makes this call; the receiving task must make a corresponding call to `receive_from()`.

If `time` is specified, it is also sent to the receiving task. The receiving call must match this sending call regarding this argument; if `time` is specified here it must also be specified in the receive; if not given here it cannot be given in the receive.

The current implementation uses `MPI_Ssend()` which does a synchronous send. That means this routine will not return until the receiving task has called the receive routine to accept the data. This may be subject to change; MPI has several other non-blocking options for send and receive.

dest_id	The MPI task id of the receiver.
srcarray	The data to be copied to the receiver.
time	If specified, send the time as well.

The send and receive subroutines must be used with care. These calls must be used in pairs; the sending task and the receiving task must make corresponding calls or the tasks will hang. Calling them with different array sizes will result in either a run-time error or a core dump. The optional time argument must either be given in both calls or in neither or one of the tasks will hang. (Executive summary: There are lots of ways to go wrong here.)

call receive_from(src_id, destarray [, time])

```
integer, intent(in)                      :: src_id
real(r8), dimension(:), intent(out)      :: destarray
type(time_type), intent(out), optional :: time
```

Use the MPI library to receive a copy of an array of data from another task. The receiving task makes this call; the sending task must make a corresponding call to `send_to()`. Unpaired calls to these routines will result in the tasks hanging.

If `time` is specified, it is also received from the sending task. The sending call must match this receiving call regarding this argument; if `time` is specified here it must also be specified in the send; if not given here it cannot be given in the send.

The current implementation uses `MPI_Recv()` which does a synchronous receive. That means this routine will not return until the data has arrived in this task. This may be subject to change; MPI has several other non-blocking options for send and receive.

`src_id`	The MPI task id of the sender.
`destarray`	The location where the data from the sender is to be placed.
`time`	If specified, receive the time as well.

See the notes section of `send_to()`.

call exit_all(exit_code)

```
integer, intent(in)    :: exit_code
```

A replacement for calling the Fortran intrinsic `exit`. This routine calls `MPI_Abort()` to kill all MPI tasks associated with this job. This ensures one task does not exit silently and leave the rest hanging. This is not the same as calling `finalize_mpi_utilities()` which waits for the other tasks to finish, flushes all messages, closes log files cleanly, etc. This call immediately and abruptly halts all tasks associated with this job.

Depending on the MPI implementation and job control system, the exit code may or may not be passed back to the calling job script.

`exit_code`	A numeric exit code.

This routine is now called from the standard error handler. To avoid circular references this is NOT a module routine. Programs which are compiled without the mpi code must now compile with the `null_mpi_utilities_mod.f90` file to satisfy the call to this routine in the error handler.

call array_broadcast(array, root)

```
real(r8), dimension(:), intent(inout) :: array
integer, intent(in)                   :: root
```

All tasks must make this call together, but the behavior in each task differs depending on whether it is the `root` or not. On the task which has a task id equal to `root` the contents of the array will be sent to all other tasks. On any task which has a task id *not* equal to `root` the array is the location where the data is to be received into. Thus `array` is intent(in) on root, and intent(out) on all other tasks.

When this routine returns, all tasks will have the contents of the root array in their own arrays.

`array`	Array containing data to send to all other tasks, or the location in which to receive data.
`root`	Task ID which will be the data source. All others are destinations.

This is another of the routines which must be called by all tasks. The MPI call used here is synchronous, so all tasks block here until everyone has called this routine.

var = iam_task0()

```
logical                         :: iam_task0
```

Returns `.TRUE.` if called from the task with MPI task id 0. Returns `.FALSE.` in all other tasks. It is frequently the case that some code should execute only on a single task. This allows one to easily write a block surrounded by `if (iam_task0()) then ...`.

`var`	Convenience function to easily test and execute code blocks on task 0 only.

call broadcast_send(from, array1 [, array2] [, array3] [, array4] [, array5] [, scalar1] [, scalar2] [, scalar3] [, scalar4] [, scalar5])

```
integer, intent(in)                   :: from
real(r8), dimension(:), intent(inout) :: array1
real(r8), dimension(:), intent(inout), optional :: array2
real(r8), dimension(:), intent(inout), optional :: array3
real(r8), dimension(:), intent(inout), optional :: array4
real(r8), dimension(:), intent(inout), optional :: array5
real(r8), intent(inout), optional :: scalar1
real(r8), intent(inout), optional :: scalar2
real(r8), intent(inout), optional :: scalar3
real(r8), intent(inout), optional :: scalar4
real(r8), intent(inout), optional :: scalar5
```

Cover routine for `array_broadcast()`. This call must be matched with the companion call `broadcast_recv()`. This routine should only be called on the task which is the root of the broadcast; it will be the data source. All other tasks must call `broadcast_recv()`. This routine sends up to 5 data arrays and 5 scalars in a single call. A common pattern in the DART filter code is sending 2 arrays, but other combinations exist.

This routine ensures that `from` is the same as the current task ID. The arguments to this call must be matched exactly in number and type with the companion call to `broadcast_recv()` or an error (or hang) will occur.

In reality the data here are `intent(in)` only but this routine will be calling `array_broadcast()` internally and so must be `intent(inout)` to match.

`from`	Current task ID; the root task for the data broadcast.
`array1`	First data array to be broadcast.
array2	If given, second data array to be broadcast.
array3	If given, third data array to be broadcast.
array4	If given, fourth data array to be broadcast.
array5	If given, fifth data array to be broadcast.
scalar1	If given, first data scalar to be broadcast.
scalar2	If given, second data scalar to be broadcast.
scalar3	If given, third data scalar to be broadcast.
scalar4	If given, fourth data scalar to be broadcast.
scalar5	If given, fifth data scalar to be broadcast.

This is another of the routines which must be called consistently; only one task makes this call and all other tasks call the companion `broadcast_recv` routine. The MPI call used here is synchronous, so all tasks block until everyone has called one of these two routines.

call broadcast_recv(from, array1 [, array2] [, array3] [, array4] [, array5] [, scalar1] [, scalar2] [, scalar3] [, scalar4] [, scalar5])

```
integer, intent(in)                    :: from
real(r8), dimension(:), intent(inout) :: array1
real(r8), dimension(:), intent(inout), optional :: array2
real(r8), dimension(:), intent(inout), optional :: array3
real(r8), dimension(:), intent(inout), optional :: array4
real(r8), dimension(:), intent(inout), optional :: array5
real(r8), intent(inout), optional :: scalar1
real(r8), intent(inout), optional :: scalar2
real(r8), intent(inout), optional :: scalar3
real(r8), intent(inout), optional :: scalar4
real(r8), intent(inout), optional :: scalar5
```

Cover routine for `array_broadcast()`. This call must be matched with the companion call `broadcast_send()`. This routine must be called on all tasks which are *not* the root of the broadcast; the arguments specify the location in which to receive data from the root. (The root task should call `broadcast_send()`.) This routine receives up to 5 data arrays and 5 scalars in a single call. A common pattern in the DART filter code is receiving 2 arrays, but other combinations exist. This routine ensures that `from` is *not* the same as the current task ID. The arguments to this call must be matched exactly in number and type with the companion call to `broadcast_send()` or an error (or hang) will occur.

In reality the data arrays here are `intent(out)` only but this routine will be calling `array_broadcast()` internally and so must be `intent(inout)` to match.

`from`	The task ID for the data broadcast source.
`array1`	First array location to receive data into.
array2	If given, second data array to receive data into.
array3	If given, third data array to receive data into.
array4	If given, fourth data array to receive data into.
array5	If given, fifth data array to receive data into.
scalar1	If given, first data scalar to receive data into.
scalar2	If given, second data scalar to receive data into.
scalar3	If given, third data scalar to receive data into.
scalar4	If given, fourth data scalar to receive data into.
scalar5	If given, fifth data scalar to receive data into.

This is another of the routines which must be called consistently; all tasks but one make this call and exactly one other task calls the companion `broadcast_send` routine. The MPI call used here is synchronous, so all tasks block until everyone has called one of these two routines.

call sum_across_tasks(addend, sum)

```
integer, intent(in)                    :: addend
integer, intent(out)                   :: sum
```

All tasks call this routine, each with their own different `addend`. The returned value in `sum` is the total of the values summed across all tasks, and is the same for each task.

`addend`	Single input value per task to be summed up.
`sum`	The sum.

This is another of those calls which must be made from each task, and the calls block until this is so.

call block_task()

Create a named pipe (fifo) and read from it to block the process in such a way that it consumes no CPU time. Beware that once you put yourself to sleep you cannot wake yourself up. Some other MPI program must call restart_task() on the same set of processors the original program was distributed over.

Even though fifos appear to be files, in reality they are implemented in the kernel. The write into the fifo must be executed on the same node as the read is pending on. See the man pages for the mkfifo(1) command for more details.

call restart_task()

Write into the pipe to restart the reading task. Note that this must be an entirely separate executable from the one which called block_task(), because it is asleep like Sleeping Beauty and cannot wake itself. See filter and wakeup_filter for examples of a program pair which uses these calls in async=4 mode.

Even though fifos appear to be files, in reality they are implemented in the kernel. The write into the fifo must be executed on the same node as the read is pending on. See the man pages for the mkfifo(1) command for more details.

call finished_task(async)

```
integer, intent(in) :: async
```

For async=4 and task id = 0, write into the main filter-to-script fifo to tell the run script that filter is exiting. Does nothing else otherwise.

Even though fifos appear to be files, in reality they are implemented in the kernel. The write into the fifo must be executed on the same node as the read is pending on. See the man pages for the mkfifo(1) command for more details.

rc = shell_execute()

```
integer                             :: shell_execute
character(len=*), intent(in)   :: execute_string
logical, intent(in), optional :: serialize
```

Wrapper routine around the system() library function to execute shell level commands from inside the Fortran program. Will wait for the command to execute and will return the error code. 0 means ok, any other number indicates error.

`rc`	Return code from the shell exit after the command has been executed.
`execute_string`	Command to be executed by the shell.
`serialize`	If specified and if .TRUE. run the command from each PE in turn, waiting for each to complete before beginning the next. The default is .FALSE. and does not require that all tasks call this routine. If given and .TRUE. then all tasks must make this call.

call sleep_seconds(naplength)

```
real(r8), intent(in) :: naplength
```

Wrapper routine for the sleep command. Argument is a real in seconds. Some systems have different lower resolutions for the minimum time it will sleep. This routine can round up to even seconds if a smaller than 1.0 time is given.

`naplength`	Number of seconds to sleep as a real value.

The amount of time this routine will sleep is not precise and might be in units of whole seconds on some platforms.

comm = get_dart_mpi_comm()

```
integer    :: get_dart_mpi_comm
```

This code creates a private communicator for DART MPI calls, in case other code in the executable is using the world communicator. This routine returns the private communicator. If it is called before the internal setup work is completed it returns MPI_COMM_WORLD. If it is called before MPI is initialized, it returns 0.

`comm`	The private DART communicator.

6.124.5 Files

- mpi module or
- mpif.h

Depending on the implementation of MPI, the library routines are either defined in an include file (`mpif.h`) or in a proper Fortran 90 module (`use mpi`). If it is available the module is preferred; it allows for better argument checking and optional arguments support in the MPI library calls.

6.124.6 References

- MPI: The Complete Reference; Snir, Otto, Huss-Lederman, Walker, Dongarra; MIT Press, 1996, ISBN 0-262-69184-1
- `` `http://www-unix.mcs.anl.gov/mpi/ `` <http://www-unix.mcs.anl.gov/mpi/>`__

6.124.7 Private components

N/A

6.125 MODULE time_manager_mod

6.125.1 Overview

Provides a set of routines to manipulate both time and calendars of various types.

Time intervals are stored and defined in terms of integer number of days and integer seconds. The minimum time resolution is 1 second. Mathematical operations (e.g. addition, subtraction, multiplication) are defined on these intervals. Seconds which roll over 86400 (the number of seconds in a day) are converted into days.

Calendars interpret time intervals in terms of years, months, days. Various calendars commonly in use in the scientific community are supported.

6.125.2 Other modules used

```
types_mod
utilities_mod
```

6.125.3 Public interfaces

use time_manager_mod, only :	time_type
	operator(+)
	operator(-)
	operator(*)
	operator(/)
	operator(>)
	operator(>=)
	operator(==)
	operator(/=)
	operator(<)
	operator(<=)
	operator(//)
	set_time
	set_time_missing
	increment_time
	decrement_time
	get_time
	interval_alarm
	repeat_alarm
	THIRTY_DAY_MONTHS
	JULIAN
	GREGORIAN
	NOLEAP
	NO_CALENDAR
	GREGORIAN_MARS
	set_calendar_type
	get_calendar_type
	get_calendar_string
	set_date
	get_date
	increment_date
	decrement_date
	days_in_month
	leap_year
	length_of_year
	days_in_year
	month_name
	julian_day
	time_manager_init
	print_time
	print_date
	write_time

continues on next page

Table 6 – continued from previous page

	read_time
	interactive_time

var = set_time(seconds [, days])

```
type(time_type)                   :: set_time
integer,            intent(in) :: seconds
integer, optional, intent(in) :: days
```

Fills a time type. If seconds are > 86400, they are converted into the appropriate number of days. Note that seconds are specified first.

`seconds`	Number of seconds. If larger than 86400, they are converted into the appropriate number of days.
days	Number of days. Default is 0.

var = set_time_missing()

```
type(time_type)                          :: set_time_missing
```

Set a time type to a missing value. The resulting time value will cause an error if used for an arithmetic operation or if get_time() is called.

var = increment_time(time, seconds [, days])

```
type(time_type)                   :: increment_time
type(time_type),    intent(in) :: time
integer,            intent(in) :: seconds
integer, optional, intent(in) :: days
```

Adds the specified number of seconds and optionally, days, to the given time and returns the new time. Increments cannot be negative (see decrement_time below).

`time`	time value to be incremented.
`seconds`	number of seconds to add to given time.
days	optionally a number of days to add to the given time.

var = decrement_time(time, seconds [, days])

```
type(time_type)                          :: decrement_time
type(time_type), intent(in)              :: time
integer,         intent(in)              :: seconds
integer,         intent(in), optional    :: days
```

Subtract the specified number of seconds and optionally, days, to the given time and returns the new time. Decrements cannot be negative (see increment_time above).

`time`	time value to be decremented.
`seconds`	number of seconds to subtract from the given time.
days	optionally a number of days to subtract from the given time.

var = interval_alarm(time, time_interval, alarm, alarm_interval)

```
logical                             :: interval_alarm
type(time_type), intent(in)         :: time
type(time_type), intent(in)         :: time_interval
type(time_type), intent(inout)      :: alarm
type(time_type), intent(in)         :: alarm_interval
```

Supports a commonly used type of test on times for models. Given the current time, and a time for an alarm, determines if this is the closest time to the alarm time given a time step of time_interval. If this is the closest time (alarm - time <= time_interval/2), the function returns true and the alarm is incremented by the alarm_interval. Watch for problems if the new alarm time is less than time + time_interval.

`time`	Current time.
`time_interval`	Bin size for determining if alarm time is close enough to now.
`alarm`	When alarm next goes off next. Updated by this routine.
`alarm_interval`	How often alarm goes off.

var = repeat_alarm(time, alarm_frequency, alarm_length)

```
type(time_type)                     :: repeat_alarm
type(time_type), intent(in)         :: time
type(time_type), intent(in)         :: alarm_frequency
type(time_type), intent(in)         :: alarm_length
```

Repeat_alarm supports an alarm that goes off with alarm_frequency and lasts for alarm_length. If the nearest occurence of an alarm time is less than half an alarm_length from the input time, repeat_alarm is true. For instance, if the alarm_frequency is 1 day, and the alarm_length is 2 hours, then repeat_alarm is true from time 2300 on day n to time 0100 on day n + 1 for all n.

`time`	Current time.
`alarm_frequency`	How often the alarm goes off.
`alarm_length`	How long the alarm is true.

var = get_calendar_type()

```
integer :: get_calendar_type
```

Returns default calendar type for mapping from time to date. Calendar types are public integer parameters that define various calendars. See elsewhere in this file for the list.

var = set_date(year, month, day [, hours, minutes, seconds])

```
type(time_type)                    :: set_date
integer, intent(in)                :: year
integer, intent(in)                :: month
integer, intent(in)                :: day
integer, intent(in), optional      :: hours
integer, intent(in), optional      :: minutes
integer, intent(in), optional      :: seconds
```

Given a date interpreted using the current calendar type, compute the corresponding time.

`year`	Integer year.
`month`	Integer month number.
`day`	Integer day number.
hours	Integer hour. Default is 0.
minutes	Integer minutes. Default is 0.
seconds	Integer seconds. Default is 0.

var = increment_date(time [, years, months, days, hours, minutes, seconds])

```
type(time_type)                    :: increment_date
type(time_type), intent(in)        :: time
integer, intent(in), optional      :: years
integer, intent(in), optional      :: months
integer, intent(in), optional      :: days
integer, intent(in), optional      :: hours
integer, intent(in), optional      :: minutes
integer, intent(in), optional      :: seconds
```

Given a time and some date increment, compute a new time. The interpretation of the date depends on the currently selected calendar type.

`time`	Current time.
year	Integer years to add. Default is 0.
month	Integer months to add. Default is 0.
day	Integer days to add. Default is 0.
hours	Integer hours to add. Default is 0.
minutes	Integer minutes to add. Default is 0.
seconds	Integer seconds to add. Default is 0.

var = decrement_date(time [, years, months, days, hours, minutes, seconds])

```
type(time_type)                    :: decrement_date
type(time_type), intent(in)        :: time
integer, intent(in), optional      :: years
integer, intent(in), optional      :: months
integer, intent(in), optional      :: days
integer, intent(in), optional      :: hours
integer, intent(in), optional      :: minutes
integer, intent(in), optional      :: seconds
```

Given a time and some date decrement, compute a new time. The interpretation of the date depends on the currently selected calendar type.

`time`	Current time.
year	Integer years to subtract. Default is 0.
month	Integer months to subtract. Default is 0.
day	Integer days to subtract. Default is 0.
hours	Integer hours to subtract. Default is 0.
minutes	Integer minutes to subtract. Default is 0.
seconds	Integer seconds to subtract. Default is 0.

var = days_in_month(time)

```
integer                            :: days_in_month
type(time_type), intent(in)        :: time
```

Given a time, determine the month based on the currently selected calendar type and return the numbers of days in that month.

`time`	Current time.

var = leap_year(time)

```
logical                          :: leap_year
type(time_type),intent(in)       :: time
```

Given a time, determine if the current year is a leap year in the currently selected calendar type.

time	Current time.

var = length_of_year()

```
integer                        :: length_of_year
```

For the currently selected calendar type, return the number of days in a year if that value is fixed (e.g. there are not leap years). For other calendar types, see days_in_year() which takes a time argument to determine the current year.

var = days_in_year(time)

```
integer                          :: days_in_year
type(time_type), intent(in)      :: time
```

Given a time, determine the year based on the currently selected calendar type and return the numbers of days in that year.

time	Current time.

var = month_name(n)

```
character(len=9)                 :: month_name
integer,          intent(in)     :: n
```

Return a character string containing the month name corresponding to the given month number.

n	Month number. Must be between 1 and 12, inclusive.

var = julian_day(year, month, day)

```
integer                         :: julian_day
integer,        intent(in)      :: year
integer,        intent(in)      :: month
integer,        intent(in)      :: day
```

Given a date in year/month/day format, compute the day number from the beginning of the year. The currently selected calendar type must be GREGORIAN.

`year`	Year number in the Gregorian calendar.
`month`	Month number in the Gregorian calendar.
`day`	Day of month in the Gregorian calendar.

var = read_time(file_unit [, form, ios_out])

```
type(time_type)                               :: read_time
integer,           intent(in)                 :: file_unit
character(len=*), intent(in),  optional :: form
integer,           intent(out), optional :: ios_out
```

Read a time from the given file unit number. The unit must already be open. The default format is ascii/formatted. If an error is encountered and ios_out is specified, the error status will be returned to the caller; otherwise the error is fatal.

`file_unit`	Integer file unit number of an already open file.
`form`	Format to read the time. Options are 'formatted' or 'unformatted'. Default is 'formatted'.
`ios_out`	On error, if specified, the error status code is returned here. If not specified, an error calls the standard error_handler and exits.

call get_time(time, seconds [, days])

```
type(time_type), intent(in)              :: time
integer,         intent(out)             :: seconds
integer,         intent(out), optional   :: days
```

Returns days and seconds (< 86400) corresponding to a time. If the optional 'days' argument is not given, the days are converted to seconds and the total time is returned as seconds. Note that seconds preceeds days in the argument list.

`time`	Time to convert into seconds and days.
`seconds`	If days is specified, number of seconds in the current day. Otherwise, total number of seconds in time.
`days`	If specified, number of days in time.

call set_calendar_type(mytype) or *call set_calendar_type(calstring)*

```
integer, intent(in)              :: mytype
 or
character(len=*), intent(in)     :: calstring
```

Selects the current calendar type, for converting between time and year/month/day. The argument can either be one of the predefined calendar integer parameter types (see elsewhere in this file for the list of types), or a string which matches the name of the integer parameters. The string interface is especially suitable for namelist use.

mytype	Integer parameter to select the calendar type.

or

calstring	Character string to select the calendar type. Valid strings match the names of the integer parameters.

call get_calendar_string(mystring)

```
character(len=*), intent(out)     :: mystring
```

Return the character string corresponding to the currently selected calendar type.

mystring	Character string corresponding to the current calendar type.

call get_date(time, year, month, day, hour, minute, second)

```
type(time_type), intent(in)       :: time
integer, intent(out)              :: year
integer, intent(out)              :: month
integer, intent(out)              :: day
integer, intent(out)              :: hour
integer, intent(out)              :: minute
integer, intent(out)              :: second
```

Given a time, compute the corresponding date given the currently selected calendar type.

time	Input time.
year	Corresponding calendar year.
month	Corresponding calendar month.
day	Corresponding calendar day.
hour	Corresponding hour.
minute	Corresponding minute.
second	Corresponding second.

call time_manager_init()

Initializes any internal data needed by the time manager code. Does not need to be called before using any of the time manager routines; it will be called internally before executing any of the other routines.

call print_time(time [, str, iunit])

```
type(time_type),  intent(in)           :: time
character(len=*), intent(in), optional :: str
integer,          intent(in), optional :: iunit
```

Print the time as days and seconds. If the optional str argument is specified, print that string as a label. If iunit is specified, write output to that unit; otherwise write to standard output/terminal.

`time`	Time to be printed as days/seconds.
str	String label to print before days/seconds. Default: 'TIME: '.
iunit	Unit number to write output on. Default is standard output/terminal (unit 6).

call print_date(time [, str, iunit])

```
type(time_type),  intent(in)           :: time
character(len=*), intent(in), optional :: str
integer,          intent(in), optional :: iunit
```

Print the time as year/month/day/hour/minute/second, as computed from the currently selected calendar type. If the optional str argument is specified, print that string as a label. If iunit is specified, write output to that unit; otherwise write to standard output/terminal.

`time`	Time to be printed as a calendar date/time.
str	String label to print before date. Default: 'DATE: '.
iunit	Unit number to write output on. Default is standard output/terminal (unit 6).

call write_time(file_unit, time [, form, ios_out])

```
integer,          intent(in)               :: file_unit
type(time_type),  intent(in)               :: time
character(len=*), intent(in),  optional    :: form
integer,          intent(out), optional    :: ios_out
```

Write a time to an already open file unit. The optional 'form' argument controls whether it is formatted or unformatted. On error, the optional 'ios_out' argument returns the error code; otherwise a fatal error is triggered.

`file_unit`	Integer unit number for an already open file.
`time`	Time to write to the file.
form	String format specifier; either 'unformatted' or 'formatted'. Defaults to 'formatted'.
ios_out	If specified, on error the i/o status error code is returned here. Otherwise, the standard error handler is called and the program exits.

call interactive_time(time)

```
type(time_type), intent(inout) :: time
```

Prompt the user for a time as a calendar date, based on the currently selected calendar type. Writes prompt to standard output and reads from standard input.

`time`	Time type to be returned.

```
type time_type
   private
   integer :: seconds
   integer :: days
end type time_type
```

This type is used to define a time interval.

```
integer :: NO_CALENDAR
integer :: GREGORIAN
integer :: GREGORIAN_MARS
integer :: JULIAN
integer :: THIRTY_DAY_MONTHS
integer :: NOLEAP
```

The public integer parameters which define different calendar types. The same names defined as strings can be used to set the calendar type.

```
operator(+)
operator(-)
operator(*)
operator(/)
operator(>)
operator(>=)
operator(==)
operator(/=)
operator(<)
operator(<=)
operator(//)
```

Arithmetic operations are defined for time types, so expressions like

```
t3 = t1 + t2
```

can be constructed. To use these operators, they must be listed on the module use statement in the form specified above.

Multiplication is one time and one scalar.

Division with a single slash is integer, and returns the largest integer for which time1 >= time2 * n. Division with a double slash returns a double precision quotient of the two times.

6.125.4 Namelist

No namelist is currently defined for the time manager code.

6.125.5 Files

- none

6.125.6 References

1. none

6.125.7 Private components

N/A

6.126 MODULE utilities_mod

6.126.1 Overview

Provides a number of tools used by most DART modules including tools for file IO, diagnostic tools for registering modules and recording namelist arguments, and an error handler.

6.126.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand ‘&’ and terminate with a slash ‘/’. Character strings that contain a ‘/’ must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&utilities_nml
   TERMLEVEL      = 2,
   logfilename    = 'dart_log.out',
   nmlfilename    = 'dart_log.nml',
   module_details = .true.,
   print_debug    = .false.,
   write_nml      = 'file'
/
```

The namelist controls how the logging, namelist, messages, and general utility routines behave.

Item	Type	Description
TERMLEVEL	integer	Level at which calls to error manager terminate program. The default setting is warnings and errors terminate the program. Setting this to 2 (E_ERR) means only errors terminate. Setting this to 3 means even errors do not cause an exit (which is not a good idea).
logfilename	character(len=129)	File to which the log messages are written.
nmlfilename	character(len=129)	File to which the namelist output is written. Can be the same name as the logfile.
module_details	logical	Each source code module can write out the repository version number and filename to the logfile. Verbose, but useful for knowing what version of the code was used during the run.
print_debug	logical	Setting this to .true. causes additional debug messages to print. These can be very verbose and by default are turned off.
write_nml	character(len=32)	String which controls where to write the namelist values that are being used for this execution. Valid values are: ‘none’, ‘file’, ‘terminal’, ‘both’. ‘none’ turns off this write. ‘file’ writes a copy only to the `nmlfilename`. Writes are always in append mode, so the most recent information will be at the end of an existing file. ‘terminal’ will write to the job’s standard output. ‘both’ will write both to the nml file and the standard output unit.

6.126.3 Other modules used

```
types_mod
netCDF
```

6.126.4 Public interfaces

use utilities, only :	file_exist
	get_unit
	open_file
	close_file
	timestamp
	register_module
	error_handler
	to_upper
	nc_check
	logfileunit
	nmlfileunit
	initialize_utilities
	finalize_utilities
	dump_unit_attributes
	find_namelist_in_file
	check_namelist_read
	find_textfile_dims
	file_to_text
	is_longitude_between
	get_next_filename
	set_filename_list
	set_tasknum
	set_output
	do_output
	E_DBG, DEBUG
	E_MSG, MESSAGE
	E_WARN, WARNING
	E_ERR, FATAL

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

var = file_exist(file_name)

```
logical                          :: file_exist
character(len=*), intent(in)  :: file_name
```

Returns true if file_name exists in the working directory, else false.

`var`	True if file_name exists in working directory.
`file_name`	Name of file to look for.

var = get_unit()

```
integer :: get_unit
```

Returns an unused unit number for IO.

`var`	An unused unit number.

var = open_file(fname [, form, action])

```
integer                                     :: open_file
character(len=*), intent(in)                :: fname
character(len=*), optional, intent(in)  :: form
character(len=*), optional, intent(in)  :: action
```

Returns a unit number that is opened to the file fname. If form is not present or if form is "formatted" or "FORMATTED", file is opened for formatted IO. Otherwise, it is unformatted. The action string is the standard action string for Fortran IO (see F90 language description).

`var`	Unit number opened to file fname.
`fname`	Name of file to be opened.
form	Format: 'formatted' or 'FORMATTED' give formatted, anything else is unformatted. Default is formatted.
action	Standard fortran string description of requested file open action.

call timestamp([string1, string2, string3,] pos)

```
character(len=*), optional, intent(in) :: string1
character(len=*), optional, intent(in) :: string2
character(len=*), optional, intent(in) :: string3
character(len=*), intent(in)           :: pos
```

Prints the message 'Time is YYYY MM DD HH MM SS' to the logfile along with three optional message strings. If the pos argument is 'end', the message printed is 'Finished... at YYYY MM DD HH MM SS' and the logfile is closed.

string1	An optional message to be printed.
string2	An optional message to be printed.
string3	An optional message to be printed.
`pos`	If 'end' terminates log_file output.

call close_file(iunit)

```
integer, intent(in) :: iunit
```

Closes the given unit number. If the unit is not open, nothing happens.

`iunit`	File unit to be closed.

call register_module(src, rev, rdate)

```
character(len=*), intent(in) :: src
character(len=*), optional, intent(in) :: rev
character(len=*), optional, intent(in) :: rdate
```

Writes the source name to both the logfileunit and to standard out. The rev and revdate are deprecated as they are unsupported by git.

`src`	source file name.
`rev`	ignored
`rdate`	ignored

call error_handler(level, routine, text, src, rev, rdate [, aut, text2, text3])

```
integer, intent(in)                        :: level
character(len=*), intent(in)               :: routine
character(len=*), intent(in)               :: text
character(len=*), intent(in)               :: src
character(len=*), intent(in)               :: rev
character(len=*), intent(in)               :: rdate
character(len=*), optional, intent(in)     :: aut
character(len=*), optional, intent(in)     :: text2
character(len=*), optional, intent(in)     :: text3
```

Prints an error message to standard out and to the logfileunit. The message contains the routine name, an error message, the source file, revision and revision date, and optionally the author. The level of severity is message, debug, warning, or error. If the level is greater than or equal to the TERMLEVEL (set in the namelist), execution is terminated. The default TERMLEVEL only stops for ERRORS.

`level`	Error severity (message, debug, warning, error). See below for specific ations.
`routine`	Name of routine generating error.
`text`	Error message.
`src`	Source file containing routine generating message.
`rev`	Revision number of source file.
`rdate`	Revision date of source file.
aut	Author of routine.
text2	If specified, the second line of text for the error message.
text3	If specified, the third line of text for the error message.

call find_namelist_in_file(namelist_file_name, nml_name, iunit, [,write_to_logfile_in])

```
character(len=*),  intent(in)          :: namelist_file_name
character(len=*),  intent(in)          :: nml_name
integer,           intent(out)         :: iunit
logical, optional, intent(in)          :: write_to_logfile_in
```

Opens the file namelist_file_name if it exists on unit iunit. A fatal error occurs if the file does not exist (DART requires an input.nml to be available, even if it contains no values). Searches through the file for a line containing ONLY the string &nml_name (for instance &filter_nml if nml_name is "filter_nml"). If this line is found, the file is rewound and the routine returns. Otherwise, a fatal error message is issued.

`namelis`	Name of file assumed to hold the namelist.
`nml_nam`	Name of the namelist to be searched for in the file, for instance, filter_nml.
`iunit`	Channel number on which file is opened.
write_to_logfile_in	While the namelist for the utilities module is read, the logfile has not yet been open because its name is in the namelist. If errors are found, have to write to standard out. So, when utilities module calls this internally, this optional argument is set to false. For all other applications, it is normally not used (default is false).

call check_namelist_read(iunit, iostat_in, nml_name, [, write_to_logfile_in])

```
integer, intent(in)                     :: iunit
integer, intent(in)                     :: iostat_in
character(len=*), intent(in)            :: nml_name
logical, optional, intent(in)           :: write_to_logfile_in
```

Once a namelist has been read from an opened namelist file, this routine checks for possible errors in the read. If the namelist read was successful, the file opened on iunit is closed and the routine returns. If iostat is not zero, an attempt is made to rewind the file on iunit and read the last line that was successfully read. If this can be done, this last line is printed with the preamble "INVALID NAMELIST ENTRY". If the attempt to read the line after rewinding fails, it is assumed that the original read (before the call to this subroutine) failed by reaching the end of the file. An error message stating that the namelist started but was never terminated is issued.

iunit	Channel number on which file is opened.
iostat_in	Error status return from an attempted read of a namelist from this file.
nml_name	The name of the namelist that is being read (for instance filter_nml).
write_to_logfile_in	While the namelist for the utilities module is read, the logfile has not yet been open because its name is in the namelist. If errors are found, have to write to standard out. So, when utilities module calls this internally, this optional argument is set to false. For all other applications, it is normally not used (default is false).

call find_textfile_dims (fname, nlines, linelen)

```
character(len=*), intent (IN)  :: fname
integer,          intent (OUT) :: nlines
integer,          intent (OUT) :: linelen
```

Determines the number of lines and maximum line length of an ASCII text file.

fname	input, character string file name
nlines	output, number of lines in the file
linelen	output, length of longest line in the file

call file_to_text (fname, textblock)

```
character(len=*),                intent (IN)  :: fname
character(len=*), dimension(:), intent (OUT) :: textblock
```

Opens the given filename and reads ASCII text lines into a character array.

fname	input, character string file name
textblock	output, character array of text in the file

var = is_longitude_between(lon, minlon, maxlon [, doradians])

```
real(r8), intent(in)           :: lon
real(r8), intent(in)           :: minlon
real(r8), intent(in)           :: maxlon
logical,  intent(in), optional :: doradians
logical                        :: is_longitude_between
```

Uniform way to test longitude ranges, in degrees, on a globe. Returns true if lon is between min and max, starting at min and going EAST until reaching max. Wraps across 0 longitude. If min equals max, all points are inside. Includes endpoints. If optional arg doradians is true, do computation in radians between 0 and 2*PI instead of default 360. There is no rejection of input values based on range; they are all converted to a known range by calling modulo() first.

`var`	True if lon is between min and max.
`lon`	Location to test.
`minlon`	Minimum longitude. Region will start here and go east.
`maxlon`	Maximum longitude. Region will end here.
doradians	Optional argument. Default computations are in degrees. If this argument is specified and is .true., do the computation in radians, and wrap across the globe at 2 * PI. All inputs must then be specified in radians.

var = get_next_filename(listname, lineindex)

```
character(len=*),  intent(in) :: listname
integer,           intent(in) :: lineindex
character(len=128)            :: get_next_filename
```

Returns the specified line of a text file, given a filename and a line number. It returns an empty string when the line number is larger than the number of lines in a file.

Intended as an easy way to process a list of files. Use a command like 'ls > out' to create a file containing the list, in order, of files to be processed. Then call this function with an increasing index number until the return value is empty.

`var`	An ascii string, up to 128 characters long, containing the contents of line `lineindex` of the input file.
`listname`	The filename to open and read lines from.
`lineindex`	Integer line number, starting at 1. If larger than the number of lines in the file, the empty string '' will be returned.

var = set_filename_list(name_array, listname, caller_name)

```
character(len=*),  intent(inout) :: name_array
character(len=*),  intent(in)    :: listname
character(len=*),  intent(in)    :: caller_name
integer                          :: var
```

Returns the count of filenames specified. Verifies that one of either the name_array or the listname was specified but not both. If the input was a listname copy the names into the name_array so when this routine returns all the filenames are in name_array(). Verifies that no more than the allowed number of names was specified if the input was a listname file.

`var`	The count of input files specified.
`name_array`	Array of input filename strings. Either this item or the listname must be specified, but not both.
`listname`	The filename to open and read filenames from, one per line. Either this item or the name_array must be specified but not both.
`caller_name`	Calling subroutine name, used for error messages.

call to_upper(string)

```
character(len=*), intent (INOUT) :: string
```

Converts the character string to UPPERCASE - in place. The input string **is** modified.

`string`	any character string

call nc_check(istatus, subr_name [, context])

```
integer, intent(in)                       :: istatus
character(len=*), intent(in)              :: subr_name
character(len=*), optional, intent(in)    :: context
```

Check the return code from a netcdf call. If no error, return without taking any action. If an error is indicated (in the `istatus` argument) then call the error handler with the subroutine name and any additional context information (e.g. which file or which variable was being processed at the time of the error). All errors are currently hardcoded to be `FATAL` and this routine will not return.

This routine calls a netCDF library routine to construct the text error message corresponding to the error code in the first argument. An example use of this routine is:

```
call nc_check(nf90_create(path = trim(ncFileID%fname), cmode = nf90_share, ncid =␣
↪ncFileID%ncid), &
             'init_diag_output', 'create '//trim(ncFileID%fname))
```

`istatus`	The return value from any netCDF call.
`subr_name`	String name of the current subroutine, used in case of error.
context	Additional text to be used in the error message, for example to indicate which file or which variable is being processed.

call set_tasknum(tasknum)

```
integer, intent(in)              :: tasknum
```

Intended to be used in the MPI multi-task case. Sets the local task number, which is then prepended to subsequent messages.

tasknum	Task number returned from MPI_Comm_Rank(). MPI task numbers are 0 based, so for a 4-task job these numbers are 0-3.

call set_output(doflag)

```
logical, intent(in)              :: doflag
```

Set the status of output. Can be set on a per-task basis if you are running with multiple tasks. If set to false only warnings and fatal errors will write to the log. The default in the multi-task case is controlled by the MPI module initialization code, which sets task 0 to .TRUE. and all other tasks to .FALSE.

doflag	Sets, on a per-task basis, whether messages are to be written to the logfile or standard output. Warnings and errors are always output.

var = do_output()

```
logical                          :: do_output
```

Returns true if this task should write to the log, false otherwise. Set by the `set_output()` routine. Defaults to true for the single task case. Can be used in code like so:

```
if (do_output()) then
 write(*,*) 'At this point in the code'
endif
```

var	True if this task should write output.

call initialize_utilities([progname] [, alternatename])

```
character(len=*), intent(in), optional :: progname
character(len=*), intent(in), optional :: alternatename
```

Reads the namelist and opens the logfile. Records the values of the namelist and registers this module.

progname	If given, use in the timestamp message in the log file to say which program is being started.
alternatename	If given, log filename to use instead of the value in the namelist. This permits, for example, different programs sharing the same input.nml file to have different logs. If not given here and no value is specified in the namelist, this defaults to dart_log.out

call finalize_utilities()

Closes the logfile; using utilities after this call is a bad idea.

call dump_unit_attributes(iunit)

```
integer, intent(in) :: iunit
```

Writes all information about the status of the IO unit to the error handler with error level message.

`iunit`	Unit about which information is requested.

```
integer :: E_DBG, DEBUG
integer :: E_MSG, MESSAGE
integer :: E_WARN, WARNING
integer :: E_ERR, FATAL
```

	Severity levels to be passed to error handler. Levels are debug, message, warning and fatal. The namelist parameter TERMLEVEL can be used to control at which level program termination should occur.

```
integer :: logfileunit
```

`logfileunit`	Unit opened to file for diagnostic output.

```
integer :: nmlfileunit
```

`nmlfileunit`	Unit opened to file for diagnostic output of namelist files. Defaults to same as `logfileunit`. Provides the flexibility to log namelists to a separate file, reducing the clutter in the log files and perhaps increasing readability.

6.126.5 Files

- assim_model_mod.nml in input.nml
- logfile, name specified in namelist

6.126.6 References

- none

6.126.7 Private components

N/A

6.127 MODULE types_mod

6.127.1 Overview

Provides some commonly used mathematical constants, and a set of Fortran integer and real kinds, to be used to select the right variable size (e.g. 4 bytes, 8 bytes) to match the rest of the DART interfaces. (DART does not depend on compiler flags to set precision, but explicitly specifies a kind for each variable in the public interfaces.)

6.127.2 Other modules used

```
none
```

6.127.3 Public interfaces

This routine provides the following constants, but no routines of any kind.

The constants defined here *may* or *may not* be declared the same as constants used in non-DART pieces of code. It would seem like a good idea to match the DART definition of 'gas_constant' to the WRF equivalent if you are going to be running WRF/DART experiments (for example).

use types_mod, only :	i4
	i8
	r4
	r8
	c4
	c8
	digits12
	PI
	DEG2RAD
	RAD2DEG
	SECPERDAY
	MISSING_R4
	MISSING_R8
	MISSING_I
	MISSING_DATA
	metadatalength
	obstypelength
	t_kelvin
	es_alpha
	es_beta
	es_gamma
	gas_constant_v
	gas_constant
	L_over_Rv
	ps0
	earth_radius
	gravity

```
integer, parameter :: i4
integer, parameter :: i8
integer, parameter :: r4
integer, parameter :: r8
integer, parameter :: c4
integer, parameter :: c8
integer, parameter :: digits12
```

These kinds are used when declaring variables, like:

```
real(r8)    :: myvariable
integer(i4) :: shortint
```

All DART public interfaces use types on the real values to ensure they are consistent across various compilers and compile-time options. The digits12 is generally only used for reals which require extra precision.

Some models are able to run with single precision real values, which saves both memory when executing and file space when writing and reading restart files. To accomplish this, the users edit this file, redefine r8 to equal r4, and then rebuild all of DART.

```
real(KIND=R8), parameter :: PI
real(KIND=R8), parameter :: DEG2RAD
real(KIND=R8), parameter :: RAD2DEG
real(KIND=R8), parameter :: SECPERDAY
```

Some commonly used math constants, defined here for convenience.

```
real(KIND=R4), parameter :: MISSING_R4
real(KIND=R8), parameter :: MISSING_R8
integer,       parameter :: MISSING_I
integer,       parameter :: MISSING_DATA
```

Numeric constants used in the DART code when a numeric value is required, but the data is invalid or missing. These are typically defined as negative and a series of 8's, so they are distinctive when scanning a list of values.

```
integer, parameter :: metadatalength
integer, parameter :: obstypelength
```

Some common string limits used system-wide by DART code. The obstypelength is limited by the Fortran-imposed maximum number of characters in a parameter; the metadatalength was selected to be long enough to allow descriptive names but short enough to keep printing to less than a single line.

```
real(KIND=R8), parameter :: t_kevin
real(KIND=R8), parameter :: es_alpha
real(KIND=R8), parameter :: es_beta
real(KIND=R8), parameter :: es_gamma
real(KIND=R8), parameter :: gas_constant_v
real(KIND=R8), parameter :: gas_constant
real(KIND=R8), parameter :: L_over_Rv
real(KIND=R8), parameter :: ps0
real(KIND=R8), parameter :: earth_radius
real(KIND=R8), parameter :: gravity
```

A set of geophysical constants, which could be argued do not belong in a DART-supplied file since they are quite probably specific to a model or a particular forward operator.

Best case would be if we could engineer the code so these constants were provided by the model and then used when compiling the forward operator files. But given that Fortran use statements cannot be circular, this poses a problem. Perhaps we could work out how the obs_def code could define these constants and then they could be used by the model code. For now, they are defined here but it is up to the model and obs_def code writers whether to use these or not.

6.127.4 Namelist

There is no namelist for this module.

6.127.5 Files

None.

6.127.6 References

1. none

6.127.7 Private components

N/A

6.128 MODULE schedule_mod

6.128.1 Overview

Provides a set of routines to generate a regular pattern of time windows. This module is only used for converting observation sequences files to netCDF format. If it stands the test of time, it will likely be used to create an assimilation schedule independent of the observation sequence file. Wouldn't that be nice . . .

6.128.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&schedule_nml
   first_bin_start      =  1601,  1,  1,  0,  0,  0
   first_bin_end        =  2999,  1,  1,  0,  0,  0
   last_bin_end         =  2999,  1,  1,  0,  0,  0
   bin_interval_days    = 1000000
   bin_interval_seconds = 0
   max_num_bins         = 1000
   calendar             = 'Gregorian'
   print_table          = .true.
  /
```

Controls various aspects of filter. The inflation control variables are all dimensioned 2, the first value being for the prior inflation and the second being for the posterior inflation.

The default values will cause (pretty much) all possible observations to be put into one output file.

Item	Type	Description
first_bin_start	integer, dimension(6)	Date/time specification for starting time of first bin.
first_bin_end	integer, dimension(6)	Date/time specification for ending time of first bin. Sets the bin width.
last_bin_end	integer, dimension(6)	Date/time specification for ending time of last bin. Sets the length of the overall time of the schedule.
bin_interval_days	integer	Sets the time between bins. Must be larger or equal to the bin width.
bin_interval_seconds	integer	Sets the time between bins. Must be larger or equal to the bin width.
max_num_bins	integer	Upper limit on the number of bins.
calendar	character(len=32)	String calendar type. Valid types are listed in the time_manager_mod file.
print_table	logical	If .TRUE., print out information about the schedule each time set_regular_schedule() is called.

6.128.3 Other modules used

```
types_mod
utilities_mod
time_manager_mod
```

6.128.4 Public interfaces

use schedule_mod, only :	schedule_type
	set_regular_schedule
	get_time_from_schedule
	get_schedule_length

Namelist `&schedule_mod_nml` may be read from file `input.nml`.

call set_regular_schedule(schedule)

```
type(schedule_type), intent(out) :: schedule
```

Uses the namelist information to compute and fill a schedule_type variable.

`schedule`	Fills this derived type with the information needed to generate a series of regularly spaced time windows.

call get_time_from_schedule(mytime, schedule, iepoch [, edge])

```
type(time_type),      intent(out) :: mytime
 or
real(digits12),       intent(out) :: mytime
type(schedule_type), intent(in)  :: schedule
integer,              intent(in)  :: iepoch
integer, optional,    intent(in)  :: edge
```

Returns either the leading or trailing time for the specified bin/epoch number for the given schedule. The time can be returned in one of two formats, depending on the variable type specified for the first argument: either a DART derived time_type, or a real of kind digits12 (defined in the types_mod).

mytime	Return value with the leading or trailing edge time for the requested bin. There are two supported return formats, either as a standard DART time_type, or as a real value which will contain the number of days plus any fraction.
schedule	Schedule type to extract information from.
iepoch	The bin number, or epoch number, to return a time for. Unless edge is specified and requests the ending time, the time returned is the starting time for this bin.
edge	If specified, and if edge is larger than 1, the trailing edge time of the bin is returned. Any other value, or if this argument is not specified, returns the leading edge time of the bin.

var = get_schedule_length()

```
integer                                  :: get_schedule_length
type(schedule_type), intent(in)          :: schedule
```

Return the total number of intervals/bins/epochs defined by this schedule.

schedule	Return number of time intervals in this schedule.

```
type schedule_type
   private
   integer :: num_bins
   integer :: current_bin
   logical :: last_bin
   integer :: calendar
   character(len=32) :: calendarstring
   type(time_type)           :: binwidth
   type(time_type)           :: bininterval
   type(time_type), pointer :: binstart(   :) => NULL()
   type(time_type), pointer :: binend(     :) => NULL()
   real(digits12),  pointer :: epoch_start(:) => NULL()
   real(digits12),  pointer :: epoch_end(  :) => NULL()
end type schedule_type
```

This type is used to define a schedule.

6.128.5 Files

filename	purpose
input.nml	to read the schedule_mod namelist

6.128.6 References

- none

6.128.7 Private components

N/A

6.129 MODULE obs_kind_mod

6.129.1 Overview

Introduction

This module provides definitions of specific observation types and generic variable quantities, routines for mapping between integer identifiers and string names, routines for reading and writing this information, and routines for determining whether and how to process observations from an observation sequence file.

The distinction between quantities and types is this: `Quantities` apply both to observations and to state vector variables. Knowing the type of an observation must be sufficient to compute the correct forward operator. The quantity associated with an observation must be sufficient to identify which variable in the state vector should be used to compute the expected value. `Types` only apply to observations, and are usually observation-platform dependent. Making distinctions between different observation sources by using different types allows users to selectively assimilate, evaluate, or ignore them.

Examples and use

Generic quantities are associated with an observation type or with a model state variable. An example quantity is `QTY_U_WIND_COMPONENT`. Multiple different specific observation types can be associated with this generic quantity, for instance `RADIOSONDE_U_WIND_COMPONENT`, `ACARS_U_WIND_COMPONENT`, and `SAT_U_WIND_COMPONENT`. Generic quantities are defined via an integer parameter statement at the start of this module. As new generic quantities are needed they are added to this list. Generic quantity integer parameters are required to start with `QTY_` and observation types are NOT allowed to start with `QTY_`.

Typically quantities are used by model-interface files `models/xx/model_mod.f90`, observation forward operator files `observations/forward_operators/obs_def_xx_mod.f90`, and observation converter programs `observations/obs_converters/xx/xx.f90`.

The obs_kind module being described here is created by the program `preprocess` from two categories of input files. First, a DEFAULT obs_kind module (normally called `DEFAULT_obs_kind_mod.F90` and documented in this directory) is used as a template into which the preprocessor incorporates information from zero or more special obs_def modules (such as `obs_def_1d_state_mod.f90` or `obs_def_reanalysis_bufr_mod.f90`) which are documented in the obs_def directory. If no special obs_def files are included in the preprocessor namelist, a minimal `obs_kind_mod.f90` is created which can only support identity forward observation operators.

All of the build scripts in DART remove the existing `obs_kind_mod.f90` file and regenerate it using the `preprocess` program. Do not add new quantities to `obs_kind_mod.f90`, because these changes will not be kept when you run *quickbuild.csh*.

Adding additional quantities

New quantities should be added to a quantity file, for example a new ocean quantity should be added to `ocean_quantities_mod.f90`. The quantity files are in `assimilation_code/modules/observations/`.

Every line in a quantity file between the start and end markers must be a comment or a quantity definition (QTY_string). Multiple name-value pairs can be specified for a quantity but are not required. For example, temperature may be defined: `!   QTY_TEMPERATURE units="K" minval=0.0`. Comments are allowed between quantity definitions or on the same line as the definition. The code snippet below shows acceptable formats for quantity definitions

`! BEGIN DART PREPROCESS QUANTITY DEFINITIONS ! ! Formats accepted:  !  ! QTY_string ! QTY_string name=value ! QTY_string name=value name2=value2 ! ! QTY_string ! comments ! ! ! comment ! ! END DART PREPROCESS QUANTITY DEFINITIONS`

Implementation details

The obs_kind module contains an automatically-generated list of integer parameters, derived from the obs_def files, an integer parameter `max_defined_types_of_obs`, and an automatically-generated list of initializers for the `obs_type_type` derived type that defines the details of each observation type that has been created by the preprocess program. Each entry contains the integer index of the observation type, the string name of the observation type (which is identical to the F90 identifier), the integer index of the associated generic quantities, and three logicals indicating whether this observation type is to be assimilated, evaluated only (forward operator is computed but not assimilated), assimilated but has externally computed forward operator values in the input observation sequence file, or ignored entirely. The logicals initially default to .false. and are set to .true. via the `&obs_kind_nml` namelist. A second derived type `obs_qty_type` maps the integer parameter for a quantity to the quantity name (a string), and stores any additional pair-value metadata for that quantity.

6.129.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand ‘&’ and terminate with a slash ‘/’. Character strings that contain a ‘/’ must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_kind_nml
   assimilate_these_obs_types          = 'null',
   evaluate_these_obs_types            = 'null'
   use_precomputed_FOs_these_obs_types = 'null'
 /
```

Controls what observation types are to be assimilated, evaluated, or ignored. For each entry, a list of observation type names can be specified. Any name in the obs_type_type table is eligible. Specifying a name that is not in the table results in an error. Specifying the same name for both namelist entries also results in an error. Observation types specified in the list for assimilate_these_obs_types are assimilated. Those in the evaluate_these_obs_types list have their forward operators computed and included in diagnostic files but are not assimilated. An observation type that is specified in neither list is ignored. Identity observations, however, are always assimilated if present in the obs_seq.out file.

Item	Type	Description
assimilate_these_obs_types	character(len=31), dimension(:)	Names of observation types to be assimilated.
evaluate_these_obs_types	character(len=31), dimension(:)	Names of observation types to be evaluated only.
use_precomputed_FOs_these_obs_types	character(len=31), dimension(:)	If the forward operator values have been precomputed outside of filter, for example for radiances or other compute intensive computations, the ensemble of forward operator values can be stored in the observation sequence file. For any type listed here, the forward operator interpolation code will not be called and the values in the file will be used instead.

For example:

```
&obs_kind_nml
   assimilate_these_obs_types = 'RADIOSONDE_TEMPERATURE',
                                'RADIOSONDE_U_WIND_COMPONENT',
                                'RADIOSONDE_V_WIND_COMPONENT',
   evaluate_these_obs_types   = 'RADIOSONDE_SURFACE_PRESSURE',
  use_precomputed_FOs_these_obs_types = 'RADIANCE'
/
```

would assimilate temperature and wind observations, but only compute the forward operators for surface pressure obs. Radiance observations have precomputed values for each ensemble member in the input observation sequence file which would be used instead of calling the forward operator code.

6.129.3 Modules used

```
utilities_mod
```

6.129.4 Public interfaces

use obs_def_mod, only :	max_defined_types_of_obs
	get_num_types_of_obs
	get_num_quantities
	get_name_for_type_of_obs
	get_name_for_quantity
	get_index_for_type_of_obs
	get_index_for_quantity
	assimilate_this_type_of_obs
	evaluate_this_type_of_obs
	get_quantity_for_type_of_obs
	write_type_of_obs_table
	read_type_of_obs_table
	get_type_of_obs_from_menu
	map_type_of_obs_table
	GENERIC_QTY_DEFINITIONS
	OBSERVATION_TYPES

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

integer, parameter :: max_defined_types_of_obs

The total number of available observation types in the obs_type_type table. This value is added by the preprocess program and depends on which `obs_def_xxx_mod.f90` files are listed in the &preprocess_nml namelist.

There is also a function interface which is an alternate method to get this value. In some cases the code requires a parameter value known at compile time (for declaring a fixed length array, for example). For an array allocated at run time the size can be returned by the function interface.

var = get_num_types_of_obs()

```
integer :: get_num_types_of_obs
```

Returns the number of different specific observation types (e.g. RADIOSONDE_TEMPERATURE, AIRCRAFT_SPECIFIC_HUMIDITY) defined in the obs_kind_mod.f90 file. This file is generated by the preprocess program. This is the same value as the public 'max_defined_types_of_obs' above.

`var`	Integer count of the total number of specific types defined in the obs_kind_mod.f90 file.

var = get_num_quantities()

```
integer :: get_num_quantities
```

Returns the number of different generic quantities (e.g. QTY_TEMPERATURE, QTY_SPECIFIC_HUMIDITY) defined in the obs_kind_mod.f90 file. This file is generated by the preprocess program.

`var`	Integer count of the total number of generic quantities defined in the obs_kind_mod.f90 file.

var = get_name_for_type_of_obs(obs_type_ind)

```
character(len=32)              :: get_name_for_type_of_obs
integer, intent(in)            :: obs_type_ind
```

Given an integer index return the string name of the corresponding specific observation type (e.g. "RADIOSONDE_TEMPERATURE", "AIRCRAFT_SPECIFIC_HUMIDITY"). This string is the same as the F90 identifier associated with the integer index.

`var`	Name string associated with this entry in the obs_type_type table.
`obs_type_ind`	An integer index into the obs_type_type table.

var = get_name_for_quantity(obs_qty_ind)

```
character(len=32)              :: get_name_for_quantity
integer, intent(in)            :: obs_qty_ind
```

Given an integer index return the string name of the corresponding generic quantity (e.g. "QTY_TEMPERATURE", "QTY_SPECIFIC_HUMIDITY"). This string is the same as the F90 identifier associated with the integer index.

`var`	Name string associated with this entry in the obs_qty_type table.
`obs_qty_ind`	An integer index into the obs_qty_type table.

var = get_index_for_type_of_obs(obs_type_name)

```
integer                           :: get_index_for_type_of_obs
character(len=*), intent(in)   :: obs_type_name
```

Given the name of a specific observation type (e.g. "RADIOSONDE_TEMPERATURE", "AIRCRAFT_SPECIFIC_HUMIDITY"), returns the index of the entry in the obs_type_type table with this name. If the name is not found in the table, a -1 is returned. The integer returned for a successful search is the value of the integer parameter with the same identifier as the name string.

get_index_for_type_of_obs	Integer index into the obs_type_type table entry with name string corresponding to obs_type_name.
obs_type_name	Name of specific observation type found in obs_type_type table.

var = get_index_for_quantity(obs_qty_name)

```
integer                           :: get_index_for_quantity
character(len=32), intent(in) :: obs_qty_name
```

Given the name of a generic quantity (e.g. "QTY_TEMPERATURE", "QTY_SPECIFIC_HUMIDITY"), returns the index of the entry in the obs_qty_type table with this name. If the name is not found in the table, a -1 is returned. The integer returned for a successful search is the value of the integer parameter with the same identifier as the name string.

get_index_for_quantity	Integer index into the obs_qty_type table entry with name string corresponding to obs_qty_name.
obs_qty_name	Name of generic kind found in obs_qty_type table.

var = assimilate_this_type_of_obs(obs_type_ind)

```
logical                 :: assimilate_this_type_of_obs
integer, intent(in)   :: obs_type_ind
```

Given the integer index associated with a specific observation type (e.g. RADIOSONDE_TEMPERATURE, AIRCRAFT_SPECIFIC_HUMIDITY), return true if this observation type is to be assimilated, otherwise false. The parameter defined by this name is used as an integer index into the obs_type_type table to return the status of this type.

var	Returns true if this entry in the obs_type_type table is to be assimilated.
obs_type_ind	An integer index into the obs_type_type table.

var = evaluate_this_type_of_obs(obs_type_ind)

```
logical              :: evaluate_this_type_of_obs
integer, intent(in)  :: obs_type_ind
```

Given the integer index associated with a specific observation type (e.g. RADIOSONDE_TEMPERATURE, AIRCRAFT_SPECIFIC_HUMIDITY), return true if this observation type is to be evaluated only, otherwise false. The parameter defined by this name is used as an integer index into the obs_type_type table to return the status of this type.

`var`</ TD>	Returns true if this entry in the obs_type_type table is to be evaluated.
`obs_type_ind`	An integer index into the obs_type_type table.

var = get_quantity_for_type_of_obs(obs_type_ind)

```
integer              :: get_quantity_for_type_of_obs
integer, intent(in)  :: obs_type_ind
```

Given the integer index associated with a specific observation type (e.g. RADIOSONDE_TEMPERATURE, AIRCRAFT_SPECIFIC_HUMIDITY), return the generic quantity associated with this type (e.g. QTY_TEMPERATURE, QTY_SPECIFIC_HUMIDITY). The parameter defined by this name is used as an integer index into the obs_type_type table to return the generic quantity associated with this type.

`var`</ TD>	Returns the integer GENERIC quantity index associated with this obs type.
`obs_type_ind`	An integer index into the obs_type_type table.

call write_type_of_obs_table(ifile [, fform, use_list])

```
integer,                       intent(in) :: ifile
character(len=*), optional,   intent(in) :: fform
integer,           optional,   intent(in) :: use_list(:)
```

Writes out information about all defined observation types from the obs_type_type table. For each entry in the table, the integer index of the observation type and the associated string are written. These appear in the header of an obs_sequence file. If given, the *use_list(:)* must be the same length as the max_obs_specific count. If greater than 0, the corresponding index will be written out; if 0 this entry is skipped. This allows a table of contents to be written which only includes those types actually being used.

`ifile`	Unit number of output observation sequence file being written.
fform	Optional format for file. Default is FORMATTED.
use_list(:)	Optional integer array the same length as the number of specific types (from get_num_types_of_obs() or the public max_defined_types_of_obs). If value is larger than 0, the corresponding type information will be written out. If 0, it will be skipped. If this argument is not specified, all values will be written.

call read_type_of_obs_table(ifile, pre_I_format [, fform])

```
integer,                      intent(in) :: ifile
logical,                      intent(in) :: pre_I_format !(deprecated)
character(len=*), optional, intent(in) :: fform
```

Reads the mapping between integer indices and observation type names from the header of an observation sequence file and prepares mapping to convert these to values defined in the obs_type_type table. If pre_I_format is true, there is no header in the observation sequence file and it is assumed that the integer indices for observation types in the file correspond to the storage order of the obs_type_type table (integer index 1 in the file corresponds to the first table entry, etc.) Support for pre_I_format is deprecated and may be dropped in future releases of DART.

`ifile`	Unit number of output observation sequence file being written.
`pre_I_format`	True if the file being read has no obs type definition header (deprecated).
fform	Optional format for file. Default is FORMATTED.

var = get_type_of_obs_from_menu()

```
integer              :: get_type_of_obs_from_menu
```

Interactive input of observation type. Prompts user with list of available types and validates entry before returning.

`var`	Integer index of observation type.

var = map_type_of_obs_table(obs_def_index)

```
integer              :: map_type_of_obs_table
integer, intent(in)  :: obs_def_index
```

Maps from the integer observation type index in the header block of an input observation sequence file into the corresponding entry in the obs_type_type table. This allows observation sequences that were created with different obs_kind_mod.f90 versions to be used with the current obs_kind_mod.

`var`	Index of this observation type in obs_type_type table.
`obs_def_index`	Index of observation type from input observation sequence file.

`integer, parameter ::  QTY_.....`

All generic quantities available are public parameters that begin with `QTY_`.

integer, parameter :: SAMPLE_OBS_TYPE

A list of all observation types that are available is provided as a set of integer parameter statements. The F90 identifiers are the same as the string names that are associated with this identifier in the obs_type_type table.

6.129.5 Files

- &obs_kind_nml in input.nml
- Files containing input or output observation sequences.

6.129.6 References

- none

6.130 MODULE DEFAULT_obs_kind_mod

6.130.1 Overview

The observation types supported by DART are defined at the time the programs are compiled. Only those observation types of interest need to be included. This is the input template file which is read by the program `preprocess` to create the `obs_kind_mod.f90` (documented separately in this directory). Information from zero or more special obs_def modules (such as `obs_def_1d_state_mod.f90` or `obs_def_reanalysis_bufr_mod.f90`, documented in the obs_def directory) are incorporated into the template provided by DEFAULT_obs_def_kind. If no special obs_def files are included in the preprocessor namelist, a minimal `obs_kind_mod.f90` is created which can only support identity forward observation operators.

To add a new specific observation type, see the *MODULE obs_def_mod* documentation. If adding this type includes defining a new generic observation quantity, then the quantity must be added to the table in this file, with a unique integer value defined for it. The new value(s) must also be added to the string-to-number section in the `initialize_module()` routine below. In a planned future update to the preprocess program it is expected that this quantities table will become autogenerated, much like the types table described below, so no editing of this file will be needed. But until then, quantities must be added by here, by hand. And note that they must be added to the template file, `DEFAULT_obs_kind_mod.F90` and not the output of the preprocess program, `obs_kind_mod.f90`. The latter file is overwritten each time preprocess runs, deleting any hand editing done on that file.

The rest of this documentation describes the special formatting that is included in the DEFAULT_obs_kind_mod in order to guide the preprocess program. Two sections of code are inserted into the DEFAULT_obs_def_kind module from each of the special obs_def modules that are requested. The insertion point for each section is denoted by a special comment line that must be included VERBATIM in DEFAULT_obs_kind_mod. These special comment lines and their significance are:

1.
   ```
   ! DART PREPROCESS INTEGER DECLARATION INSERTED HERE
   ```

   Each of the observation type identifiers are inserted here, and are given both the `parameter` and `public` attributes. The associated integer values are generated sequentially, starting at 1. If a different set of special obs_def modules is specified via the namelist for preprocess, the integer value associated with a given observation type could change so these integers are for use internal to a given program execution only. Observation types are permanently and uniquely identified by a string with character values identical to the integer identifier here. In other words, an observation quantity with integer parameter identifier RAW_STATE_VARIABLE will always be associated with a string "RAW_STATE_VARIABLE" (see below).

2.
   ```
   ! DART PREPROCESS OBS_QTY_INFO INSERTED HERE
   ```

   The entries for the obs_kind_info table are initialized here. This table has a row for each observation quantity defined in the public and integer parameter declarations discussed above. For each obs_kind, the obs_kind info table contains the following information:

   1. The integer index associated with this observation quantity. This is the value that is defined in the integer parameter statement above.
   2. A character string with maximum length 32 characters that uniquely identifies the observation quantity. This string is identical to the F90 identifier used in the integer parameter statements created above.
   3. A generic variable quantity referenced by an integer. The obs_kind_mod contains a table that defines an integer parameter that uniquely identifies each generic observation quantity. Generic quantities can be associated with both a model state variable or an observation quantity. For instance, the generic quantity QTY_U_WIND_COMPONENT might be associated with model U state variables, with RADIOSONDE_U_WIND_COMPONENT observation types, and with ACARS_U_WIND_COMPONENT observation types. The distinction between generic quantites and observation types is important and potentially confusing, so be careful to understand this point.
   4. A logical variable indicating whether observations of this type are to be assimilated in the current experiment. This variable is set from the obs_kind namelist (see obs_kind_mod.html in this directory). The default value here is false when the obs_def_mod is created by the preprocess program.
   5. A logical variable indicating whether observations of this type are to be evaluated (forward operator computed and recorded) but not assimilated (again set by namelist). The default value here is false when the obs_def_mod is created by the preprocess program.
   6. A logical variable indicating whether observations of this type should use precomputed forward operator values in the observation sequence file. The default value here is false when the obs_def_mod is created by the preprocess program.

6.131 MODULE obs_sequence_mod

6.131.1 Overview

Provides interfaces to the observation type and observation sequence type. An observation contains everything there is to know about an observation including all metadata contained in the observation definition and any number of copies of data associated with the observation (for instance an actual observation, an ensemble of first guess values, etc). An observation sequence is a time-ordered set of observations that is defined by a linked list so that observations can be easily added or deleted. A number of commands to extract observations depending on the times at which they were taken are provided. For now, the observations are only ordered by time, but the ability to add extra sort keys could be added.

These routines are commonly used in conversion programs which read observation data from various formats and create a DART observation sequence in memory, and then write it out to a file. See the observations directory for examples of programs which create and manipulate observations using this routines.

6.131.2 Other modules used

```
types_mod
location_mod (depends on model_choice)
obs_def_mod
time_manager_mod
utilities_mod
obs_kind_mod
```

6.131.3 Public interfaces

use obs_sequence_mod, only :	obs_sequence_type
	init_obs_sequence
	interactive_obs_sequence
	get_num_copies
	get_num_qc
	get_num_obs
	get_max_num_obs
	get_copy_meta_data
	get_qc_meta_data
	get_next_obs
	get_prev_obs
	get_next_obs_from_key
	get_prev_obs_from_key
	insert_obs_in_seq
	delete_obs_from_seq
	set_copy_meta_data
	set_qc_meta_data
	get_first_obs
	get_last_obs
	add_copies
	add_qc
	write_obs_seq
	read_obs_seq
	append_obs_to_seq
	get_obs_from_key
	get_obs_time_range
	set_obs
	get_time_range_keys
	get_num_times
	static_init_obs_sequence
	destroy_obs_sequence

continues on next page

Table 7 – continued from previous page

	read_obs_seq_header
	get_expected_obs
	delete_seq_head
	delete_seq_tail
	LINKS BELOW FOR OBS_TYPE INTERFACES
	obs_type
	init_obs
	destroy_obs
	get_obs_def
	set_obs_def
	get_obs_values
	set_obs_values
	replace_obs_values
	get_qc
	set_qc
	replace_qc
	write_obs
	read_obs
	interactive_obs
	copy_obs
	assignment(=)

```
type obs_sequence_type
   private
   integer                          :: num_copies
   integer                          :: num_qc
   integer                          :: num_obs
   integer                          :: max_num_obs
   character(len=64), pointer       :: copy_meta_data(:)
   character(len=64), pointer       :: qc_meta_data(:)
   integer                          :: first_time
   integer                          :: last_time
   type(obs_type), pointer          :: obs(:)
end type obs_sequence_type
```

The obs_sequence type represents a series of observations including multiple copies of data and quality control fields and complete metadata about the observations. The sequence is organized as an integer pointer linked list using a fixed array of storage for obs (type obs_type). Each observation points to the previous and next observation in time order (additional sort keys could be added if needed) and has a unique integer key (see obs_type below). The maximum number of observations in the sequence is represented in the type as max_num_obs, the current number of observations is in num_obs. The number of quality control (qc) fields per observation is num_qc and the number of data values associated with each observation is num_copies. Metadata for each copy of the data is in copy_meta_data and metadata for the qc fields is in qc_meta_data. The first and last pointers into the time linked list are in first_time and last_time. A capability to write and read an obs_sequence structure to disk is available. At present, the entire observation sequence is read in to core memory. An on-disk implementation may be necessary for very large observational datasets.

Component	Description
num_copies	Number of data values associated with each observation.
num_qc	Number of qc fields associated with each observation.
num_obs	Number of observations currently in sequence.
max_num_obs	Upper bounds on number of observations in sequence.
copy_meta_data	Text describing each copy of data associated with observations.
qc_meta_data	Text describing each quality control field.
first_time	Location of first observation in sequence.
last_time	Location of last observation in sequence.
obs	Storage for all of the observations in the sequence.

```
type obs_type
   private
   integer              :: key
   type(obs_def_type) :: def
   real(r8), pointer   :: values(:)
   real(r8), pointer   :: qc(:)
   integer              :: prev_time
   integer              :: next_time
   integer              :: cov_group
end type obs_type
```

Structure to represent everything known about a given observation and to help with storing the observation in the observation sequence structure (see above). The prev_time and next_time are integer pointers that allow a linked list sorted on time to be constructed. If needed, other sort keys could be introduced (for instance by time available?). Each observation in a sequence has a unique key and each observation has an obs_def_type that contains all the definition and metadata for the observation. A set of values is associated with the observation along with a set of qc fields. The cov_group is not yet implemented but will allow non-diagonal observation error covariances in a future release.

Component	Description
key	Unique integer key when in an obs_sequence.
def	The definition of the observation (see obs_def_mod).
values	Values associated with the observation.
qc	Quality control fields associated with the observation.
prev_time	When in an obs_sequence, points to previous time sorted observation.
next_time	When in an obs_sequence, points to next time sorted observation.
cov_group	Not currently implemented.

call init_obs_sequence(seq, num_copies, num_qc, expected_max_num_obs)

```
type(obs_sequence_type), intent(out) :: seq
integer,                 intent(in)  :: num_copies
integer,                 intent(in)  :: num_qc
integer,                 intent(in)  :: expected_max_num_obs
```

Constructor to create a variable of obs_sequence_type. This routine must be called before using an obs_sequence_type. The number of copies of the data to be associated with each observation (for instance the observation from an instrument, an ensemble of prior guesses, etc.) and the number of quality control fields associated with each observation must be specified. Also, an estimated upper bound on the number of observations to be stored in the sequence is helpful in making creation of the sequence efficient.

seq	The observation sequence being constructed
num_copies	Number of copies of data to be associated with each observation
num_qc	Number of quality control fields associated with each observation
expected_max_num_obs	An estimate of the largest number of observations the sequence might contain

var = interactive_obs_sequence()

```
type(obs_sequence_type) :: interactive_obs_sequence
```

Uses input from standard in to create an observation sequence. Initialization of the sequence is handled by the function.

var	An observation sequence created from standard input

var = get_num_copies(seq)

```
integer                              :: get_num_copies
type(obs_sequence_type), intent(in)  :: seq
```

Returns number of copies of data associated with each observation in an observation sequence.

var	Returns number of copies of data associated with each observation in sequence
seq	An observation sequence

var = get_num_qc(seq)

```
integer                              :: get_num_qc
type(obs_sequence_type), intent(in)  :: seq
```

Returns number of quality control fields associated with each observation in an observation sequence.

var	Returns number of quality control fields associated with each observation in sequence
seq	An observation sequence

var = get_num_obs(seq)

```
integer                                :: get_num_obs
type(obs_sequence_type), intent(in)  :: seq
```

Returns number of observations currently in an observation sequence.

var	Returns number of observations currently in an observation sequence
seq	An observation sequence

var = get_max_num_obs(seq)

```
integer                                :: get_max_num_obs
type(obs_sequence_type), intent(in)  :: seq
```

Returns maximum number of observations an observation sequence can hold.

var	Returns maximum number of observations an observation sequence can hold
seq	An observation sequence

var = get_copy_meta_data(seq, copy_num)

```
character(len=64)                      :: get_copy_meta_data
type(obs_sequence_type), intent(in)  :: seq
integer,                 intent(in)  :: copy_num
```

Returns metadata associated with a given copy of data in an observation sequence.

var	Returns metadata associated with a copy of data in observation sequence
seq	An observation sequence
copy_num	Return metadata for this copy

var = get_qc_meta_data(seq,qc_num)

```
character(len=64)                      :: get_qc_meta_data
type(obs_sequence_type), intent(in)  :: seq
integer,                 intent(in)  :: qc_num
```

Returns metadata associated with a given copy of quality control fields associated with observations in an observation sequence.

var	Returns metadata associated with a given qc copy
seq	An observation sequence
qc_num	Return metadata for this copy

call get_next_obs(seq, obs, next_obs, is_this_last)

```
type(obs_sequence_type), intent(in)  :: seq
type(obs_type),          intent(in)  :: obs
type(obs_type),          intent(out) :: next_obs
logical,                 intent(out) :: is_this_last
```

Given an observation in a sequence, returns the next observation in the sequence. If there is no next observation, is_this_last is set to true.

seq	An observation sequence
obs	Find the next observation after this one
next_obs	Return the next observation here
is_this_last	True if obs is the last obs in sequence

call get_prev_obs(seq, obs, prev_obs, is_this_first)

```
type(obs_sequence_type), intent(in)  :: seq
type(obs_type),          intent(in)  :: obs
type(obs_type),          intent(out) :: prev_obs
logical,                 intent(out) :: is_this_first
```

Given an observation in a sequence, returns the previous observation in the sequence. If there is no previous observation, is_this_first is set to true.

seq	An observation sequence
obs	Find the previous observation before this one
prev_obs	Return the previous observation here
is_this_first	True if obs is the first obs in sequence

call get_next_obs_from_key(seq, last_key_used, next_obs, is_this_last)

```
type(obs_sequence_type), intent(in)  :: seq
integer,                 intent(in)  :: last_key_used
type(obs_type),          intent(out) :: next_obs
logical,                 intent(out) :: is_this_last
```

Given the last key used in a sequence, returns the next observation in the sequence. If there is no next observation, is_this_last is set to true.

seq	An observation sequence
last_key_used	Find the next observation after this key
next_obs	Return the next observation here
is_this_last	True if obs is the last obs in sequence

call get_prev_obs_from_key(seq, last_key_used, prev_obs, is_this_first)

```
type(obs_sequence_type), intent(in)  :: seq
integer,                 intent(in)  :: last_key_used
type(obs_type),          intent(out) :: prev_obs
logical,                 intent(out) :: is_this_first
```

Given the last key used in a sequence, returns the previous observation in the sequence. If there is no previous observation, is_this_first is set to true.

seq	An observation sequence
last_key_used	Find the previous observation before this key
prev_obs	Return the previous observation here
is_this_first	True if obs is the first obs in sequence

call get_obs_from_key(seq, key, obs)

```
type(obs_sequence_type), intent(in)  :: seq
integer,                 intent(in)  :: key
type(obs_type),          intent(out) :: obs
```

Each entry in an observation sequence has a unique integer key. This subroutine returns the observation given an integer key.

seq	An observation sequence
key	Return the observation with this key
obs	The returned observation

call insert_obs_in_seq(seq, obs [, prev_obs])

```
type(obs_sequence_type),  intent(inout) :: seq
type(obs_type),           intent(inout) :: obs
type(obs_type), optional, intent(in)    :: prev_obs
```

Inserts an observation in a sequence in appropriate time order. If the optional argument prev_obs is present, the new observation is inserted directly after the prev_obs. If an incorrect prev_obs is provided so that the sequence is no longer time ordered, bad things will happen.

seq	An observation sequence
obs	An observation to be inserted in the sequence
prev_obs	If present, says the new observation belongs immediately after this one

call delete_obs_from_seq(seq, obs)

```
type(obs_sequence_type), intent(inout) :: seq
type(obs_type),          intent(inout) :: obs
```

Given an observation and a sequence, removes the observation with the same key from the observation sequence.

seq	An observation sequence
obs	The observation to be deleted from the sequence

call set_copy_meta_data(seq, copy_num, meta_data)

```
type(obs_sequence_type), intent(inout) :: seq
integer,                 intent(in)    :: copy_num
character(len=64),       intent(in)    :: meta_data
```

Sets the copy metadata for this copy of the observations in an observation sequence.

seq	An observation sequence
copy_num	Set metadata for this copy of data
meta_data	The metadata

call set_qc_meta_data(seq, qc_num, meta_data)

```
type(obs_sequence_type), intent(inout) :: seq
integer,                 intent(in)    :: qc_num
character(len=64),       intent(in)    :: meta_data
```

Sets the quality control metadata for this copy of the qc in an observation sequence.

seq	An observation sequence
qc_num	Set metadata for this quality control field
meta_data	The metadata

var = get_first_obs(seq, obs)

```
logical                                   :: get_first_obs
type(obs_sequence_type), intent(in)   :: seq
type(obs_type),          intent(out) :: obs
```

Returns the first observation in a sequence. If there are no observations in the sequence, the function returns false, else true.

`var`	Returns false if there are no obs in sequence
`seq`	An observation sequence
`obs`	The first observation in the sequence

var = get_last_obs(seq, obs)

```
logical                                   :: get_last_obs
type(obs_sequence_type), intent(in)   :: seq
type(obs_type),          intent(out) :: obs
```

Returns the last observation in a sequence. If there are no observations in the sequence, the function returns false, else true.

`var`	Returns false if there are no obs in sequence
`seq`	An observation sequence
`obs`	The last observation in the sequence

call add_copies(seq, num_to_add)

```
type(obs_sequence_type), intent(inout) :: seq
integer,                 intent(in)    :: num_to_add
```

Increases the number of copies of data associated with each observation by num_to_add. The current implementation re-creates the entire observation sequence by deallocating and reallocating each entry with a larger size.

`seq`	An observation sequence
`num_to_add`	Number of copies of data to add

call add_qc(seq, num_to_add)

```
type(obs_sequence_type), intent(inout) :: seq
integer,                 intent(in)    :: num_to_add
```

Increases the number of quality control fields associated with each observation by num_to_add. The current implementation re-creates the entire observation sequence by deallocating and reallocating each entry with a larger size.

`seq`	An observation sequence
`num_to_add`	Number of quality control fields to add

call read_obs_seq(file_name, add_copies, add_qc, add_obs, seq)

```
character(len=*),        intent(in)  :: file_name
integer,                 intent(in)  :: add_copies
integer,                 intent(in)  :: add_qc
integer,                 intent(in)  :: add_obs
type(obs_sequence_type), intent(out) :: seq
```

Read an observation sequence from `file_name`. The sequence will have enough space for the number of observations in the file plus any additional space requested by the "add_xx" args. It is more efficient to allocate the additional space at create time rather than try to add it in later. The arguments can specify that the caller wants to add additional data copies associated with each observation, or to add additional quality control fields, or to add space for additional observations. The format of the file (`formatted` vs. `unformatted`) has been automatically detected since the I release. The obs_sequence file format with I and later releases has a header that associates observation type strings with an integer which was not present in previous versions. I format files are no longer supported.

`file_name`	Read from this file
`add_copies`	Add this number of copies of data to the obs_sequence on file
`add_qc`	Add this number of qc fields to the obs_sequence on file
`add_obs`	Add space for this number of additional observations to the obs_sequence on file
`seq`	The observation sequence read in with any additional space

call write_obs_seq(seq, file_name)

```
type(obs_sequence_type), intent(in) :: seq
character(len=*),        intent(in) :: file_name
```

Write the observation sequence to file file_name. The format is controlled by the namelist parameter write_binary_obs_sequence.

`seq`	An observation sequence
`file_name`	Write the sequence to this file

call set_obs(seq,obs [, key_in])

```
type(obs_sequence_type), intent(inout) :: seq
type(obs_type),          intent(in)    :: obs
integer, optional,       intent(in)    :: key_in
```

Given an observation, copies this observation into the observation sequence using the key specified in the observation. If the optional key_in argument is present, the observation is instead copied into this element of the observation sequence (and the key is changed to be key_in).

`seq`	An observation sequence
`obs`	Observation to be put in sequence
key_in	If present, the obs is copied into this key of the sequence

call append_obs_to_seq(seq, obs)

```
type(obs_sequence_type), intent(inout) :: seq
type(obs_type),          intent(inout) :: obs
```

Append an observation to an observation sequence. An error results if the time of the observation is not equal to or later than the time of the last observation currently in the sequence.

`seq`	An observation sequence
`obs`	Append this observation to the sequence

call get_obs_time_range(seq, time1, time2, key_bounds, num_keys, out_of_range [, obs])

```
type(obs_sequence_type),  intent(in)  :: seq
type(time_type),          intent(in)  :: time1
type(time_type),          intent(in)  :: time2
integer, dimension(2),    intent(out) :: key_bounds
integer,                  intent(out) :: num_keys
logical,                  intent(out) :: out_of_range
type(obs_type), optional, intent(in)  :: obs
```

Given a time range specified by a beginning and ending time, find the keys that bound all observations in this time range and the number of observations in the time range. The routine get_time_range_keys can then be used to get a list of all the keys in the range if desired. The logical out_of_range is returned as true if the beginning time of the time range is after the time of the latest observation in the sequence. The optional argument obs can increase the efficiency of the search through the sequence by indicating that all observations before obs are definitely at times before the start of the time range.

seq	An observation sequence
time1	Lower time bound
time2	Upper time bound
key_bounds	Lower and upper bounds on keys that are in the time range
num_keys	Number of keys in the time range
out_of_range	Returns true if the time range is entirely past the time of the last obs in sequence
obs	If present, can start search for time range from this observation

call get_time_range_keys(seq, key_bounds, num_keys, keys)

```
type(obs_sequence_type),      intent(in)  :: seq
integer, dimension(2),        intent(in)  :: key_bounds
integer,                      intent(in)  :: num_keys
integer, dimension(num_keys), intent(out) :: keys
```

Given the keys of the observations at the start and end of a time range and the number of observations in the time range (these are returned by get_obs_time_range()), return a list of the keys of all observations in the time range. Combining the two routines allows one to get a list of all observations in any time range by key. The keys array must be at least num_keys long to hold the return values.

seq	An observation sequence
key_bounds	Keys of first and last observation in a time range
num_keys	Number of obs in the time range
keys	Output list of keys of all obs in the time range

var = get_num_times(seq)

```
integer                              :: get_num_times
type(obs_sequence_type), intent(in)  :: seq
```

Returns the number of unique times associated with observations in an observation sequence.

var	Number of unique times for observations in a sequence
seq	An observation sequence

var = get_num_key_range(seq, key1, key2)

```
integer                              :: get_num_key_range
type(obs_sequence_type), intent(in)  :: seq
integer, optional,       intent(in)  :: key1, key2
```

Returns the number of observations between the two given keys. The default key numbers are the first and last in the sequence file. This routine can be used to count the actual number of observations in a sequence and will be accurate even if the sequence has been trimmed with delete_seq_head() or delete_seq_tail().

`var`	Number of unique times for observations in a sequence
`seq`	An observation sequence
`key1`	The starting key number. Defaults to the first observation in the sequence.
`key2`	The ending key number. Defaults to the last observation in the sequence.

call static_init_obs_sequence()

Initializes the obs_sequence module and reads namelists. This MUST BE CALLED BEFORE USING ANY OTHER INTERFACES.

call destroy_obs_sequence(seq)

```
type(obs_sequence_type), intent(inout) :: seq
```

Releases all allocated storage associated with an observation sequence.

`seq`	An observation sequence

call read_obs_seq_header(file_name, num_copies, num_qc, num_obs, max_num_obs, file_id, read_format, pre_I_format [, close_the_file])

```
character(len=*),   intent(in)  :: file_name
integer,            intent(out) :: num_copies
integer,            intent(out) :: num_qc
integer,            intent(out) :: num_obs
integer,            intent(out) :: max_num_obs
integer,            intent(out) :: file_id
character(len=*),   intent(out) :: read_format
logical,            intent(out) :: pre_I_format
logical, optional,  intent(in)  :: close_the_file
```

Allows one to see the global metadata associated with an observation sequence that has been written to a file without reading the whole file.

file_name	File contatining an obs_sequence
num_copies	Number of copies of data associated with each observation
num_qc	Number of quality control fields associated with each observation
num_obs	Number of observations in sequence
max_num_obs	Maximum number of observations sequence could hold
file_id	File channel/descriptor returned from opening the file
read_format	Either the string 'unformatted' or 'formatted'
pre_I_format	Returns .true. if the file was written before the observation type string/index number table was added to the standard header starting with the I release.
close_the_file	If specified and .TRUE. close the file after the header has been read. The default is to leave the file open.

call init_obs(obs, num_copies, num_qc)

```
type(obs_type), intent(out) :: obs
integer,        intent(in)  :: num_copies
integer,        intent(in)  :: num_qc
```

Initializes an obs_type variable. This allocates storage for the observation type and creates the appropriate obs_def_type and related structures. IT IS ESSENTIAL THAT OBS_TYPE VARIABLES BE INITIALIZED BEFORE USE.

obs	An obs_type data structure to be initialized
num_copies	Number of copies of data associated with observation
num_qc	Number of qc fields associated with observation

call destroy_obs(obs)

```
type(obs_type), intent(inout) :: obs
```

Destroys an observation variable by releasing all associated storage.

obs	An observation variable to be destroyed

call get_obs_def(obs, obs_def)

```
type(obs_type),     intent(in)  :: obs
type(obs_def_type), intent(out) :: obs_def
```

Extracts the definition portion of an observation.

`obs`	An observation
`obs_def`	The definition portion of the observation

call set_obs_def(obs, obs_def)

```
type(obs_type),     intent(out) :: obs
type(obs_def_type), intent(in)  :: obs_def
```

Given an observation and an observation definition, insert the definition in the observation structure.

`obs`	An observation whose definition portion will be updated
`obs_def`	The observation definition that will be inserted in obs

call get_obs_values(obs, values [, copy_indx])

```
type(obs_type),         intent(in)  :: obs
real(r8), dimension(:), intent(out) :: values
integer, optional,      intent(in)  :: copy_indx
```

Extract copies of the data from an observation. If *copy_indx* is present extract a single value indexed by *copy_indx* into `values(1)`. *copy_indx* must be between 1 and `num_copies`, inclusive. If *copy_indx* is not present extract all copies of data into the `values` array which must be `num_copies` long (See `get_num_copies`.)

`obs`	Observation from which to extract values
`values`	The values extracted
copy_indx	If present extract only this copy, otherwise extract all copies

call get_qc(obs, qc [, qc_indx])

```
type(obs_type),         intent(in)  :: obs
real(r8), dimension(:), intent(out) :: qc
integer, optional,      intent(in)  :: qc_indx
```

Extract quality control fields from an observation. If *qc_indx* is present extract a single field indexed by *qc_indx* into `qc(1)`. *qc_indx* must be between 1 and `num_qc`, inclusive. If *qc_indx* is not present extract all quality control fields into the `qc` array which must be `num_qc` long (See `get_num_qc`.)

`obs`	Observation from which to extract qc field(s)
`qc`	Extracted qc fields
qc_indx	If present extract only this field, otherwise extract all qc fields

call set_obs_values(obs, values [, copy_indx])

```
type(obs_type),          intent(out) :: obs
real(r8), dimension(:), intent(in)   :: values
integer, optional,       intent(in)   :: copy_indx
```

Set value(s) of data in this observation. If *copy_indx* is present set the single value indexed by *copy_indx* to `values(1)`. *copy_indx* must be between 1 and `num_copies`, inclusive. If *copy_indx* is not present set all copies of data from the `values` array which must be `num_copies` long (See `get_num_copies`.)

`obs`	Observation whose values are being set
`values`	Array of value(s) to be set
copy_indx	If present set only this copy of data, otherwise set all copies

call replace_obs_values(seq, key, values [, copy_indx])

```
type(obs_sequence_type), intent(inout) :: seq
integer,                 intent(in)    :: key
real(r8), dimension(:),  intent(in)    :: values
integer, optional,       intent(in)    :: copy_indx
```

Set value(s) of data in the observation from a sequence with the given `key`. If *copy_indx* is present set the single value indexed by *copy_indx* to `values(1)`. *copy_indx* must be between 1 and `num_copies`, inclusive. If *copy_indx* is not present set all copies of data from the `values` array which must be `num_copies` long (See `get_num_copies`.)

`seq`	Sequence which contains observation to update
`key`	Key to select which observation
`values`	Array of value(s) to be set
copy_indx	If present set only this copy of data, otherwise set all copies

call set_qc(obs, qc [, qc_indx])

```
type(obs_type),          intent(out) :: obs
real(r8), dimension(:), intent(in)   :: qc
integer, optional,       intent(in)   :: qc_indx
```

Sets the quality control fields in an observation. If *qc_indx* is present set a single field indexed by *qc_indx* to `qc(1)`. *qc_indx* must be between 1 and `num_qc`, inclusive. If *qc_indx* is not present set all quality control fields from the `qc` array which must be `num_qc` long (See `get_num_qc`.)

obs	Observation having its qc fields set
qc	Input values of qc fields
qc_indx	If present update only this field, otherwise update all qc fields

call replace_qc(seq, key, qc [, qc_indx])

```
type(obs_sequence_type), intent(inout) :: seq
integer,                 intent(in)    :: key
real(r8), dimension(:),  intent(in)    :: qc
integer, optional,       intent(in)    :: qc_indx
```

Set value(s) of the quality control fields in the observation from a sequence with the given `key`. If *qc_indx* is present set the single value indexed by *qc_indx* to `qc(1)`. *qc_indx* must be between 1 and `num_qc`, inclusive. If *qc_indx* is not present set all quality control fields from the `qc` array which must be `num_qc` long (See `get_num_qc`.)

seq	Observation sequence containing observation to update
key	Key to select which observation
qc	Input values of qc fields
qc_indx	If present, only update single qc field, else update all qc fields

call write_obs(obs, file_id, num_copies, num_qc)

```
type(obs_type), intent(in) :: obs
integer,        intent(in) :: file_id
integer,        intent(in) :: num_copies
integer,        intent(in) :: num_qc
```

Writes an observation and all its associated metadata to a disk file that has been opened with a format consistent with the namelist parameter `write_binary_obs_sequence`.

obs	Observation to be written to file
file_id	Channel open to file for writing
num_copies	The number of copies of data associated with the observation to be output
num_qc	The number of qc fields associated with the observation to be output

call read_obs(file_id, num_copies, add_copies, num_qc, add_qc, key, obs, read_format [, max_obs])

```
integer,              intent(in)    :: file_id
integer,              intent(in)    :: num_copies
integer,              intent(in)    :: add_copies
```

(continues on next page)

(continued from previous page)

```
integer,            intent(in)    :: num_qc
integer,            intent(in)    :: add_qc
integer,            intent(in)    :: key
type(obs_type),     intent(inout) :: obs
character(len=*),   intent(in)    :: read_format
integer, optional,  intent(in)    :: max_obs
```

Reads an observation from an obs_sequence file. The number of copies of data and the number of qc values associated with each observation must be provided. If additional copies of data or additional qc fields are needed, arguments allow them to be added. WARNING: The key argument is no longer used and should be removed.

`file_id`	Channel open to file from which to read
`num_copies`	Number of copies of data associated with observation in file
`add_copies`	Number of additional copies of observation to be added
`num_qc`	Number of qc fields associated with observation in file
`add_qc`	Number of additional qc fields to be added
`key`	No longer used, should be deleted
`obs`	The observation being read in
`read_format`	Either the string `'formatted'` or `'unformatted'`
max_obs	If present, specifies the largest observation key number in the sequence. This is used only for additional error checks on the next and previous obs linked list values.

call interactive_obs(num_copies, num_qc, obs, key)

```
integer,        intent(in)    :: num_copies
integer,        intent(in)    :: num_qc
type(obs_type), intent(inout) :: obs
integer,        intent(in)    :: key
```

Use standard input to create an observation. The number of values, number of qc fields, and an observation type-specific key associated with the observation are input. (Note that the key here is not the same as the key in an observation sequence.)

`num_copies`	Number of copies of data to be associated with observation
`num_qc`	Number of qc fields to be associated with observation
`obs`	Observation created via standard input
`key`	An observation type-specific key can be associated with each observation for use by the obs_def code.

call copy_obs(obs1, obs2)

```
type(obs_type), intent(out) :: obs1
type(obs_type), intent(in)  :: obs2
```

Copies the observation type obs2 to obs1. If the sizes of obs fields are not compatible, the space in obs1 is deallocated and reallocated with the appropriate size. This is overloaded to assignment(=).

obs1	Copy obs2 to here (destination)
obs2	Copy into obs1 (source)

call get_expected_obs_from_def_distrib_state(state_handle, ens_size, copy_indices, key, & obs_def, obs_kind_ind, state_time, isprior, assimilate_this_ob, evaluate_this_ob, expected_obs, & istatus)

```
type(ensemble_type), intent(in)  :: state_handle
integer,             intent(in)  :: ens_size
integer,             intent(in)  :: copy_indices(ens_size)
integer,             intent(in)  :: key
type(obs_def_type),  intent(in)  :: obs_def
integer,             intent(in)  :: obs_kind_ind
type(time_type),     intent(in)  :: state_time
logical,             intent(in)  :: isprior
integer,             intent(out) :: istatus(ens_size)
logical,             intent(out) :: assimilate_this_ob, evaluate_this_ob
real(r8),            intent(out) :: expected_obs(ens_size)
```

Used to compute the expected value of a set of observations in an observation sequence given a model state vector. Also returns a status variable that reports on problems taking forward operators. This version returns forward operator values for the entire ensemble in a single call.

state_handle	An observation sequence
keys	List of integer keys that specify observations in seq
ens_index	The ensemble number for this state vector
state	Model state vector
state_time	The time of the state data
obs_vals	Returned expected values of the observations
istatus	Integer error code for use in quality control (0 means no error)
assimilate_this_ob	Returns true if this observation type is being assimilated
evaluate_this_ob	Returns true if this observation type is being evaluated but not assimilated

call delete_seq_head(first_time, seq, all_gone)

```
type(time_type),         intent(in)    :: first_time
type(obs_sequence_type), intent(inout) :: seq
logical,                 intent(out)   :: all_gone
```

Deletes all observations in the sequence with times before first_time. If no observations remain, return all_gone as .true. If no observations fall into the time window (e.g. all before first_time or empty sequence to begin with), no deletions are done and all_gone is simply returned as .true.

`first_time`	Delete all observations with times before this
`seq`	An observation sequence
`all_gone`	Returns true if there are no valid observations remaining in the sequence after first_time

call delete_seq_tail(last_time, seq, all_gone)

```
type(time_type),         intent(in)     :: last_time
type(obs_sequence_type), intent(inout) :: seq
logical,                 intent(out)   :: all_gone
```

Deletes all observations in the sequence with times after last_time. If no observations remain, return all_gone as .true. If no observations fall into the time window (e.g. all after last_time or empty sequence to begin with), no deletions are done and all_gone is simply returned as .true.

`last_time`	Delete all observations with times after this
`seq`	An observation sequence
`all_gone`	Returns true if there are no valid observations remaining in the sequence before last_time

6.131.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_sequence_nml
   write_binary_obs_sequence = .false.
   read_binary_file_format   = 'native'
  /
```

Item	Type	Description
write_binary_obs_sequence	logical	If true, write binary obs_sequence files. If false, write ascii obs_sequence files.
read_binary_file_format	character(len=32)	The 'endian'ness of binary obs_sequence files. May be 'native' (endianness matches hardware default), 'big-endian', 'little-endian', and possibly 'cray'. Ignored if observation sequence files are ASCII.

6.131.5 Files

- obs_sequence_mod.nml in input.nml
- Files for reading and writing obs_sequences and obs specified in filter_nml.

6.131.6 References

- none

6.131.7 Private components

N/A

6.132 MODULE smoother_mod

6.132.1 Overview

Implements a fixed lag ensemble smoother as part of the filter. For now, this is done inefficiently with a separate call to `assim_tools_mod:filter_assim()` for each lag.

To enable the smoother, set the number of lags (num_lags) to something larger than 0 in the `smoother_nml` section of your `input.nml` file and run `filter` as before.

```
&smoother_nml
   num_lags              = 10,
   start_from_restart    = .false.,
   output_restart        = .true.,
   restart_in_file_name  = "ics",
   restart_out_file_name = "restart"  /
```

In the low order models, 10 is a plausible number.

In addition to generating `preassim.nc` and `analysis.nc` files, files of the form `Lag_NNNNN_Diag.nc` will be generated. Each of these has N fewer timesteps than the lag=0 run, starting at the same time but ending N timesteps sooner. The `obs_seq.final` file and the `preassim.nc` and `analysis.nc` files will be the same as the non-lagged version; the new output will be in each of the `Lag_NNNNN_Diag.nc` files.

6.132.2 Example

If you have a `true_state.nc` file and want to use the `plot_total_err` matlab function to plot the error, you must do the following steps to generate analogs of lagged `true_state.nc` files to use as a comparison. (The logic is not currently implemented in the matlab scripts to be able to compare netCDF files with unequal time coordinates.)

Make N separate versions of the true_state.nc with the last N timesteps removed. Using the netCDF NCO operator program ‘ncks’ is one way. If the true_state.nc file has 1000 time steps, then this command removes the last one:

ncks -d time,0,998 true_state.nc True_Lag01.nc

Note that the first time is at index 0, so the last timestep is index 999 in the full file, and 998 in the truncated file. Repeat this step for all N lags. Here are NCO commands to generate 10 truth files for num_lags = 10, 1000 time steps in true_state.nc:

ncks -d time,0,998 true_state.nc True_Lag01.nc ncks -d time,0,997 true_state.nc True_Lag02.nc ncks -d time,0,996 true_state.nc True_Lag03.nc ncks -d time,0,995 true_state.nc True_Lag04.nc ncks -d time,0,994 true_state.nc True_Lag05.nc ncks -d time,0,993 true_state.nc True_Lag06.nc ncks -d time,0,992 true_state.nc True_Lag07.nc ncks -d time,0,991 true_state.nc True_Lag08.nc ncks -d time,0,990 true_state.nc True_Lag09.nc ncks -d time,0,989 true_state.nc True_Lag10.nc

Here is an example matlab session which plots the lag=0 results and then odd numbered lags from 1 to 9. It uses the `plot_total_err` function from the $DART/matlab directory:

```
datadir    = '.';
truth_file = fullfile(datadir,'true_state.nc');
diagn_file = fullfile(datadir,'preassim.nc');
plot_total_err
reply = input('original data.  hit enter to continue ');

truth_file = fullfile(datadir,'True_Lag01.nc');
diagn_file = fullfile(datadir,'Lag_00001_Diag.nc');
plot_total_err
reply = input('Lag 01.  hit enter to continue ');

truth_file = fullfile(datadir,'True_Lag03.nc');
diagn_file = fullfile(datadir,'Lag_00003_Diag.nc');
plot_total_err
reply = input('Lag 03.  hit enter to continue ');

truth_file = fullfile(datadir,'True_Lag05.nc');
diagn_file = fullfile(datadir,'Lag_00005_Diag.nc');
plot_total_err
reply = input('Lag 05.  hit enter to continue ');

truth_file = fullfile(datadir,'True_Lag07.nc');
diagn_file = fullfile(datadir,'Lag_00007_Diag.nc');
plot_total_err
reply = input('Lag 07.  hit enter to continue ');

truth_file = fullfile(datadir,'True_Lag09.nc');
diagn_file = fullfile(datadir,'Lag_00009_Diag.nc');
plot_total_err
reply = input('Lag 09.  hit enter to continue ');
```

6.132.3 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand ‘&’ and terminate with a slash ‘/’. Character strings that contain a ‘/’ must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&smoother_nml
   num_lags              = 0,
   start_from_restart    = .false.,
   output_restart        = .false.,
   restart_in_file_name  = 'ics',
   restart_out_file_name = 'restart'
/
```

Item	Type	Description
num_lags	integer	Number of smoother lags; < 1 means no smoother.
start_from_restart	logical	True if smoother states are to come from restart file(s). False if they are to be spun up from scratch.
output_restart	logical	True if restart file(s) are to be written, else false.
restart_in_file_name	character(len=129)	String used to construct the file name from which to read restart data. `Lag_NNNNN_` will be prepended to the specified value to create the actual filename. If each ensemble is to be read from a separate file, the .NNNN ensemble number will also be appended. e.g. specifying 'ics' here results in 'Lag_00001_ics' if all ensemble members are read from a single file, 'Lag_00001_ics.0001', 'Lag_00001_ics.0002', etc for multiples.
restart_out_file_name	character(len=129)	String used to construct the file name to which to write restart data. `Lag_NNNNN_` will be prepended to the specified value to create the actual filename. If each ensemble is to be written to a separate file, the .NNNN ensemble number will also be appended. e.g. specifying 'restart' here results in 'Lag_00001_restart' if all ensemble members are written to a single file, 'Lag_00001_restart.0001', 'Lag_00001_restart.0002', etc for multiples.

6.132.4 Other modules used

```
types_mod
mpi_utilities_mod
utilities_mod
ensemble_manager_mod
time_manager_mod
assim_model_mod
assim_tools_mod
obs_sequence_mod
adaptive_inflate_mod
```

6.132.5 Public interfaces

use smoother_mod, only :	smoother_read_restart
	advance_smoother
	smoother_gen_copy_meta_data
	smoother_write_restart
	init_smoother
	do_smoothing
	smoother_mean_spread
	smoother_assim
	filter_state_space_diagnostics
	smoother_ss_diagnostics
	smoother_end

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

call smoother_read_restart(ens_handle, ens_size, model_size, time1, init_time_days)

```
type(ensemble_type), intent(inout) :: ens_handle
integer, intent(in)                :: ens_size
integer, intent(in)                :: model_size
type(time_type), intent(inout)     :: time1
integer, intent(in)                :: init_time_days
```

Reads in ensemble of states for all lag estimates from a restart file.

`ens_handle`	Handle of ensemble manager structure of single state; copied into all lags for startup.
`ens_size`	Size of the ensemble.
`model_size`	Size of the model state vector.
`time1`	Overwrite the time in the restart file with this value if init_time_days is non-negative.
`init_time_days`	If non-negative, use time1 instead of time in restart file.

call advance_smoother(ens_handle)

```
type(ensemble_type), intent(in) :: ens_handle
```

Advances smoother state estimates at all lags forward in time. This entails copying the most recent smoother state, contained in ens_handle, into the lag 1 smoother state and pushing back all other lags by 1 (i.e. lag 1 becomes lag 2, etc.).

`ens_handle`	Ensemble handle with most recent filtered state.

call smoother_gen_copy_meta_data(num_output_state_members, output_inflation)

```
integer, intent(in) :: num_output_state_members
logical, intent(in) :: output_inflation
```

Initializes the metadata required for the smoother state space diagnostic files.

`num_output_state_members`	Number of copies of smoother state vector that should be in state space diagnostic output.
`output_inflation`	True if smoother state space output should include inflation values.

call smoother_write_restart(start_copy, end_copy)

```
integer, intent(in) :: start_copy
integer, intent(in) :: end_copy
```

Outputs restart files for all lags of smoother state. Integer arguments specify the start and end global indices of a continguous set of copies that contain the ensemble members.

`start_copy`	Global index of ensemble copy that starts the actual ensemble members for smoother.
`end_copy`	Global index of ensemble copy that ends the actual ensemble members for smoother.

call init_smoother(ens_handle, POST_INF_COPY, POST_INF_SD_COPY)

```
type(ensemble_type), intent(inout) :: ens_handle
integer, intent(in)                :: POST_INF_COPY
integer, intent(in)                :: POST_INF_SD_COPY
```

Initializes the storage needed for a smoother. Also initializes an adaptive inflation type that does NO inflation (not currently supported for smoothers).

ens_handle	An ensemble handle for the filter that contains information about ensemble and model size.
POST_INF_COPY	Global index of ensemble copy that holds posterior state space inflation values.
POST_INF_SD_COPY	Global index of ensemble copy that holds posterior inflation standard deviation values.

var = do_smoothing()

```
logical, intent(out) :: do_smoothing
```

Returns true if smoothing is to be done, else false.

do_smoothing	Returns true if smoothing is to be done.

call smoother_mean_spread(ens_size,ENS_MEAN_COPY,ENS_SD_COPY, output_state_ens_mean,output_state_ens_spread)

```
integer, intent(in) :: ens_size
integer, intent(in) :: ENS_MEAN_COPY
integer, intent(in) :: ENS_SD_COPY
logical, intent(in) :: output_state_ens_mean
logical, intent(in) :: output_state_ens_spread
```

Computes the ensemble mean (and spread if required) of all state variables for all lagged ensembles. Spread is only computed if it is required for output.

ens_size	Size of ensemble.
ENS_MEAN_COPY	Global index of copy that stores ensemble mean.
ENS_SD_COPY	Global index of copy that stores ensemble spread.
output_state_ens_mean	True if the ensemble mean is to be output to state diagnostic file.
output_state_ens_spread	True if ensemble spread is to be output to state diagnostic file.

call smoother_assim(obs_ens_handle, seq, keys, ens_size, num_groups, obs_val_index, ENS_MEAN_COPY, ENS_SD_COPY, PRIOR_INF_COPY, PRIOR_INF_SD_COPY, OBS_KEY_COPY, OBS_GLOBAL_QC_COPY, OBS_PRIOR_MEAN_START, OBS_PRIOR_MEAN_END, OBS_PRIOR_VAR_START, OBS_PRIOR_VAR_END)

```
type(ensemble_type), intent(inout)  :: obs_ens_handle
type(obs_sequence_type), intent(in) :: seq
integer, dimension(:), intent(in)   :: keys
integer, intent(in)                 :: ens_size
integer, intent(in)                 :: num_groups
integer, intent(in)                 :: obs_val_index
```

(continues on next page)

(continued from previous page)

```
integer, intent(in)                    :: ENS_MEAN_COPY
integer, intent(in)                    :: ENS_SD_COPY
integer, intent(in)                    :: PRIOR_INF_COPY
integer, intent(in)                    :: PRIOR_INF_SD_COPY
integer, intent(in)                    :: OBS_KEY_COPY
integer, intent(in)                    :: OBS_GLOBAL_QC_COPY
integer, intent(in)                    :: OBS_PRIOR_MEAN_START
integer, intent(in)                    :: OBS_PRIOR_MEAN_END
integer, intent(in)                    :: OBS_PRIOR_VAR_START
integer, intent(in)                    :: OBS_PRIOR_VAR_END
```

Does assimilation of a set of observations for each smoother lag.

obs_ens_handle	Handle for ensemble manager holding prior estimates of observations.
seq	Observation sequence being assimilated.
keys	A one dimensional array containing indices in seq of observations to as similate at current time.
ens_size	Ensemble size.
num_groups	Number of groups in filter.
obs_val_index	Integer index of copy of data in seq that contains the observed value from instruments.
ENS_MEAN_COPY	Global index in smoother's state ensemble that holds ensemble mean.
ENS_SD_COPY	Global index in smoother's state ensemble that holds ensemble standard deviation.
PRIOR_INF_COPY	Global index in obs_ens_handle that holds inflation values (not used for smoother).
PRIOR_INF_SD_COPY	Global index in obs_ens_handle that holds inflation sd values (not used for smoother).
OBS_KEY_COPY	Global index in obs_ens_handle that holds the key for the observation.
OBS_GLOBAL_QC_COPY	Global index in obs_ens_handle that holds the quality control value.
OBS_PRIOR_MEAN_START	Global index in obs_ens_handle that holds the first group's prior mean.
OBS_PRIOR_MEAN_END	Global index in obs_ens_handle that holds the last group's prior mean.
OBS_PRIOR_VAR_START	Global index in obs_ens_handle that holds the first group's prior variance.
OBS_PRIOR_VAR_END	Global index in obs_ens_handle that holds the last group's prior variance.

call filter_state_space_diagnostics(out_unit, ens_handle, model_size, num_output_state_members, output_state_mean_index, output_state_spread_index, output_inflation, temp_ens, ENS_MEAN_COPY, ENS_SD_COPY, inflate, INF_COPY, INF_SD_COPY)

```
type(netcdf_file_type), intent(inout)   :: out_unit
type(ensemble_type), intent(inout)      :: ens_handle
integer, intent(in)                     :: model_size
integer, intent(in)                     :: num_output_state_members
integer, intent(in)                     :: output_state_mean_index
integer, intent(in)                     :: output_state_spread_index
logical, intent(in)                     :: output_inflation
real(r8), intent(out)                   :: temp_ens(model_size)
integer, intent(in)                     :: ENS_MEAN_COPY
integer, intent(in)                     :: ENS_SD_COPY
type(adaptive_inflate_type), intent(in) :: inflate
integer, intent(in)                     :: INF_COPY
integer, intent(in)                     :: INF_SD_COPY
```

Writes state space diagnostic values including ensemble members, mean and spread, and inflation mean and spread to a netcdf file.

out_unit	Descriptor for the netcdf file being written.
ens_handle	Ensemble handle whose state space values are to be written.
model_size	Size of the model state vector.
num_output_state_members	Number of individual state members to be output.
output_state_mean_index	Index in netcdf file for ensemble mean.
output_state_spread_index	Index in netcdf file for ensemble spread.
output_inflation	True if the inflation values are to be output. Default is .TRUE.
temp_ens	Storage passed in to avoid having to allocate extra space.
ENS_MEAN_COPY	Global index in ens_handle for ensemble mean.
ENS_SD_COPY	Global index in ens_handle for ensemble spread.
inflate	Contains description and values of state space inflation.
INF_COPY	Global index in ens_handle of inflation values.
INF_SD_COPY	Global index in ens_handle of inflation standard deviation values.

call smoother_ss_diagnostics(model_size, num_output_state_members, output_inflation, temp_ens, ENS_MEAN_COPY, ENS_SD_COPY, POST_INF_COPY, POST_INF_SD_COPY)

```
integer, intent(in)   :: model_size
integer, intent(in)   :: num_output_state_members
logical, intent(in)   :: output_inflation
real(r8), intent(out) :: temp_ens(model_size)
integer, intent(in)   :: ENS_MEAN_COPY
integer, intent(in)   :: ENS_SD_COPY
integer, intent(in)   :: POST_INF_COPY
integer, intent(in)   :: POST_INF_SD_COPY
```

Outputs state space diagnostics files for all smoother lags.

model_size	Size of the model state vector.
num_output_state_members	Number of state copies to be output in the state space diagnostics file.
output_inflation	True if the inflation values are to be output. Default is .TRUE.
temp_ens	Storage passed in to avoid having to allocate extra space.
ENS_MEAN_COPY	Global index of the ensemble mean in the lag smoother ensemble handles.
ENS_SD_COPY	Global index of the ensemble spread in the lag smoother ensemble handles.
POST_INF_COPY	Global index of the inflation value in the lag smoother ensemble handles (not currently used).
POST_INF_SD_COPY	Global index of the inflation spread in the lag smoother ensemble handles (not currently used).

call smoother_end()

Releases storage allocated for smoother.

call smoother_inc_lags()

Increments the number of lags that are in use for smoother. Used when a smoother is being started up and there have not been enough times to propagate the state to all requested lags.

6.132.6 Files

- input.nml
- smoother initial condition files
- smoother restart files

6.132.7 References

1. none

6.132.8 Private components

N/A

6.133 MODULE assim_model_mod

6.133.1 Overview

This module acts as an intermediary between DART compliant models and the filter. At one time the assim_model_type, which combines a state vector and a time_type, was envisioned as being fundamental to how DART views model states. This paradigm is gradually being abandoned so that model state vectors and times are handled as separate data types. It is important to call static_init_assim_model before using routines in assim_model_mod. Interfaces to work with model time stepping, restart files, and computations about the locations of model state variables and the distance between observations and state variables. Many of the interfaces are passed through nearly directly to the model_mod.

Notes

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

6.133.2 Namelist

This module does not have a namelist.

6.133.3 Other modules used

```
types_mod
location_mod (model dependent choice)
time_manager_mod
utilities_mod
model_mod
netcdf
typeSizes (part of netcdf)
```

6.133.4 Public interfaces

use assim_model_mod, only :	
	adv_1step
	aoutput_diagnostics
	aread_state_restart
	assim_model_type
	awrite_state_restart
	close_restart
	copy_assim_model
	end_assim_model
	ens_mean_for_model
	finalize_diag_output
	get_close_maxdist_init
	get_close_obs
	get_close_obs_init
	get_closest_state_time_to
	get_diag_input_copy_meta_data
	get_initial_condition
	get_model_size
	get_model_state_vector
	get_model_time
	get_model_time_step
	get_state_meta_data
	init_assim_model
	init_diag_input
	init_diag_output
	input_diagnostics
	interpolate
	nc_append_time
	nc_get_tindex
	nc_write_calendar_atts
	netcdf_file_type
	open_restart_read
	open_restart_write

continues on next page

Table 8 – continued from previous page

	output_diagnostics
	pert_model_state
	read_state_restart
	set_model_state_vector
	set_model_time
	static_init_assim_model
	write_state_restart

```
type assim_model_type
   private
   real(r8), pointer   :: state_vector(:)
   type(time_type)     :: time
   integer             :: model_size
   integer             :: copyID
end type assim_model_type
```

This type is used to represent both the state and time of a state from a model.

Component	Description
state_vector	A one dimensional representation of the model state vector.
time	The time of the model state.
model_s	Size of the model state vector.
copyID	Not used in present implementation.

```
type netcdf_file_type
   integer             :: ncid
   integer             :: Ntimes
   integer             :: NtimesMAX
   real(r8), pointer   :: rtimes(:)
   type(time_type), pointer :: times(:)
   character(len = 80)      :: fname
end type netcdf_file_type
```

Basically, we want to keep a local mirror of the unlimited dimension coordinate variable (i.e. time) because dynamically querying it causes unacceptable performance degradation over “long” integrations.

Component	Description
ncid	The netcdf file unit id.
Ntimes	The current working length.
NtimesMAX	Allocated length.
rtimes	Times as real (r8).
times	Times as time_types.
fname	Netcdf file name.

call static_init_assim_model()

Initializes the assim_model class. Must be called before any other assim_model_mod interfaces are used. Also calls the static initialization for the underlying model. There are no arguments.

ncFileID = init_diag_output(FileName, global_meta_data, copies_of_field_per_time, meta_data_per_copy [, lagID])

```
type(netcdf_file_type)              :: init_diag_output
character (len = *), intent(in)  :: FileName
character (len = *), intent(in)  :: global_meta_data
integer, intent(in)                 :: copies_of_field_per_time
character (len = *), intent(in)  :: meta_data_per_copy(copies_of_field_per_time)
integer, optional, intent(in)     :: lagID
```

Initializes a netCDF file for output of state space diagnostics. A handle to the channel on which the file is opened is returned.

`ncFileID`	Identifier for the netcdf file is returned. This is not an integer unit number, but a derived type containing additional information about the opened file.
`FileName`	Name of file to open.
`global_meta_data`	Global metadata that describes the contents of this file.
`copies_of_field_per_time`	Number of copies of data to be written at each time. For instance, these could be the prior ensemble members, prior ensemble mean, prior ensemble spread, posterior ensemble members, posterior spread and mean, etc..
`meta_data_per_copy`	Metadata describing each of the copies.
lagID	If using the smoother, which lag number this output is for.

var = get_model_size()

```
integer :: get_model_size
```

Returns the size of the model state vector. This is a direct pass through to the model_mod.

var = get_closest_state_time_to(model_time, time)

```
type(time_type)              ::  get_closest_state_time_to
type(time_type), intent(in)  ::  model_time
type(time_type), intent(in)  ::  time
```

Returns the closest time that a model is capable of advancing a given state to a specified time. For instance, what is the closest time to 12GMT 01 January, 2004 that a model state at 00GMT 01 January, 2004 can be advanced? If the model time is past the time, the model time is returned (new feature in releases after Hawaii).

`var`	The closest time to which the model can be advanced is returned.
`model_time`	The time of a model state vector.
`time`	A time that one would like to get close to with the model.

call get_state_meta_data()

Pass through to model_mod. See model_mod documentation for arguments and description.

var = get_model_time(assim_model)

```
type(time_type)                        :: get_model_time
type(assim_model_type), intent(in)  :: assim_model
```

Returns time from an assim_model type.

`var`	Returned time from assim_model
`assim_model`	Assim_model type from which to extract time

var = get_model_state_vector(assim_model)

```
real(r8)                               :: get_model_state_vector(model_size)
type(assim_model_type), intent(in)  :: assim_model
```

Returns the state vector component from an assim_model_type.

`var`	Returned state vector
`assim_model`	Input assim_model_type

call copy_assim_model(model_out, model_in)

```
type(assim_model_type), intent(out) :: model_out
type(assim_model_type), intent(in)  :: model_in
```

Copies one assim_model_type to another.

`model_out`	Copy.
`model_in`	Data to be copied.

call interpolate(x, location, loctype, obs_vals, istatus)

```
real(r8),            intent(in)  :: x(:)
type(location_type), intent(in)  :: location
integer,             intent(in)  :: loctype
real(r8),            intent(out) :: obs_vals
integer,             intent(out) :: istatus
```

Interpolates a given model state variable type to a location given the model state vector. Nearly direct call to model_interpolate in model_mod. See model_mod for the error return values in istatus.

`x`	Model state vector.
`location`	Location to which to interpolate.
`loctype`	Type of variable to interpolate.
`obs_vals`	Returned interpolated value.
`istatus`	Returned as 0 if all is well, else various errors.

call set_model_time(assim_model, time)

```
type(assim_model_type), intent(inout) :: assim_model
type(time_type), intent(in)           :: time
```

Sets the time in an assim_model_type.

`assim_model`	Set the time in this assim_model_type.
`time`	Set to this time

call set_model_state_vector(assim_model, state)

```
type(assim_model_type), intent(inout) :: assim_model
real(r8), intent(in)                  :: state(:)
```

Set the state in an assim_model_type.

`assim_model`	Set the state vector in this assim_model_type.
`state`	The state vector to be inserted.

call write_state_restart(assim_model, funit [, target_time])

```
type(assim_model_type),    intent(in) :: assim_model
integer,                   intent(in) :: funit
type(time_type), optional, intent(in) :: target_time
```

Writes a restart from an assim_model_type with an optional target_time.

`assim_model`	Write a restart from this assim_model_type.
`funit`	Integer file unit id open for output of restart files.
target_time	If present, put this target time at the front of the restart file.

call read_state_restart(assim_model, funit [, target_time])

```
type(assim_model_type),    intent(out) :: assim_model
integer,                   intent(in)  :: funit
type(time_type), optional, intent(out) :: target_time
```

Read a state restart file into assim_model_type. Optionally read a prepended target time.

`assim_model`	Read the time and state vector from restart into this.
`funit`	File id that has been opened for reading restart files.
target_time	If present, read a target time from the front of the file into this.

call output_diagnostics(ndFileID, state [, copy_index])

```
type(netcdf_file_type), intent(inout) :: ndFileID
type(assim_model_type), intent(in)    :: state
integer, optional,      intent(in)    :: copy_index
```

Writes one copy of the state time and vector to a netCDF file.

`ndFileID`	An identifier for a netCDF file
`state`	State vector and time
copy_index	Which copy of state is to be output

call end_assim_model()

Called to clean-up at end of assim_model use. For now just passes through to model_mod.

call input_diagnostics(file_id, state, copy_index)

```
integer,                intent(in)    :: file_id
type(assim_model_type), intent(inout) :: state
integer,                intent(out)   :: copy_index
```

Used to read in a particular copy of the state vector from an open state diagnostics file.

`file_id`	Integer descriptor (channel number) for a diagnostics file being read.
`state`	Assim_model_type to read in data.
`copy_index`	Which copy of state to be read.

var = init_diag_input(file_name, global_meta_data, model_size, copies_of_field_per_time)

```
integer                           :: init_diag_input
character(len=*), intent(in)  :: file_name
character(len=*), intent(out) :: global_meta_data
integer,          intent(out) :: model_size
integer,          intent(out) :: copies_of_field_per_time
```

Opens a state diagnostic file and reads the global meta data, model size, and number of data copies.

`var`	Returns the unit number on which the file is open.
`file_name`	File name of state diagnostic file.
`global_meta_data`	Global metadata string from file.
`model_size`	Size of model.
`copies_of_field_per_time`	Number of copies of the state vector at each time.

call init_assim_model(state)

```
type(assim_model_type), intent(inout) :: state
```

Creates storage for an assim_model_type.

`state`	An assim_model_type that needs storage created.

call get_diag_input_copy_meta_data(file_id, model_size_out, num_copies, location, meta_data_per_copy)

```
integer,             intent(in)  :: file_id
integer,             intent(in)  :: model_size_out
integer,             intent(in)  :: num_copies
type(location_type), intent(out) :: location(model_size_out)
character(len = *)               :: meta_data_per_copy(num_copies)
```

Reads meta-data describing state vectors in a state diagnostics file. Given the file, the model_size, and the number of copies, returns the locations of each state variable and the text description of each copy.

`file_id`	Integer channel open to state diagostic file being read
`Model_size_out`	model size
`num_copies`	Number of copies of state in file
`location`	Returned locations for state vector
`meta_data_per_copy`	Meta data describing what is in each copy of state vector

var = finalize_diag_output(ncFileID)

```
integer                                 :: finalize_diag_output
type(netcdf_file_type), intent(inout) :: ncFileID
```

Used to complete writing on and open netcdf file. An error return is provided for passing to the netcdf error handling routines.

`var`	Returns an error value.
`ncFileID`	Netcdf file id of an open file.

call aread_state_restart(model_time, model_state, funit [, target_time])

```
type(time_type),           intent(out) :: model_time
real(r8),                  intent(out) :: model_state(:)
integer,                   intent(in)  :: funit
type(time_type), optional, intent(out) :: target_time
```

Reads a model time and state, and optionally a prepended target time, from a state restart file.

`model_time`	Returned time of model state
`model_state`	Returned model state.
`funit`	Channel open for reading a state restart file.
target_time	If present, this time is read from the front of the restart file.

call aoutput_diagnostics(ncFileID, model_time, model_state [, copy_index])

```
type(netcdf_file_type), intent(inout) :: ncFileID
type(time_type),        intent(in)    :: model_time
real(r8),               intent(in)    :: model_state(:)
integer, optional,      intent(in)    :: copy_index
```

Write a state vector to a state diagnostics netcdf file.

`ncFileID`	Unit for a state vector netcdf file open for output.
`model_time`	The time of the state to be output
`model_state`	A model state vector to be output.
copy_index	Which copy of state vector is to be written, default is copy 1

call awrite_state_restart(model_time, model_state, funit [, target_time])

```
type(time_type),           intent(in) :: model_time
real(r8),                  intent(in) :: model_state(:)
integer,                   intent(in) :: funit
type(time_type), optional, intent(in) :: target_time
```

Writes a model time and state vector to a restart file and optionally prepends a target time.

`model_time`	Time of model state.
`model_state`	Model state vector.
`funit`	Channel of file open for restart output.
target_time	If present, time to be prepended to state time / vector.

call pert_model_state()

Passes through to pert_model_state in model_mod. See model_mod documentation for arguments and details.

var = nc_append_time(ncFileID, time)

```
integer                                       :: nc_append_time
type(netcdf_file_type), intent(inout) :: ncFileID
type(time_type),        intent(in)    :: time
```

Appends the time to the time coordinate variable of the netcdf file. The new length of the time variable is returned. Requires that time is a coordinate variable AND it is the unlimited dimension.

`var`	Returns new length of time variable.
`ncFileID`	Points to open netcdf file.
`time`	The next time to be added to the file.

var = nc_write_calendar_atts(ncFileID, TimeVarID)

```
integer                                :: nc_write_calendar_atts
type(netcdf_file_type), intent(in)  :: ncFileID
integer,                intent(in)  :: TimeVarID
```

Sets up the metadata for the appropriate calendar being used in the time manager an writes it to a netcdf file.

`var`	Returns a netcdf error code.
`ncFileID`	Netcdf file id pointing to a file open for writing.
`TimeVarID`	The index of the time variable in the netcdf file.

var = nc_get_tindex(ncFileID, statetime)

```
integer                                   :: nc_get_tindex
type(netcdf_file_type), intent(inout) :: ncFileID
type(time_type),        intent(in)    :: statetime
```

Returns the index of a time from the time variable in a netcdf file. This function has been replaced with more efficient approaches and may be deleted from future releases.

`var`	The index of the time in the netcdf file.
`ncFileID`	File id for an open netcdf file.
`statetime`	The time to be found in the netcdf file.

var = get_model_time_step()

```
type(time_type) :: get_model_time_step
```

This passes through to model_mod. See model_mod documentation for arguments and details.

`var`	Returns time step of model.

var = open_restart_read(file_name)

```
integer                          :: open_restart_read
character(len=*), intent(in) :: file_name
```

Opens a restart file for readig.

var	Returns a file descriptor (channel number).
file_name	Name of restart file to be open for reading.

var = open_restart_write(file_name)

```
integer                          :: open_restart_write
character(len=*), intent(in)     :: file_name
```

Open a restart file for writing.

var	Returns a file descriptor (channel) for a restart file.
file_name	File name of restart file to be opened.

call close_restart(file_unit)

```
integer, intent(in) :: file_unit
```

Closes a restart file.

file_unit	File descriptor (channel number) of open restart file.

call adv_1step()

Advances a model by one step. Pass through to model_mod. See model_mod documentation for arguments and details.

call get_initial_condition(time, x)

```
type(time_type), intent(out) :: time
real(r8),        intent(out) :: x
```

Obtains an initial condition from models that support this option.

time	the valid time of the model state
x	the initial model state

call ens_mean_for_model(ens_mean)

```
type(r8), intent(in) :: ens_mean(:)
```

An array of length model_size containing the ensemble means. This is a direct pass through to the model_mod.

ens_mean	Array of length model_size containing the mean for each entry in the state vector.

call get_close_maxdist_init(gc, maxdist)

```
type(get_close_type), intent(inout) :: gc
type(r8), intent(in)                :: maxdist
```

Sets the threshold distance. Anything closer than this is deemed to be close. This is a direct pass through to the model_mod, which in turn can pass through to the location_mod.

gc	Data for efficiently finding close locations.
maxdist	Anything closer than this distance is a close location.

call get_close_obs(gc, base_obs_loc, base_obs_kind, obs, obs_kind, num_close, close_ind [, dist])

```
type(get_close_type), intent(in)  :: gc
type(location_type),  intent(in)  :: base_obs_loc
integer,              intent(in)  :: base_obs_kind
type(location_type),  intent(in)  :: obs(:)
integer,              intent(in)  :: obs_kind(:)
integer,              intent(out) :: num_close
integer,              intent(out) :: close_ind(:)
real(r8),  optional,  intent(out) :: dist(:)
```

Given a single location and a list of other locations, returns the indices of all the locations close to the single one along with the number of these and the distances for the close ones. The observation kinds are passed in to allow more sophisticated distance computations to be done if needed. This is a direct pass through to the model_mod, which in turn can pass through to the location_mod.

gc	Data for efficiently finding close locations.
base_obs_loc	Single given location.
base_obs_kind	Kind of the single location.
obs	List of observations from which close ones are to be found.
obs_kind	Kind associated with observations in obs list.
num_close	Number of observations close to the given location.
close_ind	Indices of those locations that are close.
dist	Distance between given location and the close ones identified in close_ind.

call get_close_obs_init(gc, num, obs)

```
type(get_close_type), intent(inout) :: gc
integer,              intent(in)    :: num
type(location_type),  intent(in)    :: obs(:)
```

Initialize storage for efficient identification of locations close to a given location. Allocates storage for keeping track of which 'box' each observation in the list is in. This is a direct pass through to the model_mod, which in turn can pass through to the location_mod.

gc	Data for efficiently finding close locations.
num	The number of locations in the list.
obs	The location of each element in the list, not used in 1D implementation.

6.133.5 Files

filename	purpose/comment
filter_restart	specified in &filter_nml:restart_in_filename
filter_restart	specified in &filter_nml:restart_out_filename
input.nml	to read namelists

6.133.6 References

- none

6.133.7 Private components

N/A

6.134 MODULE assim_tools_mod

6.134.1 Overview

This module provides subroutines that implement the parallel versions of the sequential scalar filter algorithms. These include the standard sequential filter as described in Anderson 2001, 2003 along with systematic correction algorithms for both mean and spread. In addition, algorithms to do a variety of flavors of filters including the EAKF, ENKF, particle filter, and kernel filters are included. The parallel implementation that allows each observation to update all state variables that are close to it at the same time is described in Anderson and Collins, 2007.

6.134.2 Filter types

Available observation space filter types include:

- 1 = EAKF (Ensemble Adjustment Kalman Filter, see Anderson 2001)
- 2 = ENKF (Ensemble Kalman Filter)
- 3 = Kernel filter
- 4 = Observation Space Particle filter
- 5 = Random draw from posterior (contact dart@ucar.edu before using)
- 6 = Deterministic draw from posterior with fixed kurtosis (ditto)
- 7 = Boxcar kernel filter
- 8 = Rank Histogram filter (see Anderson 2010)
- 9 = Particle filter (see Poterjoy 2016)

We recommend using type=1, the EAKF. Note that although the algorithm is expressed in a slightly different form, the EAKF is identical to the EnSRF (Ensemble Square Root Filter) described by Whitaker and Hamill in 2002. Highly non-gaussian distributions may get better results from type=8, Rank Histogram filter.

6.134.3 Localization

Localization controls how far the impact of an observation extends. The namelist items related to localization are spread over several different individual namelists, so we have made a single collected description of them here along with some guidance on setting the values.

This discussion centers on the mechanics of how you control localization in DART with the namelist items, and a little bit about pragmatic approaches to picking the values. There is no discussion about the theory behind localization - contact Jeff Anderson for more details. Additionally, the discussion here applies specifically to models using the 3d-sphere location module. The same process takes place in 1d models but the details of the location module namelist is different.

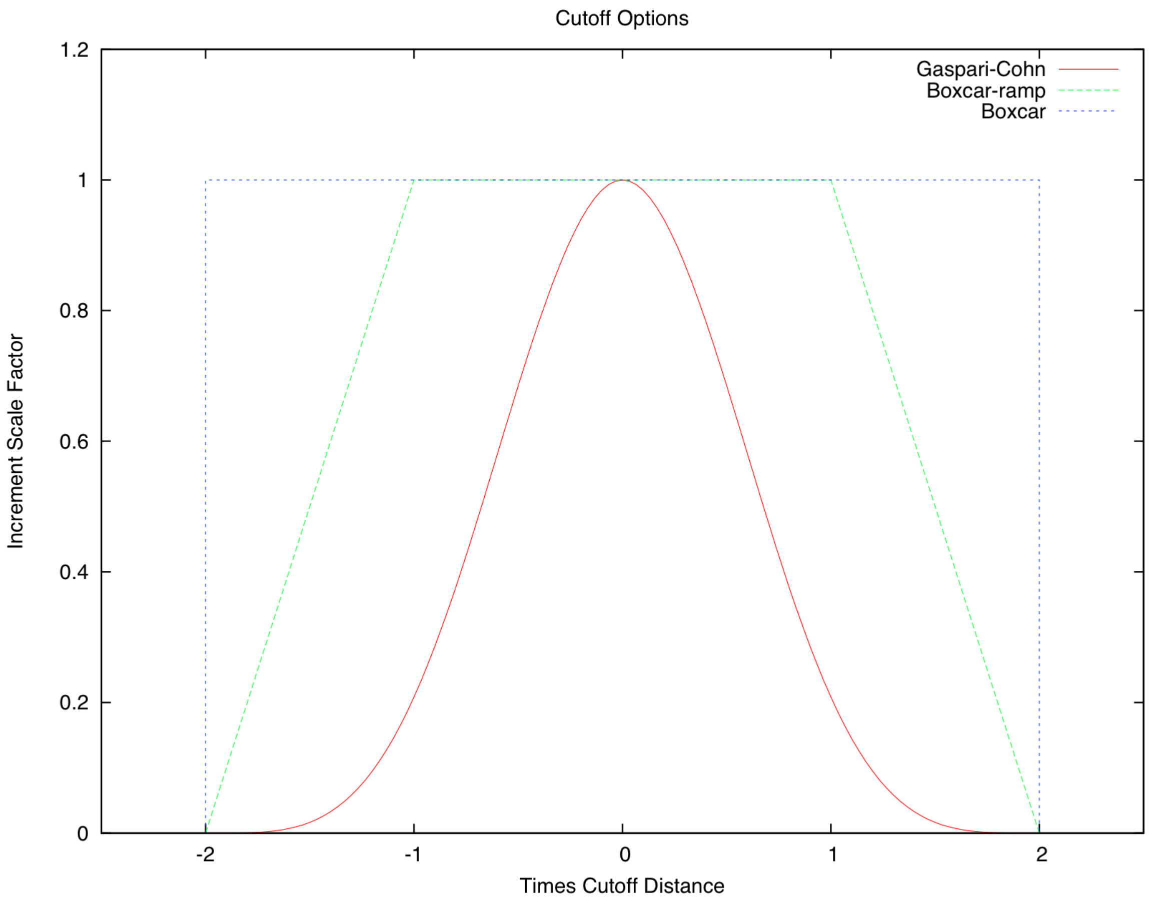
The following namelist items related to 3d-sphere localization are all found in the `input.nml` file:

`&assim_tools_nml :: cutoff` *valid values:* 0.0 to infinity

> This is the value, in radians, of the half-width of the localization radius (this follows the terminology of an early paper on localization). For each observation, a state vector item increment is computed based on the covariance values. Then a multiplier, based on the 'select_localization' setting (see below) decreases the increment as the distance between the obs and the state vector item increases. In all cases if the distance exceeds 2*cutoff, the increment is 0.

`&cov_cutoff_nml :: select_localization` *valid values:* 1=Gaspari-Cohn; 2=Boxcar; 3=Ramped Boxcar

> Controls the shape of the multiplier function applied to the computed increment as the distance increases between the obs and the state vector item. Most users use type 1 localization.
>
> - Type 1 (Gaspari-Cohn) has a value of 1 at 0 distance, 0 at 2*cutoff, and decreases in an approximation of a gaussian in between.
> - Type 2 (Boxcar) is 1 from 0 to 2*cutoff, and then 0 beyond.
> - Type 3 (Ramped Boxcar) is 1 to cutoff and then ramps linearly down to 0 at 2*cutoff.



&location_nml :: horiz_dist_only *valid values:* .true., .false.

If set to .true., then the vertical location of all items, observations and state vector both, are ignored when computing distances between pairs of locations. This has the effect that all items within a vertical-cylindrical area are considered the same distance away.

If set to .false., then the full 3d separation is computed. Since the localization is computed in radians, the 2d distance is easy to compute but a scaling factor must be given for the vertical since vertical coordinates can be in meters, pressure, or model levels. See below for the 'vert_normalization_xxx' namelist items.

&location_nml :: vert_normalization_{pressure,height,level,scale_height} *valid values:* real numbers, in pascals, meters, index, and value respectively

If 'horiz_dist_only' is set to .true., these are ignored. If set to .false., these are required. They are the amount of that quantity that is equivalent to 1 radian in the horizontal. If the model is an earth-based one, then one radian is roughly 6366 kilometers, so if vert_normalization_height is set to 6366000 meters, then the localization cutoff will be a perfect sphere. If you want to localize over a larger distance in the vertical than horizontal, use a larger value. If you want to localize more sharply in the vertical, use a smaller number. The type of localization used is set by which type of vertical coordinate the observations and state vector items have.

If you have observations with different vertical coordinates (e.g. pressure and height), or if your observations have a different vertical coordinate than your state vector items, or if you want to localize in a different type of unit than your normal vertical coordinate (e.g. your model uses pressure in the vertical but you wish to localize in meters), then you will need to modify or add a `get_close()` routine in your `model_mod.f90` file. See the discussion in the *MODULE location_mod (threed_sphere)* documentation for how to transform vertical coordinates before localization.

&assim_tools_nml ::adaptive_localization_threshold *valid values:* integer counts, or -1 to disable

Used to dynamically shrink the localization cutoff in areas of dense observations. If set to something larger than 0, first the number of other observations within 2*cutoff is computed. If it is larger than this given threshold, the cutoff is decreased proportionally so if the observations were evenly distributed in space, the number of observations within 2*revised_cutoff would now be the threshold value. The cutoff value is computed for each observation as it is assimilated, so can be different for each one.

&assim_tools_nml :: adaptive_cutoff_floor *valid values:* 0.0 to infinity, or -1 to disable

If using adaptive localization (adaptive_localization_threshold set to a value greater than 0), then this value can be used to set a minimum cutoff distance below which the adaptive code will not shrink. Set to -1 to disable. Ignored if not using adaptive localization.

&assim_tools_nml :: output_localization_diagnostics *valid values:* .true., .false.

If .true. and if adaptive localization is on, a single text line is printed to a file giving the original cutoff and number of observations, and the revised cutoff and new number of counts within this smaller cutoff for any observation which has nearby observations which exceed the adaptive threshold count.

&assim_tools_nml :: localization_diagnostics_file *valid values:* text string

Name of the file where the adaptive localization diagnostic information is written.

&assim_tools_nml :: special_localization_obs_types *valid values:* list of 1 or more text strings

The cutoff localization setting is less critical in DART than it might be in other situations since during the assimilation DART computes the covariances between observations and nearby state vector locations and that is the major factor in controlling the impact an observation has. For conventional observations fine-tuning the cutoff based on observation type is not recommended (it is possible to do more harm than good with it). But in certain special cases there may be valid reasons to want to change the localization cutoff distances drastically for certain kinds of observations. This and the following namelist items allow this.

Optional list of observation types (e.g. “RADAR_REFLECTIVITY”, “AIRS_TEMPERATURE”) which will use a different cutoff distance. Any observation types not listed here will use the standard cutoff distance (set by the ‘cutoff’ namelist value). This is only implemented for the threed_sphere location module (the one used by most geophysical models.)

&assim_tools_nml :: special_localization_cutoffs *valid values:* list of 1 or more real values, 0.0 to infinity

A list of real values, the same length as the list of observation types, to be used as the cutoff value for each of the given observation types. This is only implemented for the threed_sphere location module (the one used by most geophysical models.)

Guidance regarding localization

There are a large set of options for localization. Individual cases may differ but in general the following guidelines might help. Most users use the Gaspari-Cohn covariance cutoff type. The value of the cutoff itself is the item most often changed in a sensitivity run to pick a good general value, and then left as-is for subsequent runs. Most localize in the vertical, but tend to use large values so as to not disturb vertical structures. Users do not generally use adaptive localization, unless their observations are very dense in some areas and sparse in others.

The advice for setting good values for the cutoff value is to err on the larger side - to estimate for all types of observations under all conditions what the farthest feasible impact or correlated structure size would be. The downsides of guessing too large are 1) run time is slower, and 2) there can be spurious correlations between state vector items and observations which aren’t physically related and noise can creep into the assimilation results this way. The downside of guessing too small is that state vector items that should get an impact from an observation won't. This might disrupt organized features in a field and the model may take more time to recover/reconstruct the feature.

6.134.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&assim_tools_nml
   filter_kind                       = 1
   cutoff                            = 0.2
   distribute_mean                   = .false.
   sort_obs_inc                      = .false.
   spread_restoration                = .false.
   sampling_error_correction         = .false.
   adaptive_localization_threshold   = -1
   adaptive_cutoff_floor             = 0.0
   output_localization_diagnostics   = .false.
   localization_diagnostics_file     = "localization_diagnostics"
   print_every_nth_obs               = 0
   rectangular_quadrature            = .true.
   gaussian_likelihood_tails         = .false.
   close_obs_caching                 = .true.
   allow_missing_in_clm              = .false.
   adjust_obs_impact                 = .false.
   obs_impact_filename               = ""
   allow_any_impact_values           = .false.
   convert_all_obs_verticals_first   = .true.
   convert_all_state_verticals_first = .false.
   special_localization_obs_types    = 'null'
   special_localization_cutoffs      = -888888.0
  /
```

Description of each namelist entry

`filter_kind` *type:* integer

Selects the variant of filter to be used.

- 1 = EAKF (Ensemble Adjustment Kalman Filter, see Anderson 2001)
- 2 = ENKF (Ensemble Kalman Filter)
- 3 = Kernel filter
- 4 = Observation Space Particle filter
- 5 = Random draw from posterior (contact dart@ucar.edu before using)
- 6 = Deterministic draw from posterior with fixed kurtosis (ditto)
- 7 = Boxcar kernel filter
- 8 = Rank Histogram filter (see Anderson 2010)
- 9 = Particle filter (see Poterjoy 2016)

The EAKF is the most commonly used filter. Note that although the algorithm is expressed in a slightly different form, the EAKF is identical to the EnSRF (Ensemble Square Root Filter) described by Whitaker and Hamill in 2002.

The Rank Histgram filter can be more successful for highly nongaussian distributions.

Jon Poterjoy's Particle filter is included with this code release. To use, it, overwrite `assim_tools_mod.f90` with `assim_tools_mod.pf.f90` and rebuild filter.

```
$ mv assimilation_code/modules/assimilation/assim_tools_mod.pf.f90 assimilation_
↪code/modules/assimilation/assim_tools_mod.f90
```

There are additional namelist items in this version specific to the particle filter. Read the code for more details.

`cutoff` *type:* real(r8)

Cutoff controls a distance dependent weight that modulates the impact of an observation on a state variable. The units depend both on the location module being used and on the covariance cutoff module options selected. As defined in the original paper, this is the half-width; the localization goes to 0 at 2 times this value.

`distribute_mean` *type:* logical

If your model uses coordinates that have no options for different vertical coordinates then this setting has no effect on speed and should be .true. to use less memory. If your model has code to convert between different coordinate systems, for example Pressure, Height, Model Levels, etc, then setting this .false. will generally run much faster at assimilation time but will require more memory per MPI task. If you run out of memory, setting this to .true. may allow you to run but take longer.

`sort_obs_inc` *type:* logical

If true, the final increments from obs_increment are sorted so that the mean increment value is as small as possible. This minimizes regression errors when non-deterministic filters or error correction algorithms are applied. HOWEVER, when using deterministic filters (filter_kind == 1 or 8) with no inflation or a combination of a determinstic filter and deterministic inflation (filter_nml:inf_deterministic = .TRUE.) sorting the increments is both unnecessary and expensive. A warning is printed to stdout and the log and the sorting is skipped.

`spread_restoration` *type:* logical

True turns on algorithm to restore amount of spread that would be expected to be lost if underlying obs/state variable correlation were really 0.

`sampling_error_correction` *type:* logical

If true, apply sampling error corrections to the correlation values based on the ensemble size. See Anderson 2012. This option uses special input files generated by the gen_sampling_err_table tool in the assimilation_code/programs directory. The values are generated for a specific ensemble size and most common ensemble sizes have precomputed entries in the table. There is no dependence on which model is being used, only on the number of ensemble members. The input file must exist in the directory where the filter program is executing.

`adaptive_localization_threshold` *type:* integer

Used to reduce the impact of observations in densely observed regions. If the number of observations close to a given observation is greater than the threshold number, the cutoff radius for localization is adjusted to try to make the number of observations close to the given observation be the threshold number. This should be dependent on the location module and is tuned for a three_dimensional spherical implementation for numerical weather prediction models at present.

`adaptive_cutoff_floor` *type:* real

If adaptive localization is enabled and if this value is greater than 0, then the adaptive cutoff distance will be set to a value no smaller than the distance specified here. This guarentees a minimum cutoff value even in regions of very dense observations.

output_localization_diagnostics *type:* logical

Setting this to .true. will output an additional text file that contains the obs key, the obs time, the obs location, the cutoff distance and the number of other obs which are within that radius. If adaptive localization is enabled, the output also contains the updated cutoff distance and the number of other obs within that new radius. Without adaptive localization there will be a text line for each observation, so this file could get very large. With adaptive localization enabled, there will only be one line per observation where the radius is changed, so the size of the file will depend on the number of changed cutoffs.

localization_diagnostics_file *type:* character(len=129)

Filename for the localization diagnostics information. This file will be opened in append mode, so new information will be written at the end of any existing data.

print_every_nth_obs *type:* integer

If set to a value N greater than 0, the observation assimilation loop prints out a progress message every Nth observations. This can be useful to estimate the expected run time for a large observation file, or to verify progress is being made in cases with suspected problems.

rectangular_quadrature *type:* logical

Only relevant for filter type 8 and recommended to leave .true..

gaussian_likelihood_tails *type:* logical

Only relevant for filter type 8 and recommended to leave .false..

close_obs_caching *type:* logical

Should remain .TRUE. unless you are using specialized_localization_cutoffs. In that case to get accurate results, set it to .FALSE.. This also needs to be .FALSE. if you have a get_close_obs() routine in your model_mod file that uses the types/kinds of the obs to adjust the distances.

allow_missing_in_clm *type:* logical

If true, missing values (MISSING_R8 as defined in the types_mod.f90 file) are allowed in the state vector. Model interpolation routines must be written to recognize this value and fail the interpolation. During assimilation any state vector items where one or more of the ensemble members are missing will be skipped and their values will be unchanged by the assimilation. The system currently has limited support for this option; the CLM model has been tested and is known to work. If your model would benefit from setting missing values in the state vector, contact DAReS staff by emailing dart@ucar.edu.

adjust_obs_impact *type:* logical

If true, reads a table of observation quantities and types which should be artifically adjusted regardless of the actual correlation computed during assimilation. Setting the impact value to 0 prevents items from being adjusted by that class of observations. The input file can be constructed by the 'obs_impact_tool' program, included in this release. See the documentation for more details.

obs_impact_filename *type:* character(len=256)

If adjust_obs_impact is true, the name of the file with the observation types and quantities and state quantities that should have have an additional factor applied to the correlations during assimilation.

allow_any_impact_values *type:* logical

If .false., then the impact values can only be zero or one (0.0 or 1.0) - any other value will throw an error. .false. is the recommended setting.

convert_all_obs_verticals_first *type:* logical

Should generally always be left .True.. For models without vertical conversion choices the setting of this item has no impact.

convert_all_state_verticals_first *type:* logical

If the model has multiple choices for the vertical coordinate system during localization (e.g. pressure, height, etc) then this should be .true. if previous versions of get_state_meta_data() did a vertical conversion or if most of the state is going to be impacted by at least one observation. If only part of the state is going to be updated or if get_state_meta_data() never used to do vertical conversions, leave it .false.. The results should be the same but the run time may be impacted by doing unneeded conversions up front. For models without vertical conversion choices the setting of this item has no impact.

special_localization_obs_types *type:* character(len=32), dimension(:)

Optional list of observation types (e.g. "RADAR_REFLECTIVITY", "RADIOSONDE_TEMPERATURE") which will use a different cutoff value other than the default specified by the 'cutoff' namelist. This is only implemented for the 'threed_sphere' locations module.

special_localization_cutoffs *type:* real(r8), dimension(:)

Optional list of real values which must be the same length and in the same order as the observation types list given for the 'special_localization_obs_types' item. These values will set a different cutoff distance for localization based on the type of the observation currently being assimilated. Any observation type not in the list will use the default cutoff value. This is only implemented for the 'threed_sphere' locations module.

6.134.5 Other modules used

```
types_mod
utilities_mod
sort_mod
random_seq_mod
obs_sequence_mod
obs_def_mod
cov_cutoff_mod
reg_factor_mod
location_mod (model dependent choice)
ensemble_manager_mod
mpi_utilities_mod
adaptive_inflate_mod
time_manager_mod
assim_model_mod
```

6.134.6 Public interfaces

use assim_tools_mod, only :	filter_assim

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

call filter_assim(ens_handle, obs_ens_handle, obs_seq, keys, ens_size, num_groups, obs_val_index, inflate, ens_mean_copy, ens_sd_copy, ens_inf_copy, ens_inf_sd_copy, obs_key_copy, obs_global_qc_copy, obs_prior_mean_start, obs_prior_mean_end, obs_prior_var_start, obs_prior_var_end, inflate_only)

```
type(ensemble_type), intent(inout)         :: ens_handle
type(ensemble_type), intent(inout)         :: obs_ens_handle
type(obs_sequence_type), intent(in)        :: obs_seq
integer, intent(in)                        :: keys(:)
integer, intent(in)                        :: ens_size
integer, intent(in)                        :: num_groups
integer, intent(in)                        :: obs_val_index
type(adaptive_inflate_type), intent(inout) :: inflate
integer, intent(in)                        :: ens_mean_copy
integer, intent(in)                        :: ens_sd_copy
integer, intent(in)                        :: ens_inf_copy
integer, intent(in)                        :: ens_inf_sd_copy
integer, intent(in)                        :: obs_key_copy
integer, intent(in)                        :: obs_global_qc_copy
integer, intent(in)                        :: obs_prior_mean_start
integer, intent(in)                        :: obs_prior_mean_end
integer, intent(in)                        :: obs_prior_var_start
integer, intent(in)                        :: obs_prior_var_end
logical, intent(in)                        :: inflate_only
```

Does assimilation and inflation for a set of observations that is identified by having integer indices listed in keys. Only the inflation is updated if inflation_only is true, otherwise the state is also updated.

`ens_handle`	Contains state variable ensemble data and description.
`obs_ens_handle`	Contains observation prior variable ensemble and description.
`obs_seq`	Contains the observation sequence including observed values and error variances.
`keys`	A list of integer indices of observations in obs_seq that are to be used at this time.
`ens_size`	Number of ensemble members in state and observation prior ensembles.
`num_groups`	Number of groups being used in assimilation.
`obs_val_index`	Integer index of copy in obs_seq that contains the observed value from instrument.
`inflate`	Contains inflation values and all information about inflation to be used.
`ens_mean_copy`	Index of copy containing ensemble mean in ens_handle.
`ens_sd_copy`	Index of copy containing ensemble standard deviation in ens_handle.
`ens_inf_copy`	Index of copy containing state space inflation in ens_handle.
`ens_inf_sd_copy`	Index of copy containing state space inflation standard deviation in ens_handle.
`obs_key_copy`	Index of copy containing unique key for observation in obs_ens_handle.
`obs_global_qc_copy`	Index of copy containing global quality control value in obs_ens_handle.
`obs_prior_mean_start`	Index of copy containing first group's prior mean in obs_ens_handle.
`obs_prior_mean_end`	Index of copy containing last group's prior mean in obs_ens_handle.
`obs_prior_var_start`	Index of copy containing first group's ensemble variance in obs_ens_handle.
`obs_prior_var_end`	Index of copy containing last group's ensemble variance in obs_ens_handle.
`inflate_only`	True if only inflation is to be updated, and not state.

6.134.7 Files

filename	purpose
input.nml	to read `assim_tools_nml`

6.134.8 References

- Anderson, J. L., 2001: An Ensemble Adjustment Kalman Filter for Data Assimilation. Mon. Wea. Rev., 129, 2884-2903. doi: 10.1175/1520-0493(2001)129<2884:AEAKFF>2.0.CO;2
- Anderson, J. L., 2003: A Local Least Squares Framework for Ensemble Filtering. Mon. Wea. Rev., 131, 634-642. doi: 10.1175/1520-0493(2003)131<0634:ALLSFF>2.0.CO;2
- Anderson, J., Collins, N., 2007: Scalable Implementations of Ensemble Filter Algorithms for Data Assimilation. Journal of Atmospheric and Oceanic Technology, 24, 1452-1463. doi: 10.1175/JTECH2049.1
- Anderson, J. L., 2010: A Non-Gaussian Ensemble Filter Update for Data Assimilation. Mon. Wea. Rev., 139, 4186-4198. doi: 10.1175/2010MWR3253.1
- Anderson, J. L., 2012:, Localization and Sampling Error Correction in Ensemble Kalman Filter Data Assimilation. Mon. Wea. Rev., 140, 2359-2371. doi: 10.1175/MWR-D-11-00013.1
- Poterjoy, J., 2016:, A localized particle filter for high-dimensional nonlinear systems. Mon. Wea. Rev. 144 59-76. doi:10.1175/MWR-D-15-0163.1

6.134.9 Private components

N/A

6.135 MODULE cov_cutoff_mod

6.135.1 Overview

Computes the weight with which an observation should impact a state variable that is separated by a given distance. The distance is in units determined by the location module being used.

6.135.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&cov_cutoff_nml
   select_localization = 1
/
```

Item	Type	Description
select_localization	integer	Selects the localization function. • 1 = Gaspari-Cohn 5th order polynomial with halfwidth c. • 2 = Boxcar with halfwidth c (goes to 0 for z_in < 2c). • 3 = Ramped Boxcar. Has value 1 for z_in < c and then reduces linearly to 0 at z_in = 2c.

6.135.3 Other modules used

```
types_mod
utilities_mod
location_mod
```

6.135.4 Public interfaces

use cov_factor_mod, only :	comp_cov_factor

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

var = comp_cov_factor(z_in, c [, obs_loc] [, obs_type] [, target_loc] [, target_kind] [, localization_override])

```
real(r8)                                    :: comp_cov_factor
real(r8), intent(in)                        :: z_in
real(r8), intent(in)                        :: c
type(location_type), optional, intent(in)  :: obs_loc
integer, optional, intent(in)              :: obs_type
type(location_type), optional, intent(in)  :: target_loc
integer, optional, intent(in)              :: target_kind
integer, optional, intent(in)              :: localization_override
```

Returns a weighting factor for observation and a target variable (state or observation) separated by distance z_in and with a half-width distance, c. Three options are provided and controlled by a namelist parameter. The optional argument localization_override controls the type of localization function if present. The optional arguments obs_loc, obs_type and target_loc, target_kind are not used in the default code. They are made available for users who may want to design more sophisticated localization functions.

`var`	Weighting factor.
`z_in`	The distance between observation and target.
`c`	Factor that describes localization function. Describes half-width of functions used here.
obs_loc	Location of the observation.
obs_type	Observation specific type.
target_loc	Location of target.
target_kind	Generic kind of target.
localization_override	Controls localization type if present. Same values as for namelist control.

6.135.5 Files

filename	purpose
input.nml	to read `cov_cutoff_nml`

6.135.6 References

1. Gaspari and Cohn, 1999, QJRMS, **125**, 723-757. (eqn. 4.10)

6.135.7 Private components

N/A

6.136 MODULE obs_model_mod

6.136.1 Overview

The code in this module computes the assimilation windows, and decides if the model needs to run in order for the data to be at the appropriate time to assimilate the next available observations. It also has the code to write out the current states, advance the model (in a variety of ways) and then read back in the updated states.

6.136.2 Other modules used

```
types_mod
utilities_mod
assim_model_mod
obs_sequence_mod
obs_def_mod
time_manager_mod
ensemble_manager_mod
mpi_utilities_mod
```

6.136.3 Public interfaces

use obs_model_mod, only :	advance_state
	move_ahead

call move_ahead(ens_handle, ens_size, seq, last_key_used, window_time, key_bounds, num_obs_in_set, curr_ens_time, next_ens_time, trace_messages)

```
type(ensemble_type),      intent(in)  :: ens_handle
integer,                  intent(in)  :: ens_size
type(obs_sequence_type), intent(in)  :: seq
integer,                  intent(in)  :: last_key_used
type(time_type),          intent(in)  :: window_time
integer, dimension(2),    intent(out) :: key_bounds
integer,                  intent(out) :: num_obs_in_set
type(time_type),          intent(out) :: curr_ens_time
type(time_type),          intent(out) :: next_ens_time
logical, optional,        intent(in)  :: trace_messages
```

Given an observation sequence and an ensemble, determines how to advance the model so that the next set of observations can be assimilated. Also returns the first and last keys and the number of observations to be assimilated at this time. The algorithm implemented here (one might want to have other variants) first finds the time of the next observation that has not been assimilated at a previous time. It also determines the time of the ensemble state vectors. It then uses information about the model's time stepping capabilities to determine the time to which the model can be advanced that is CLOSEST to the time of the next observation. For now, this algorithm assumes that the model's timestep is a constant. A window of width equal to the model timestep is centered around the closest model time to the next observation and all observations in this window are added to the set to be assimilated.

Previous versions of this routine also made the call which actually advanced the model before returning. This is no longer true. The routine only determines the time stepping and number of observations. The calling code must then call advance_state() if indeed the next observation to be assimilated is not within the current window. This is determined by comparing the current ensemble time with the next ensemble time. If equal no advance is needed. Otherwise, next ensemble time is the target time for advance_state().

`ens_handle`	Identifies the model state ensemble
`ens_size`	Number of ensemble members
`seq`	An observation sequence
`last_key_used`	Identifies the last observation from the sequence that has been used
`window_time`	Reserved for future use.
`key_bounds`	Returned lower and upper bound on observations to be used at this time
`num_obs_in_set`	Number of observations to be used at this time
`curr_ens_time`	The time of the ensemble data passed into this routine.
`next_ens_time`	The time the ensemble data should be advanced to. If equal to curr_ens_time, the model does not need to advance to assimilate the next observation.
`trace_messages`	Optional argument. By default, detailed time trace messages are disabled but can be turned on by passing this in as .True. . The messages will print the current window times, data time, next observation time, next window time, next data time, etc.

call advance_state(ens_handle, ens_size, target_time, async, adv_ens_command, tasks_per_model_advance)

```
type(ensemble_type), intent(inout) :: ens_handle
integer, intent(in)                :: ens_size
type(time_type), intent(in)        :: target_time
integer, intent(in)                :: async
character(len=*), intent(in)       :: adv_ens_command
integer, intent(in)                :: tasks_per_model_advance
```

Advances all ensemble size copies of an ensemble stored in ens_handle to the target_time. If async=0 this is done by repeated calls to the `adv_1step()` subroutine. If async=2, a call to the shell with the command `adv_ens_command` is used. If async=4, the filter program synchronizes with the MPI job shell script using the `block_task()` and `restart_task()` routines to suspend execution until all model advances have completed. The script can start the model advances using MPI and have it execute in parallel in this mode.

`ens_handle`	Structure for holding ensemble information and data
`ens_size`	Ensemble size.
`target_time`	Time to which model is to be advanced.
`async`	How to advance model: 0 = subroutine adv_1step 2 = shell executes adv_ens_command 4 = MPI job script advances models and syncs with filter task
`adv_ens_command`	Command to be issued to shell to advance model if async=2.
`tasks_per_model_advance`	Reserved for future use.

6.136.4 Namelist

This module does not have a namelist.

6.136.5 Files

filename	purpose
assim_model_state_ic####	a binary representation of the state vector prepended by a small header consisting of the 'advance-to' time and the 'valid-time' of the state vector. The *####* represents the ensemble member number if `&ensemble_manager_nml: single_restart_file_out = .true.`.
assim_model_state_ud####	a binary representation of the state vector prepended by a small header consisting of the 'valid-time' of the state vector. This is the 'updated' model state (after the model has advanced the state to the desired 'advance-to' time).
filter_control####	a text file containing information needed to advance the ensemble members; i.e., the ensemble member number, the input state vector file, the output state vector file - that sort of thing.

6.136.6 References

- none

6.136.7 Private components

N/A

6.137 MODULE reg_factor

6.137.1 Overview

Computes a weighting factor to reduce the impact of observations on state variables using information from groups of ensembles. Can be run using groups or using archived summary information available from previous group filter experiments.

6.137.2 Other modules used

```
types_mod
utilities_mod
time_manager_mod
```

6.137.3 Public interfaces

use reg_factor_mod, only :	comp_reg_factor

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

var = comp_reg_factor(num_groups, regress, obs_index, state_index [, obs_state_ind] [, obs_state_max])

```
real(r8)                                   :: comp_reg_factor
integer,                       intent(in)  :: num_groups
real(r8), dimension(num_groups), intent(in) :: regress
integer,                       intent(in)  :: obs_index
integer,                       intent(in)  :: state_index
integer, optional,             intent(in)  :: obs_state_ind
integer, optional,             intent(in)  :: obs_state_max
```

Returns a weighting factor given regression factors from each group of a group filter or retrieves a factor generated by previous group filter runs from a file.

`num_groups`	Number of groups. Set to 1 when using information from previously run group filter from file.
`regress`	Regression factor from each group for a given state variable and observation variable pair.
`obs_index`	Integer index of the observation being processed. Not used in current implementation .
`state_index`	Integer index of state variable being processed. Not used in current implementation.
obs_state_ind	Index into file generated for Bgrid model which could be duplicated in other large models.
obs_state_max	Maximum number of observation state variable pairs with non-zero impacts for a given model and observation sequence. Used for generating Bgrid statistic files.

6.137.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&reg_factor_nml
   select_regression    = 1,
   input_reg_file       = "time_mean_reg",
   save_reg_diagnostics = .false.,
   reg_diagnostics_file = "reg_diagnostics"
/
```

Item	Type	Description
select_regression	integer	Selects the method for computing regression factor. • 1 = compute using sampling theory for any ensemble size. • 2 = low order model format. Works from archived time mean or time median regression files generated by low-order models like Lorenz-96. • 3 = selects bgrid archived file. This is not currently supported in released versions.
input_reg_file	character(len=129)	File name from which statistics are to be read for select_regression = 3.
save_reg_diagnostics	logical	True if regression diagnostics should be computed.
reg_diagnostics_file	character(len=129)	File name to which to write diagnostics.

6.137.5 Files

- (optional) input regression file from namelist variable input_reg_file.
- reg_factor_mod.nml in input.nml

filename	purpose
from `input.nml`&reg_factor_mod:input_reg_file	file of regression coefficients

6.137.6 References

- none

6.137.7 Private components

N/A

6.138 MODULE adaptive_inflate_mod

6.138.1 Overview

This module implements a variety of hierarchical Bayesian adaptive inflation algorithms for use with ensemble filters. It can provide constant valued inflation in state or observation space, consistent with previous DART releases. It can provide spatially-constant, time-varying adaptive inflation. It can provide spatially-varying, time-varying adaptive inflation and it can provide temporally-varying observation space inflation. And finally, it can provide adaptive damped inflation, which decreases inflation through time when observation density varies. Diagnostic output and restart files are available. Several papers on the NCAR IMAGe/DAReS web page document the algorithms in detail. The `DART/tutorial/section12` chapter has more information.

Details on controlling the inflation options are contained in the documentation for the filter. The filter_nml controls what inflation options are used.

Inflation flavor 3 (spatially-constant state space) reads and writes a restart file that is the full size of the state vector, however it takes the first value in the array and replicates that throughout the array. This allows one to switch between flavors 2 and 3. Going from inflation flavor 3 to 2 the initial value for all items in the state vector will be a constant value and will then start to adapt. Going from inflation flavor 2 to 3 whatever value is in the array at index 1 will be replicated and used for the entire rest of the state vector items.

6.138.2 Other modules used

```
types_mod
utilities_mod
random_seq_mod
time_manager_mod
ensemble_manager_mod
```

6.138.3 Public interfaces

use adaptive_inflate_mod, only :	update_inflation
	adaptive_inflate_end
	inflate_ens
	output_inflate_diagnostics
	do_obs_inflate
	do_single_ss_inflate
	do_varying_ss_inflate
	adaptive_inflate_init
	adaptive_inflate_type
	get_inflate
	set_inflate
	set_sd
	set_sd
	deterministic_inflate

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

call update_inflation(inflate_handle, inflate, inflate_sd, prior_mean, prior_var, obs, obs_var, gamma)

```
type(adaptive_inflate_type), intent(in)    :: inflate_handle
real(r8),                    intent(inout) :: inflate
real(r8),                    intent(inout) :: inflate_sd
real(r8),                    intent(in)    :: prior_mean
real(r8),                    intent(in)    :: prior_var
real(r8),                    intent(in)    :: obs
real(r8),                    intent(in)    :: obs_var
real(r8),                    intent(in)    :: gamma
```

Updates the mean and standard deviation of an inflation distribution given the prior values, the prior observation ensemble mean and variance, and the observation and its error variance. The factor gamma is the expected impact (0 to 1) of the state variable corresponding to the inflation on the observation and is the product of the ensemble correlation plus an additional localization factor or group regression factors.

inflate_handle	Handle to object that describes the inflation type and values.
inflate	Prior mean value of the inflation distribution.
inflate_sd	Prior standard deviation of the inflation distribution.
prior_mean	Mean of the prior observation ensemble.
prior_var	Variance of the prior observation ensemble.
obs	The observed value.
obs_var	Observational error variance.
gamma	Expected impact factor, product of correlation, localization, regression factor.

call adaptive_inflate_end(inflate_handle, ens_handle, ss_inflate_index, ss_inflate_sd_index)

```
type(adaptive_inflate_type), intent(in)    :: inflate_handle
type(ensemble_type),         intent(inout) :: ens_handle
integer,                     intent(in)    :: ss_inflate_index
integer,                     intent(in)    :: ss_inflate_sd_index
```

Outputs the values of inflation to restart files using the ensemble_manager for state space inflation and file output for observation space inflation. Releases allocated storage in inflate_handle.

inflate_handle	Handle for the details of the inflation being performed.
ens_handle	Handle for ensemble storage that holds values of state space inflation.
ss_inflate_index	Index in ensemble storage copies for state space inflation.
ss_inflate_sd_index	Index in ensemble storage copies for state space inflation standard deviation.

call inflate_ens(inflate_handle, ens,mean, inflate [,var_in])

```
type(adaptive_inflate_type),                 intent(in)  :: inflate_handle
real(r8),                    dimension(:),  intent(out) :: ens
real(r8),                                    intent(in)  :: mean
real(r8),                                    intent(in)  :: inflate
real(r8),                    optional,      intent(in)  :: var_in
```

Given an ensemble, its mean and the covarance inflation factor, inflates the ensemble.

inflate_handle	Handle for the details of the inflation being performed.
ens	Values for the ensemble to be inflated
mean	The mean of the ensemble.
inflate	The covariance inflation factor.
var_in	The variance of the ensemble.

call output_inflate_diagnostics(inflate_handle, time)

```
type(adaptive_inflate_type), intent(in) :: inflate_handle
type(time_type),             intent(in) :: time
```

Outputs diagnostic record of inflation for the observation space of spatially constant state space inflation. Spatially varying state space diagnostics are in the Posterior and Prior Diagnostic netcdf files and are written with calls from filter.f90.

inflate_handle	Handle for the details of the inflation being performed.
time	Time of this diagnostic info.

var = do_obs_inflate(inflate_handle)

```
logical,               intent(out) :: do_obs_inflate
adaptive_inflate_type, intent(in)  :: inflate_handle
```

Returns true if observation space inflation is being done by this handle.

do_obs_inflate	True if obs space inflation is being done by this handle.
inflate_handle	Handle to inflation details.

var = do_varying_ss_inflate(inflate_handle)

```
logical,               intent(out) :: do_varying_ss_inflate
adaptive_inflate_type, intent(in)  :: inflate_handle
```

Returns true if spatially varying state space inflation is being done by this handle.

do_varying_ss_inflate	True if spatially varying state space inflation is being done by this handle.
inflate_handle	Handle to inflation details.

var = do_single_ss_inflate(inflate_handle)

```
logical,               intent(out) :: do_single_ss_inflate
adaptive_inflate_type, intent(in)  :: inflate_handle
```

Returns true if spatially fixed state space inflation is being done by this handle.

do_single_ss_inflate	True if spatially fixed state space inflation is being done by this handle.
inflate_handle	Handle to inflation details.

call adaptive_inflate_init(inflate_handle, inf_flavor, mean_from_restart, sd_from_restart, output_restart, deterministic, in_file_name, out_file_name, diag_file_name, inf_initial, sd_initial, inf_lower_bound, inf_upper_bound, sd_lower_bound, ens_handle, ss_inflate_index, ss_inflate_sd_index, label)

```
type(adaptive_inflate_type), intent(inout) :: inflate_handle
integer, intent(in)                        :: inf_flavor
logical, intent(in)                        :: mean_from_restart
logical, intent(in)                        :: sd_from_restart
logical, intent(in)                        :: output_restart
logical, intent(in)                        :: deterministic
character(len=*), intent(in)               :: in_file_name
character(len=*), intent(in)               :: out_file_name
character(len=*), intent(in)               :: diag_file_name
real(r8), intent(in)                       :: inf_initial
real(r8), intent(in)                       :: sd_initial
real(r8), intent(in)                       :: inf_lower_bound
real(r8), intent(in)                       :: inf_upper_bound
real(r8), intent(in)                       :: sd_lower_bound
type(ensemble_type), intent(inout)         :: ens_handle
integer, intent(in)                        :: ss_inflate_index
integer, intent(in)                        :: ss_inflate_sd_index
character(len=*), intent(in)               :: label
```

Initializes a descriptor of an inflation object.

`inflate_handle`	Handle for the inflation descriptor being initialized.
`inf_flavor`	Type of inflation, 1=obs_inflate, 2=varying_ss_inflate, 3=single_ss_inflate.
`mean_from_restart`	True if inflation mean values to be read from restart file.
`sd_from_restart`	True if inflation standard deviation values to be read from restart file.
`output_restart`	True if an inflation restart file is to be output.
`deterministic`	True if deterministic inflation is to be done.
`in_file_name`	File name from which to read restart.
`out_file_name`	File name to which to write restart.
`diag_file_name`	File name to which to write diagnostic output; obs space inflation only .
`inf_initial`	Initial value of inflation for start_from_restart=.false.
`sd_initial`	Initial value of inflation standard deviation for start_from_restart=.false.
`inf_lower_bound`	Lower bound on inflation value.
`inf_upper_bound`	Upper bound on inflation value.
`sd_lower_bound`	Lower bound on inflation standard deviation.
`ens_handle`	Ensemble handle with storage for state space inflation.
`ss_inflate_index`	Index op copy in ensemble storage for inflation value.
`ss_inflate_sd_index`	Index of copy in ensemble storage for inflation standard deviation.
`label`	Character label to be used in diagnostic output (e.g. ‘Prior’, ‘Posterior’).

var = get_sd(inflate_handle)

```
real(r8), intent(out)                    :: get_sd
type(adaptive_inflate_type), intent(in)  :: inflate_handle
```

Returns value of observation space inflation standard deviation.

`get_sd`	Returns the value of observation space inflation.
`inflate_handle`	Handle for inflation descriptor.

var = get_inflate(inflate_handle)

```
real(r8), intent(out)                    :: get_inflate
type(adaptive_inflate_type), intent(in)  :: inflate_handle
```

Returns value of observation space inflation.

`get_inflate`	Returns the value of observation space inflation.
`inflate_handle`	Handle for inflation descriptor.

call set_inflate(inflate_handle, inflate)

```
type(adaptive_inflate_type), intent(inout) :: inflate_handle
real(r8), intent(in)                       :: inflate
```

Set the value of observation space inflation.

`inflate_handle`	Handle for inflation descriptor.
`inflate`	Set observation space inflation to this value.

call set_sd(inflate_handle, sd)

```
type(adaptive_inflate_type), intent(inout) :: inflate_handle
real(r8), intent(in)                       :: sd
```

Set the value of observation space inflation standard deviation.

`inflate_handle`	Handle for inflation descriptor.
`sd`	Set observation space inflation standard deviation to this value.

var = deterministic_inflate(inflate_handle)

```
logical, intent(out)                    :: deterministic_inflate
type(adaptive_inflate_type), intent(in) :: inflate_handle
```

Returns true if deterministic inflation is being done.

`deterministic_inflate`	Returns true if deterministic inflation is being done.
`inflate_handle`	Handle for inflation descriptor.

```
type adaptive_inflate_type
   private
   integer :: inflation_flavor
   integer :: obs_diag_unit
   logical :: start_from_restart
   logical :: output_restart
   logical :: deterministic
   character(len = 129) :: in_file_name
   character(len = 129) :: out_file_name
   character(len = 129) :: diag_file_name
   real(r8) :: inflate
   real(r8) :: sd
   real(r8) :: sd_lower_bound
   real(r8) :: inf_lower_bound
   real(r8) :: inf_upper_bound
   type(random_seq_type) :: ran_seq
end type adaptive_inflate_type
```

Provides a handle for a descriptor of inflation. Includes type of inflation, values controlling it, input and output file names, an output file descriptor for observation space inflation diagnostics, and a random sequence for doing reproducible non-determinstic inflation. There are 2 instances of this type, one for Prior and one for Posterior inflation.

Component	Description
inflation_flavor	Type of inflation; 0=none, 1=obs. space, 2=spatially varying, 3=spatially-fixed.
obs_diag_unit	Unit descriptor for output diagnostic file.
start_from_restart	True if initial inflation to be read from file.
output_restart	True if final inflation values to be written to file.
deterministic	True if inflation is to be done be deterministic algorithm.
in_file_name	File name containing restart.
out_file_name	File to contain output restart.
diag_file_name	File to hold observation space diagnostics.
inflate	Initial value of inflation for all types; current value for obs. space.
sd	Initial value of sd for all types; current value for obs. space.
sd_lower_bound	Don’t allow standard deviation to get smaller than this.
inf_lower_bound	Don’t let inflation get smaller than this.
inf_upper_bound	Don’t let inflation get larger than this.
ran_seq	Handle to random number sequence to allow reproducing non-deterministic inflate.

6.138.4 Namelist

The adaptive_inflate module no longer has a namelist. Control has been moved to &filter_nml in filter.

6.138.5 Files

Three files are opened from this module, but all names are passed in from the filter_nml now, and there are 2 values for each name: one for the prior and one for the posterior inflation.

- inf_in_file_name Mean and standard deviation values read in restart file format.
- inf_out_file_name Mean and standard deviation values written in restart file format.
- inf_diag_file_name Contains diagnostic history of inflation values for obs space and spatially-fixed state space inflation. Diagnostics for spatially-varying state space inflation are extra fields on the Posterior and Prior diagnostic netcdf files created in filter.f90.

6.138.6 References

- Anderson, J. L., 2007: An adaptive covariance inflation error correction algorithm for ensemble filters. Tellus A, 59, 210-224. doi: 10.1111/j.1600-0870.2006.00216.x
- Anderson, J. L., 2009: Spatially and temporally varying adaptive covariance inflation for ensemble filters. Tellus A, 61, 72-83. doi: 10.1111/j.1600-0870.2008.00361.x

6.138.7 Private components

no discussion

6.139 MODULE quality_control_mod

6.139.1 Overview

Routines in this module deal with two different types of quality control (QC) related functions. The first is to support interpretation of the *incoming* data quality, to reject observations at assimilation time which are marked as poor quality. The second is to document how DART disposed of each observation; whether it was successfully assimilated or rejected, and if rejected, for which reason.

6.139.2 Usage

Incoming data quality control

DART currently supports a single incoming quality control scheme compatible with NCEP usage. Lower values are considered better and higher values are considered poorer. A single namelist item, `input_qc_threshold` sets the boundary between accepted and rejected observations. Values *larger* than this value are rejected; values equal to or lower are accepted. Note that observations could be subsequently rejected for other reasons, including failing the outlier threshold test or all observations of this type being excluded by namelist control. See the obs_kind_mod namelist documentation for more details on how to enable or disable assimilation by observation type at runtime.

The incoming quality control value is set when an observation sequence file is created. If the data provider user a different scheme the values must be translated into NCEP-consistent values. Generally we use the value 3 for most runs.

Observations can also be rejected by the assimilation if the observation value is too far from the mean of the ensemble of expected values (the forward operator results). This is controlled by the `outlier_threshold` namelist item.

Specifically, the outlier test computes the difference between the observation value and the prior ensemble mean. It then computes a standard deviation by taking the square root of the sum of the observation error variance and the prior ensemble variance for the observation. If the difference between the ensemble mean and the observation value is more than the specified number of standard deviations then the observation is not used. This can be an effective way to discard clearly erroneous observation values. A commonly used value is 3. -1 disables this check.

There is an option to add code to this module to specialize the outlier threshold routine. For example, it is possible to allow all observations of one type to be assimilated regardless of the outlier value, and enforce the outlier threshold only on other types of observations. To enable this capability requires two actions: setting the `enable_special_outlier_code` namelist to `.TRUE.`, and adding your custom code to the `failed_outlier()` subroutine in this module.

DART outgoing quality control

As DART assimilates each observation it adds a *DART Quality Control* value to the output observation sequence (frequently written to a file named `obs_seq.final)`. This flag indicates how the observation was used during the assimilation. The flag is a numeric value with the following meanings:

0:	Observation was assimilated successfully
1:	Observation was evaluated only so not used in the assimilation
2:	The observation was used but one or more of the posterior forward observation operators failed
3:	The observation was evaluated only so not used AND one or more of the posterior forward observation operators failed
4:	One or more prior forward observation operators failed so the observation was not used
5:	The observation was not used because it was not selected in the namelist to be assimilated or evaluated
6:	The incoming quality control value was larger than the threshold so the observation was not used
7:	Outlier threshold test failed (as described above)
8:	The location conversion to the vertical localization unit failed so the observation was not used

6.139.3 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&quality_control_nml
   input_qc_threshold          = 3
   outlier_threshold           = -1
   enable_special_outlier_code = .false.
  /
```

Items in this namelist control whether an observation is assimilated or not.

Item	Type	Description
input_qc_threshold	integer	Numeric value indicating whether this observation is considered "good quality" and should be assimilated, or whether it is suspect because of previous quality control processes. This value would have been set when the observation was created and added to the observation sequence file. Observations with an incoming QC value larger than this threshold are rejected and not assimilated.
outlier threshold	integer	This numeric value defines the maximum number of standard deviations an observation value can be away from the ensemble forward operator mean and still be assimilated. Setting it to the value -1 disables this check.
enable_special_outlier_code	logical	Setting this value to .TRUE. will call a subroutine `failed_outlier()` instead of using the default code. The user can then customize the tests in this subroutine, for example to accept all observations of a particular type, or use different numerical thresholds for different observation types or locations.

6.139.4 Discussion

Small ensemble spread

If an ensemble is spun up from a single state the ensemble spread may be very small to begin and many observations may be rejected by the `outlier_threshold`. But as the ensemble spread increases the assimilation should be able to assimilate more and more observations as the model trajectory becomes consistent with those observations.

6.139.5 Other modules used

```
types_mod
utilities_mod
random_seq_mod
```

6.139.6 Public interfaces

`use quality_control_mod, only :`	initialize_qc
	input_qc_ok
	get_dart_qc
	check_outlier_threshold
	good_dart_qc
	set_input_qc
	dart_flags

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

call check_outlier_threshold(obs_prior_mean, obs_prior_var, obs_val, obs_err_var, & obs_seq, this_obs_key, dart_qc)

```
real(r8),                 intent(in)    :: obs_prior_mean !>  prior observation mean
real(r8),                 intent(in)    :: obs_prior_var  !>  prior observation␣
↪variance
real(r8),                 intent(in)    :: obs_val        !>  observation value
real(r8),                 intent(in)    :: obs_err_var    !>  observation error␣
↪variance
type(obs_sequence_type), intent(in)    :: obs_seq        !>  observation sequence
integer,                  intent(in)    :: this_obs_key   !>  index for this␣
↪observation
integer,                  intent(inout) :: dart_qc        !>  possibly modified DART QC
```

Computes whether this observation failed the outlier threshold test and if so, updates the DART QC.

var = input_qc_ok(input_qc, qc_to_use)

```
real(r8), intent(in)  :: input_qc    !> incoming QC data value
integer,  intent(out) :: qc_to_use   !> resulting DART QC
logical               :: input_qc_ok !> true if input_qc is good
```

Returns true if the input qc indicates this observation is good to use.

```
! Dart quality control variables
integer, parameter :: DARTQC_ASSIM_GOOD_FOP        = 0
integer, parameter :: DARTQC_EVAL_GOOD_FOP         = 1
integer, parameter :: DARTQC_ASSIM_FAILED_POST_FOP = 2
integer, parameter :: DARTQC_EVAL_FAILED_POST_FOP  = 3
integer, parameter :: DARTQC_FAILED_FOP            = 4
integer, parameter :: DARTQC_NOT_IN_NAMELIST       = 5
integer, parameter :: DARTQC_BAD_INCOMING_QC       = 6
integer, parameter :: DARTQC_FAILED_OUTLIER_TEST   = 7
integer, parameter :: DARTQC_FAILED_VERT_CONVERT   = 8
!!integer, parameter :: DARTQC_OUTSIDE_DOMAIN        = 9  ! we have no way (yet) for␣
↪the model_mod to signal this
```

These are public constants for use in other parts of the DART code.

6.139.7 Files

filename	purpose
input.nml	to read the quality_control_mod namelist

6.139.8 References

1. none

6.139.9 Error codes and conditions

Routine	Message	Comment
routine name	output string	description or comment

6.139.10 Future plans

Should support different incoming data QC schemes.

It would be nice to have a different DART QC flag for observations which fail the forward operator because they are simply outside the model domain. The diagnosic routines may indicate a large number of failed forward operators which make it confusing to identify observations where the forward operator should have been computed and can skew the statistics. Unfortunately, this requires adding an additional requirement on the model-dependent *model_mod.f90* code in the `model_interpolate()` routine. The current interface defines a return status code of 0 as success, any positive value as failure, and negative numbers are reserved for other uses. To identify obs outside the domain would require reserving another value that the interpolate routine could return.

At this time the best suggestion is to cull out-of-domain obs from the input observation sequence file by a preprocessing program before assimilation.

6.139.11 Private components

N/A

6.140 MODULE filter_mod

6.140.1 Overview

Main module for driving ensemble filter assimilations. Used by filter.f90, perfect_model_obs.f90, model_mod_check.f90, and a variety of test programs. See the *PROGRAM filter* for a general description of filter capabilities and controls.

`filter_mod` is a Fortran 90 module, and provides a large number of options for controlling execution behavior and parameter configuration that are driven from its namelist. See the namelist section below for more details. The number of assimilation steps to be done is controlled by the input observation sequence and by the time-stepping capabilities of the model being used in the assimilation.

See the DART web site for more documentation, including a discussion of the capabilities of the assimilation system, a diagram of the entire execution cycle, the options and features.

6.140.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&filter_nml
   single_file_in               = .false.,
   input_state_files            = '',
   input_state_file_list        = '',
   init_time_days               = 0,
   init_time_seconds            = 0,
   perturb_from_single_instance = .false.,
   perturbation_amplitude       = 0.2,

   stages_to_write              = 'output'

   single_file_out              = .false.,
   output_state_files           = '',
   output_state_file_list       = '',
   output_interval              = 1,
   output_members               = .true.,
   num_output_state_members     = 0,
   output_mean                  = .true.,
   output_sd                    = .true.,
   write_all_stages_at_end      = .false.,
   compute_posterior            = .true.

   ens_size                     = 20,
   num_groups                   = 1,
   distributed_state            = .true.,

   async                        = 0,
   adv_ens_command              = "./advance_model.csh",
   tasks_per_model_advance      = 1,

   obs_sequence_in_name         = "obs_seq.out",
   obs_sequence_out_name        = "obs_seq.final",
   num_output_obs_members       = 0,
   first_obs_days               = -1,
   first_obs_seconds            = -1,
   last_obs_days                = -1,
   last_obs_seconds             = -1,
   obs_window_days              = -1,
   obs_window_seconds           = -1,

   inf_flavor                   = 0,                        0,
   inf_initial_from_restart     = .false.,                  .false.,
   inf_sd_initial_from_restart  = .false.,                  .false.,
   inf_deterministic            = .true.,                   .true.,
   inf_initial                  = 1.0,                      1.0,
   inf_lower_bound              = 1.0,                      1.0,
   inf_upper_bound              = 1000000.0,                1000000.0,
   inf_damping                  = 1.0,                      1.0,
   inf_sd_initial               = 0.0,                      0.0,
   inf_sd_lower_bound           = 0.0,                      0.0,
   inf_sd_max_change            = 1.05,                     1.05,
```

(continues on next page)

(continued from previous page)

```
   trace_execution              = .false.,
   output_timestamps            = .false.,
   output_forward_op_errors     = .false.,
   write_obs_every_cycle        = .false.,
   silence                      = .false.,
 /
```

Particular options to be aware of are: async, ens_size, cutoff (localization radius), inflation flavor, outlier_threshold, restart filenames (including inflation), obs_sequence_in_name, horiz_dist_only, binary or ascii controls for observation sequence file formats. Some of these important items are located in other namelists, but all are in the same input.nml file.

The inflation control variables are all dimensioned 2, the first value controls the prior inflation and the second controls the posterior inflation.

Item

Type

Description

single_file_in

logical

True means all ensemble members are read from a single NetCDF file. False means each member is in a separate file. NOT SUPPORTED as of March, 2017 only multiple files can be used.

input_state_files

character(len=256) dimension(MAXFILES)

A list of the NetCDF files to open to read the state vectors. Models using multiple domains must put the domain and ensemble numbers in the file names. The order and format of those is to be determined. NOT SUPPORTED as of March, 2017.

input_state_file_list

character(len=256) dimension(MAXFILES)

A list of files, one per domain. Each file must be a text file containing the names of the NetCDF files to open, one per ensemble member, one per line.

init_time_days

integer

If negative, don't use. If non-negative, override the initial days read from state data restart files.

init_time_seconds

integer

If negative don't use. If non-negative, override the initial seconds read from state data restart files.

perturb_from_single_instance

logical

False means start from an ensemble-sized set of restart files. True means perturb a single state vector from one restart file. This may be done by model_mod, if model_mod provides subroutine `pert_model_copies`.

perturbation_amplitude

real(r8)

Standard deviation for the gaussian noise added when generating perturbed ensemble members. Ignored if `perturb_from_single_instance = .false.` or the perturbed ensemble is created in model_mod.
Random noise values drawn from a gaussian distribution with this standard deviation will be added to the data in a single initial ensemble member to generate the rest of the members.
This option is more frequently used in the low order models and less frequently used in large models. This is in part due to the different scales of real geophysical variable values, and the resulting inconsistencies between related field values. A more successful initial condition generation strategy is to generate climatological distributions from long model runs which have internally consistent structures and values and then use observations with a 'spin-up' period of assimilation to shape the initial states into a set of members with enough spread and which match the current set of observations.

stages_to_write

character(len=10) dimension(4)

Controls diagnostic and restart output. Valid values are 'input', 'preassim', 'postassim', 'output', and 'null'.

single_file_out

logical

True means all ensemble members are written to a single NetCDF file. False means each member is output in a separate file. NOT SUPPORTED as of March, 2017 - only multiple files can be used.

output_state_files

character(len=256) dimension(MAXFILES)

A list of the NetCDF files to open for writing updated state vectors. Models using multiple domains must put the domain and ensemble numbers in the file names. The order and format of those is to be determined. NOT SUPPORTED as of March, 2017.

output_state_file_list

character(len=256) dimension(MAXFILES)

A list of files, one per domain. Each file must be a text file containing the names of the NetCDF files to open, one per ensemble member, one per line.

output_interval

integer

Output state and observation diagnostics every 'N'th assimilation time, N is output_interval.

output_members

logical

True means output the ensemble members in any stage that is enabled.

num_output_state_members

integer

Number of ensemble members to be included in the state diagnostic output for stages 'preassim' and 'postassim'. output_members must be TRUE.

output_mean

logical

True means output the ensemble mean in any stage that is enabled.

output_sd

logical

True means output the ensemble standard deviation (spread) in any stage that is enabled.

write_all_stages_at_end

logical

For most cases this should be .false. and data will be output as it is generated for the 'preassim', 'postassim' diagnostics, and then restart data will be output at the end. However, if I/O time dominates the runtime, setting this to .true. will store the data and it can all be written in parallel at the end of the execution. This will require slightly more memory at runtime, but can lower the cost of the job significantly in some cases.

compute_posterior

logical

If .false., skip computing posterior forward operators and do not write posterior values in the obs_seq.final file. Saves time and memory. Cannot enable posterior inflation and skip computing the posteriors. For backwards compatibility the default for this is .true.

ens_size

integer

Size of ensemble.

num_groups

integer

Number of groups for hierarchical filter. It should evenly divide ens_size.

distributed_state

logical

True means the ensemble data is distributed across all tasks as it is read in, so a single task never has to have enough memory to store the data for an ensemble member. Large models should always set this to .true., while for small models it may be faster to set this to .false. This is different from *&assim_tools_mod :: distributed_mean* .

async

integer

Controls method for advancing model:

- 0 is subroutine call
- 2 is shell command
- 4 is mpi-job script

Ignored if filter is not controlling the model advance, e.g. in CESM assimilations.

adv_ens_command

character(len=256)

Command sent to shell if async is 2.

tasks_per_model_advance

integer

Number of tasks to assign to each ensemble member advance.

obs_sequence_in_name

character(len=256)

File name from which to read an observation sequence.

obs_sequence_out_name

character(len=256)

File name to which to write output observation sequence.

num_output_obs_members

integer

Number of ensemble members to be included in the output observation sequence file.

first_obs_days

integer

If negative, don't use. If non-negative, ignore all observations before this time.

first_obs_seconds

integer

If negative, don't use. If non-negative, ignore all observations before this time.

last_obs_days

integer

If negative, don't use. If non-negative, ignore all observations after this time.

last_obs_seconds

integer

If negative, don't use. If non-negative, ignore all observations after this time.

obs_window_days

integer

Assimilation window days; defaults to model timestep size.

obs_window_seconds

integer

Assimilation window seconds; defaults to model timestep size.

All variables named `inf_*` are arrays of length 2.

The first element controls the prior inflation, the second element controls the posterior inflation. See filter.html for a discussion of inflation and effective strategies.

inf_flavor

integer array dimension(2)

Inflation flavor for [prior, posterior]

- 0 = none
- 2 = spatially-varying state-space (gaussian)
- 3 = spatially-fixed state-space (gaussian)
- 4 = Relaxation To Prior Spread (Posterior inflation only)
- 5 = enhanced spatially-varying state-space (inverse gamma)

(See inf_sd_initial below for how to set the time evolution options.)

inf_initial_from_restart

logical array dimension(2)

If true, get initial mean values for inflation from restart file. If false, use the corresponding namelist value `inf_initial`.

inf_sd_initial_from_restart

logical array dimension(2)

If true, get initial standard deviation values for inflation from restart file. If false, use the corresponding namelist value `inf_sd_initial`.

inf_deterministic

logical array dimension(2)

True means deterministic inflation, false means stochastic.

inf_initial

real(r8) dimension(2)

Initial value of inflation if not read from restart file.

inf_lower_bound

real(r8) dimension(2)

Lower bound for inflation value.

inf_upper_bound

real(r8) dimension(2)

Upper bound for inflation value.

inf_damping

real(r8) dimension(2)

Damping factor for inflation mean values. The difference between the current inflation value and 1.0 is multiplied by this factor before the next assimilation cycle. The value should be between 0.0 and 1.0. Setting a value of 0.0 is full damping, which in fact turns all inflation off by fixing the inflation value at 1.0. A value of 1.0 turns inflation damping off leaving the original inflation value unchanged.

inf_sd_initial

real(r8) dimension(2)

Initial value of inflation standard deviation if not read from restart file. If 0, do not update the inflation values, so they are time-constant. If positive, the inflation values will adapt through time, so they are time-varying.

inf_sd_lower_bound

real(r8) dimension(2)

Lower bound for inflation standard deviation. If using a negative value for `sd_initial` this should also be negative to preserve the setting.

inf_sd_max_change

real(r8) dimension(2)

For inflation type 5 (enhanced inflation), controls the maximum change of the inflation standard deviation when adapting for the next assimilation cycle. The value should be between 1.0 and 2.0. 1.0 prevents any changes, while 2.0 allows 100% change. For the enhanced inflation option, if the standard deviation initial value is equal to the standard deviation lower bound the standard deviation will not adapt in time. See this section for a discussion of how the standard deviation adapts based on different types of inflation.

trace_execution

logical

True means output very detailed messages about what routines are being called in the main filter loop. Useful if a job hangs or otherwise doesn't execute as expected.

output_timestamps

logical

True means write timing information to the log before and after the model advance and the observation assimilation phases.

output_forward_op_errors

logical

True means output errors from forward observation operators. This is the 'istatus' error return code from the model_interpolate routine. An ascii text file `prior_forward_op_errors` and/or `post_forward_op_errors` will be created in the current directory. For each ensemble member which returns a non-zero return code, a line will be written to this file. Each line will have three values listed: the observation number, the ensemble member number, and the istatus return code. Be cautious when turning this option on. The number of lines in this file can be up to the number of observations times the number of ensemble members times the number of assimilation cycles performed. This option is generally most useful when run with a small observation sequence file and a small number of ensemble members to diagnose forward operator problems.

write_obs_every_cycle

logical

For debug use; this option can significantly slow the execution of filter. True means to write the entire output observation sequence diagnostic file each time through the main filter loop even though only observations with times up to and including the current model time will have been assimilated. Unassimilated observations have the value -888888.0 (the DART "missing value"). If filter crashes before finishing it may help to see the forward operator values of observations that have been assimilated so far.

silence

logical

True means output almost no runtime messages. Not recommended for general use, but can speed long runs of the lower order models if the execution time becomes dominated by the volume of output.

6.140.3 Modules used

```
types_mod
obs_sequence_mod
obs_def_mod
obs_def_utilities_mod
time_manager_mod
utilities_mod
assim_model_mod
assim_tools_mod
obs_model_mod
ensemble_manager_mod
adaptive_inflate_mod
mpi_utilities_mod
smoother_mod
random_seq_mod
state_vector_io_mod
io_filenames_mod
forward_operator_mod
quality_control_mod
```

6.140.4 Files

See the filter overview for the list of files.

6.141 MODULE location_mod

6.141.1 Overview

DART provides a selection of options for the coordinate system in which all observations and all model state vector locations are described. All executables are built with a single choice from the available location modules. The names of these modules are all `location_mod`.

6.141.2 Introduction

The core algorithms of DART work with many different models which have a variety of coordinate systems. This directory provides code for creating, setting/getting, copying location information (coordinates) independently of the actual specific coordinate information. It also contains distance routines needed by the DART algorithms.

Each of the different location_mod.f90 files provides the same set of interfaces and defines a ‘module location_mod’, so by selecting the proper version in your path_names_xxx file you can compile your model code with the main DART routines.

- *MODULE location_mod (threed_sphere)*: The most frequently used version for real-world 3d models. It uses latitude and longitude for horizontal coordinates, plus a vertical coordinate which can be meters, pressure, model level, surface, or no specific vertical location.
- *MODULE (1D) location_mod*: The most frequently used for small models (e.g. the Lorenz family). It has a cyclic domain from 0 to 1.
- others:
  - *MODULE location_mod (threed_cartesian)*: A full 3D X,Y,Z coordinate system.
  - column: no x,y but 1d height, pressure, or model level for vertical.
  - annulus: a hollow 3d cylinder with azimuth, radius, and depth.
  - twod: a periodic 2d domain with x,y coordinates between 0 and 1.
  - twod_sphere: a 2d shell with latitude, longitude pairs.
  - threed: a periodic 3d domain with x,y,z coordinates between 0 and 1.
  - *MODULE location_mod (channel)*: a 3d domain periodic in x, limited in y, and unlimited z.

Other schemes can be added, as needed by the models. Possible ideas are a non-periodic version of the 1d, 2d cartesian versions. Email dart at ucar.edu if you have a different coordinate scheme which we might want to support.

6.141.3 Namelist

Each location module option has a different namelist. See the specific documentation for the location option of choice.

6.141.4 Files

- none

6.141.5 References

- none

6.141.6 Private components

N/A

6.142 forward operator test README

6.142.1 Contents

6. *Terms of Use*

6.142.2 Overview

The `developer_tests/forward_operators` directory contains the testing framework for three kinds of tests:

- *rttov_test.f90* which tests basic functionality of the RTTOV radiance forward operators for AMSUA and AIRS observations (MW and IR, respectively)
  - create a dummy rttov_sensor_db_file to read
  - check the IR instrument ID
  - check the MW instrument ID
  - exercise the IR (direct) forward operator
  - exercise the MW (scatt) forward operator
- *rttov_unit_tests.f90* performs a host of unit tests.
  - If run with a TERMLEVEL of 3, all tests will be completed even if previous tests are not successful.
- *make_COS_input* and *make_assim_list* creates input for create_obs_sequence (COS) and the appropriate namelist settings to test the forward operator code.

Different sets of observations are grouped into separate files based on certain criteria - are they atmospheric observations, oceanic … do they require special metadata, etc. The following files are intended to be supplied as input to *make_COS_input* and will result in a text file that will generate an observation sequence file when used as input to *create_obs_sequence*.

- all_atm_obs_types
- all_commoncode_atm_obs_types
- all_f90s
- all_fwdop_atm_obs_types
- all_obs_types
- forward_op_code
- no_special_forward_op_code

See the *make_COS_input* section for more detail.

6.142.3 rttov_test.f90

This test requires several coefficient files that are not part of the default set provided by the RTTOV 12.3 distribution. Specifically:

- *rtcoef_eos_2_amsua.dat*
- *rtcoef_eos_2_airs.H5*
- *mietable_eos_amsua.dat* (same file as *mietable_noaa_amsua.dat*)

These coefficient files may be downloaded by using the *rtcoef_rttov12/rttov_coef_download.sh* script provided in the RTTOV distribution.

6.142.4 rttov_unit_tests.f90

These unit tests are best run with a TERMLEVEL of 3, which allows DART to continue past errors that would otherwise be fatal. If any of the unit tests are unable to start, the error code from *rttov_unit_tests* is 102. This is to give an error for *test_dart.csh* to detect.

Test	Pass	Fail
metadata growth	metadata arrays grow correctly as observations are added	incorrect metadata length
metadata content	metadata arrays contain correct data	incorrect data

6.142.5 make_COS_input

make_COS_input takes one filename as an argument and creates a text file that can be used as input for *create_obs_sequence*. The output text file has a name based on the input filename. For example:

```
<prompt> ./make_COS_input forward_op_code
 ready to run create_obs_sequence < forward_op_code_COS.in
```

create_obs_sequence must be created with the *preprocess_nml* settings to support the observation definitions required by the input file.

6.142.6 make_assim_list

make_assim_list is a follow-on step to *make_COS_input* and simply creates the text for the *input.nml:filter_nml:assimilate_these_obs* variable.

```
<prompt> forward_operators > ./make_assim_list forward_op_code
 created forward_op_code_obskind.nml
 add this section to your &obs_kind_nml in input.nml
 <prompt> head -n 10 forward_op_code_obskind.nml
 assimilate_these_obs_types =
 'ACARS_DEWPOINT,',
 'ACARS_RELATIVE_HUMIDITY,',
 'AIRCRAFT_DEWPOINT,',
 'AIRCRAFT_RELATIVE_HUMIDITY,',
 'AIREP_DEWPOINT,',
 'AIRS_DEWPOINT,',
 'AIRS_RELATIVE_HUMIDITY,',
 'AMDAR_DEWPOINT,',
 'AMSR_TOTAL_PRECIPITABLE_WATER,',
```

6.142.7 Terms of Use

6.143 PROGRAM `PrecisionCheck`

6.143.1 Overview

This is a self-contained program to explore the interaction between the compiler options to ‘autopromote’ variables from one precision to another and the intrinsic F90 mechanism for getting consistent behavior without relying on autopromotion - namely, the `SELECT_INT_KIND()` and `SELECT_REAL_KIND()` functions. The most portable code explicity types the variables to avoid relying on compiler flags. The core DART code abides by these rules; some pieces that are derived from dynamical models may have original code fragments.

All that is required is to compile the single file and run the resulting executable. There are no required libraries - any F90 compiler should have no trouble with this program. There is no input of any kind.

You are encouraged to view the source code. It’s pretty obvious what is being tested.

6.143.2 Examples

The following examples have differences from the default configuration highlighted in boldface. You are strongly encouraged to test your compiler and its autopromotion options. The Absoft compiler actually does what I consider to be reasonable and logical (as long as you know that “-dp” means **d**emote **p**recision). Many other compilers are surprising.

PowerPC chipset : Absoft Pro Fortran 9.0

```
[~/DART/utilities] % f90 PrecisionCheck.f90
[~/DART/utilities] % ./a.out

 This explores the use of the intrinisc SELECTED_[REAL,INT]_KIND() functions
 and the interplay with the compiler options. You are encouraged to use the
 "autopromotion" flags on your compiler and compare the results.

----------------------------------------------
 "integer"
 DIGITS    =   31
 HUGE      =   2147483647
 KIND      =   4
----------------------------------------------
 "integer(i4)" i4 = SELECTED_INT_KIND(8)
 DIGITS    =   31
 HUGE      =   2147483647
 KIND      =   4
----------------------------------------------
 "integer(i8)" i8 = SELECTED_INT_KIND(13)
 DIGITS    =   63
 HUGE      =   9223372036854775807
 KIND      =   8
----------------------------------------------
 "real"
 DIGITS    =   24
 EPSILON   =   1.192093E-07
 HUGE      =   3.402823E+38
 KIND      =   4
 PRECISION =   6
----------------------------------------------
```

(continues on next page)

(continued from previous page)

```
 "real(r4)" r4 = SELECTED_REAL_KIND(6,30)
 DIGITS    =   24
 EPSILON   =   1.192093E-07
 HUGE      =   3.402823E+38
 KIND      =   4
 PRECISION =   6
----------------------------------------------
 "real(r8)" r8 = SELECTED_REAL_KIND(13)
 DIGITS    =   53
 EPSILON   =   2.220446049250313E-016
 HUGE      =   1.797693134862315E+308
 KIND      =   8
 PRECISION =   15
----------------------------------------------
 "double precision"
 DIGITS    =   53
 EPSILON   =   2.220446049250313E-016
 HUGE      =   1.797693134862315E+308
 KIND      =   8
 PRECISION =   15
```

PowerPC chipset : Absoft Pro Fortran 9.0 : "-dp"

```
[~/DART/utilities] % f90 -dp PrecisionCheck.f90
[~/DART/utilities] % ./a.out

 This explores the use of the intrinisc SELECTED_[REAL,INT]_KIND() functions
 and the interplay with the compiler options. You are encouraged to use the
 "autopromotion" flags on your compiler and compare the results.

----------------------------------------------
 "integer"
 DIGITS    =   31
 HUGE      =   2147483647
 KIND      =   4
----------------------------------------------
 "integer(i4)" i4 = SELECTED_INT_KIND(8)
 DIGITS    =   31
 HUGE      =   2147483647
 KIND      =   4
----------------------------------------------
 "integer(i8)" i8 = SELECTED_INT_KIND(13)
 DIGITS    =   63
 HUGE      =   9223372036854775807
 KIND      =   8
----------------------------------------------
 "real"
 DIGITS    =   24
 EPSILON   =   1.192093E-07
 HUGE      =   3.402823E+38
 KIND      =   4
 PRECISION =   6
----------------------------------------------
 "real(r4)" r4 = SELECTED_REAL_KIND(6,30)
 DIGITS    =   24
```

(continues on next page)

(continued from previous page)

```
 EPSILON   =   1.192093E-07
 HUGE      =   3.402823E+38
 KIND      =   4
 PRECISION =   6
---------------------------------------------
 "real(r8)" r8 = SELECTED_REAL_KIND(13)
 DIGITS    =   53
 EPSILON   =   2.220446049250313E-016
 HUGE      =   1.797693134862315E+308
 KIND      =   8
 PRECISION =   15
---------------------------------------------
 "double precision"
 DIGITS    =   24
 EPSILON   =   1.192093E-07
 HUGE      =   3.402823E+38
 KIND      =   4
 PRECISION =   6
```

PowerPC chipset : Absoft Pro Fortran 9.0 : "-n113"

```
[~/DART/utilities] % f90 -N113 PrecisionCheck.f90
[~/DART/utilities] % ./a.out

 This explores the use of the intrinisc SELECTED_[REAL,INT]_KIND() functions
 and the interplay with the compiler options. You are encouraged to use the
 "autopromotion" flags on your compiler and compare the results.

---------------------------------------------
 "integer"
 DIGITS    =   31
 HUGE      =   2147483647
 KIND      =   4
---------------------------------------------
 "integer(i4)" i4 = SELECTED_INT_KIND(8)
 DIGITS    =   31
 HUGE      =   2147483647
 KIND      =   4
---------------------------------------------
 "integer(i8)" i8 = SELECTED_INT_KIND(13)
 DIGITS    =   63
 HUGE      =   9223372036854775807
 KIND      =   8
---------------------------------------------
 "real"
 DIGITS    =   53
 EPSILON   =   2.220446049250313E-016
 HUGE      =   1.797693134862315E+308
 KIND      =   8
 PRECISION =   15
---------------------------------------------
 "real(r4)" r4 = SELECTED_REAL_KIND(6,30)
 DIGITS    =   24
 EPSILON   =   1.192093E-07
 HUGE      =   3.402823E+38
```

(continues on next page)

(continued from previous page)

```
 KIND      =   4
 PRECISION =   6
----------------------------------------------
 "real(r8)" r8 = SELECTED_REAL_KIND(13)
 DIGITS    =   53
 EPSILON   =   2.220446049250313E-016
 HUGE      =   1.797693134862315E+308
 KIND      =   8
 PRECISION =   15
----------------------------------------------
 "double precision"
 DIGITS    =   53
 EPSILON   =   2.220446049250313E-016
 HUGE      =   1.797693134862315E+308
 KIND      =   8
 PRECISION =   15
```

6.144 MODULE obs_def_gps_mod

6.144.1 Overview

DART GPS Radio Occultation observation module, including the observation operators for both local and non-local refractivity computations.

Author and Contact information:

- GPS Science: Hui Liu, hliu at ncar.edu
- DART Code: Nancy Collins, nancy at ucar.edu

6.144.2 Namelist

This namelist is now enabled by default. The maximum number of GPS observations is settable at runtime by changing the value in the namelist. If you get an error about a missing namelist add &obs_def_gps_nml using the example below to your input.nml namelist file and rerun. No recompiling is needed.

This namelist is read from the file input.nml. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_def_gps_nml
  max_gpsro_obs = 100000,
/
```

Item	Type	Description
max_gpsro_obs	integer	The maximum number of GPS refractivity observations supported for a single execution. Generally the default will be sufficient for a single run of filter, but not enough for a long diagnostics run to produce a time series.

6.144.3 Other modules used

```
types_mod
utilities_mod
location_mod (threed_sphere)
assim_model_mod
obs_kind_mod
```

6.144.4 Public interfaces

use obs_def_gps_mod, only :	read_gpsro_ref
	write_gpsro_ref
	get_expected_gpsro_ref
	interactive_gpsro_ref
	set_gpsro_ref
	get_gpsro_ref

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

call read_gpsro_ref(gpskey, ifile, [, fform])

```
integer,           intent(out)           :: gpskey
integer,           intent(in)            :: ifile
character(len=*), intent(in), optional :: fform
```

Refractivity observations have several items of auxiliary data to read or write. This routine reads in the data for the next observation and returns the private GPS key index number that identifies the auxiliary data for this observation.

gpskey	GPS key number returned to the caller.
ifile	Open file unit number to read from.
fform	If specified, indicate whether the file was opened formatted or unformatted. Default is 'formatted'.

call write_gpsro_ref(gpskey, ifile, [, fform])

```
integer,           intent(in)           :: gpskey
integer,           intent(in)           :: ifile
character(len=*), intent(in), optional :: fform
```

Refractivity observations have several items of auxiliary data to read or write. This routine writes out the auxiliary data for the specified observation to the file unit given.

`gpskey`	GPS key number identifying which observation to write aux data for.
`ifile`	Open file unit number to write to.
fform	If specified, indicate whether the file was opened formatted or unformatted. Default is 'formatted'.

call get_expected_gpsro_ref(state_vector, location, gpskey, ro_ref, istatus)

```
real(r8),            intent(in)  :: state_vector(:)
type(location_type), intent(in)  :: location
integer,             intent(in)  :: gpskey
real(r8),            intent(out) :: ro_ref
integer,             intent(out) :: istatus
```

Given a location and the state vector from one of the ensemble members, compute the model-predicted GPS refractivity that would be observed at that location. There are two types of operators: modeled *local* refractivity (N-1)*1.0e6 or *non_local* refractivity (excess phase, m) The type is indicated in the auxiliary information for each observation.

`state_vector`	A one dimensional representation of the model state vector
`location`	Location of this observation
`gpskey`	Integer key identifying which GPS observation this is, so the correct corresponding auxiliary information can be accessed.
`ro_ref`	The returned GPS refractivity value
`istatus`	Returned integer status code describing problems with applying forward operator. 0 is a good value; any positive value indicates an error; negative values are reserved for internal DART use only.

call interactive_gpsro_ref(gpskey)

```
integer, intent(out) :: gpskey
```

Prompts the user for the auxiliary information needed for a GPS refractivity observation, and returns the new key associated with this data.

`gpskey`	Unique identifier associated with this GPS refractivity observation. In this code it is an integer index into module local arrays which hold the additional data. This routine returns the incremented value associated with this data.

call set_gpsro_ref(gpskey, nx, ny, nz, rfict0, ds, htop, subset0)

```
integer,           intent(out) :: gpskey
real(r8),          intent(in)  :: nx
real(r8),          intent(in)  :: ny
real(r8),          intent(in)  :: nz
real(r8),          intent(in)  :: rfict0
real(r8),          intent(in)  :: ds
real(r8),          intent(in)  :: htop
character(len=6), intent(in)  :: subset0
```

Sets the auxiliary information associated with a GPS refractivity observation. This routine increments and returns the new key associated with these values.

`gpskey`	Unique identifier associated with this GPS refractivity observation. In this code it is an integer index into module local arrays which hold the additional data. This routine returns the incremented value associated with this data.
`nx`	X component of direction of ray between the LEO (detector) satellite and the GPS transmitter satellite at the tangent point.
`ny`	Y component of tangent ray.
`nz`	Z component of tangent ray.
`rfict0`	Local curvature radius (meters).
`ds`	Delta S, increment to move along the ray in each direction when integrating the non-local operator (meters).
`htop`	Elevation (in meters) where integration stops along the ray.
`subset0`	The string 'GPSREF' for the local operator (refractivity computed only at the tangent point), or 'GPSEXC' for the non-local operator which computes excess phase along the ray.

call get_gpsro_ref(gpskey, nx, ny, nz, rfict0, ds, htop, subset0)

```
integer,           intent(in)  :: gpskey
real(r8),          intent(out) :: nx
real(r8),          intent(out) :: ny
real(r8),          intent(out) :: nz
real(r8),          intent(out) :: rfict0
real(r8),          intent(out) :: ds
real(r8),          intent(out) :: htop
character(len=6), intent(out) :: subset0
```

Gets the auxiliary information associated with a GPS refractivity observation, based on the GPS key number specified.

gpskey	Unique identifier associated with this GPS refractivity observation. In this code it is an integer index into module local arrays which hold the additional data. The value specified selects which observation to return data for.
nx	X component of direction of ray between the LEO (detector) satellite and the GPS transmitter satellite at the tangent point.
ny	Y component of tangent ray.
nz	Z component of tangent ray.
rfict0	Local curvature radius (meters).
ds	Delta S, increment to move along the ray in each direction when integrating the non-local operator (meters).
htop	Elevation (in meters) where integration stops along the ray.
subset	The string ‘GPSREF’ for the local operator (refractivity computed only at the tangent point), or ‘GPSEXC’ for the non-local operator which computes excess phase along the ray.

6.144.5 Files

- A DART observation sequence file containing GPS obs.

6.144.6 References

- Assimilation of GPS Radio Occultation Data for Numerical Weather Prediction, Kuo,Y.H., Sokolovskiy,S.V., Anthes,R.A., Vendenberghe,F., Terrestrial Atm and Ocn Sciences, Vol 11, pp157-186, 2000.

6.145 MODULE obs_def_dew_point_mod

6.145.1 Overview

Provides a subroutine to calculate the dew point temperature from model temperature, specific humidity, and pressure.

Revision 2801 implements a more robust method (based on Bolton’s Approximation) for calculating dew point. This has been further revised to avoid a numerical instability that could lead to failed forward operators for dewpoints almost exactly 0 C.

6.145.2 Other modules used

```
types_mod
utilities_mod
location_mod (most likely threed_sphere)
assim_model_mod
obs_kind_mod
```

6.145.3 Public interfaces

use obs_def_dew_point_mod, only :	get_expected_dew_point

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

call get_expected_dew_point(state_vector, location, key, td, istatus)

```
real(r8),             intent(in)  :: state_vector
type(location_type),  intent(in)  :: location
integer,              intent(in)  :: key
real(r8),             intent(out) :: td
integer,              intent(out) :: istatus
```

Calculates the dew point temperature (Kelvin).

`state_vector`	A one dimensional representation of the model state vector
`location`	Location for this obs
`key`	Controls whether upper levels (key = 1) or 2-meter (key = 2) is required.
`td`	The returned dew point temperature value
`istatus`	Returned integer describing problems with applying forward operator

6.145.4 Files

- NONE

6.145.5 References

1. Bolton, David, 1980: The Computation of Equivalent Potential Temperature. Monthly Weather Review, 108, 1046-1053.

6.146 MODULE obs_def_ocean_mod

6.146.1 Overview

DART includes a flexible, powerful, and slightly complicated mechanism for incorporating new types of observations. The `obs_def_ocean_mod` module being described here is used by the program `preprocess` to insert appropriate definitions of ocean observations into the `DEFAULT_obs_def_mod.f90` template and generate the source files `obs_def_mod.f90` and `obs_kind_mod.f90` that are used by `filter` and other DART programs.

There are no code segments in this module, only definitions of observation types that map specific observation types to generic observation quantities. DART contains logic that supports a limited inheritance of attributes. If you need to interpolate observations of 'FLOAT_TEMPERATURE', DART will check to see if a specific routine is provided for that type, if none exists, the interpolation routine for the generic 'QTY_TEMPERATURE' will be used; that way one interpolation routine may support many observation types.

The mandatory header line is followed by lines that have the observation type name (an all caps Fortran 90 identifier) and their associated generic quantity identifier from the obs_kind module. If there is no special processing needed for an observation type, and no additional data needed beyond the standard contents of an observation, then a third word on the line, the `COMMON_CODE` will instruct the preprocess program to automatically generate all stubs and code needed for this type. For observation types needing any special code or additional data, this word should not be specified and the user must supply the code manually. One of the future extensions of this module will be to support acoustic tomographic observations, which will necessitate specific support routines.

Ocean variable types and their corresponding quantities

```
! BEGIN DART PREPROCESS KIND LIST
!SALINITY,                      QTY_SALINITY,              COMMON_CODE
!TEMPERATURE,                   QTY_TEMPERATURE,           COMMON_CODE
!U_CURRENT_COMPONENT,           QTY_U_CURRENT_COMPONENT,   COMMON_CODE
!V_CURRENT_COMPONENT,           QTY_V_CURRENT_COMPONENT,   COMMON_CODE
!SEA_SURFACE_HEIGHT,            QTY_SEA_SURFACE_HEIGHT,    COMMON_CODE
!ARGO_U_CURRENT_COMPONENT,      QTY_U_CURRENT_COMPONENT,   COMMON_CODE
!ARGO_V_CURRENT_COMPONENT,      QTY_V_CURRENT_COMPONENT,   COMMON_CODE
!ARGO_SALINITY,                 QTY_SALINITY,              COMMON_CODE
!ARGO_TEMPERATURE,              QTY_TEMPERATURE,           COMMON_CODE
!ADCP_U_CURRENT_COMPONENT,      QTY_U_CURRENT_COMPONENT,   COMMON_CODE
!ADCP_V_CURRENT_COMPONENT,      QTY_V_CURRENT_COMPONENT,   COMMON_CODE
!ADCP_SALINITY,                 QTY_SALINITY,              COMMON_CODE
!ADCP_TEMPERATURE,              QTY_TEMPERATURE,           COMMON_CODE
!FLOAT_SALINITY,                QTY_SALINITY,              COMMON_CODE
!FLOAT_TEMPERATURE,             QTY_TEMPERATURE,           COMMON_CODE
!DRIFTER_U_CURRENT_COMPONENT,   QTY_U_CURRENT_COMPONENT,   COMMON_CODE
!DRIFTER_V_CURRENT_COMPONENT,   QTY_V_CURRENT_COMPONENT,   COMMON_CODE
!DRIFTER_SALINITY,              QTY_SALINITY,              COMMON_CODE
!DRIFTER_TEMPERATURE,           QTY_TEMPERATURE,           COMMON_CODE
!GLIDER_U_CURRENT_COMPONENT,    QTY_U_CURRENT_COMPONENT,   COMMON_CODE
!GLIDER_V_CURRENT_COMPONENT,    QTY_V_CURRENT_COMPONENT,   COMMON_CODE
!GLIDER_SALINITY,               QTY_SALINITY,              COMMON_CODE
!GLIDER_TEMPERATURE,            QTY_TEMPERATURE,           COMMON_CODE
!MOORING_U_CURRENT_COMPONENT,   QTY_U_CURRENT_COMPONENT,   COMMON_CODE
!MOORING_V_CURRENT_COMPONENT,   QTY_V_CURRENT_COMPONENT,   COMMON_CODE
!MOORING_SALINITY,              QTY_SALINITY,              COMMON_CODE
!MOORING_TEMPERATURE,           QTY_TEMPERATURE,           COMMON_CODE
!SATELLITE_MICROWAVE_SST,       QTY_TEMPERATURE,           COMMON_CODE
!SATELLITE_INFRARED_SST,        QTY_TEMPERATURE,           COMMON_CODE
!SATELLITE_SSH,                 QTY_SEA_SURFACE_HEIGHT,    COMMON_CODE
!SATELLITE_SSS,                 QTY_SALINITY,              COMMON_CODE
! END DART PREPROCESS KIND LIST
```

New observation types may be added to this list with no loss of generality. Supporting the observations and actually **assimilating** them are somewhat different and is controlled by the `input.nml&obs_kind_nml` assimilate_these_obs_types variable. This provides the flexibility to have an observation sequence file containing many different observation types and being able to selectively choose what types will be assimilated.

6.146.2 Other modules used

none

6.146.3 Public interfaces

none

6.146.4 Public components

none

6.146.5 Files

none

6.146.6 References

none

6.146.7 Private components

N/A

6.147 MODULE `obs_def_1d_state_mod`

6.147.1 Overview

The list of observation types to be supported by the DART executables is defined at compile time. The observations DART supports can be changed at any time by adding or removing items from the preprocess namelist and rerunning *quickbuild.csh*.

`Preprocess` takes observation specific code sections from special obs_def files to generate `obs_def_mod.f90` and `obs_kind_mod.f90` which are then compiled into `filter` and other DART programs. One of the motivations behind creating `obs_def_1d_state_mod.f90` was to provide a prototype for people developing more complicated specialized observation definition modules.

`Obs_def_1d_state_mod.f90` is an extended format Fortran 90 module that provides the definition for observation types designed for use with idealized low-order models that use the 1D location module and can be thought of as having a state vector that is equally spaced on a 1D cyclic domain. Observation types include:

- RAW_STATE_VARIABLE - A straight linear interpolation to a point on a [0,1] domain.
- RAW_STATE_VAR_POWER - The interpolated RAW_STATE_VARIABLE raised to a real-valued power.
- RAW_STATE_1D_INTEGRAL - An area-weighted ‘integral’ of the state variable over some part of the cyclic 1D domain.

RAW_STATE_VAR_POWER is convenient for studying non-gaussian, non-linear assimilation problems. RAW_STATE_VAR_POWER can be used to do idealized studies related to remote sensing observations that are best thought of as weighted integrals of some quantity over a finite volume.

The RAW_STATE_1D_INTEGRAL has an associated half_width and localization type (see the *MODULE cov_cutoff_mod* documentation) and a number of points at which to compute the associated integral by quadrature. The location of the observation defines the center of mass of the integral. The integral is centered around the location and extends outward on each side to 2*half_width. The weight associated with the integral is defined by the weight of the localization function (for instance Gaspari Cohn) using the same localization options as defined by the cov_cutoff module. The number of points are used to equally divide the range for computing the integral by quadrature.

Special observation modules like `obs_def_1d_state_mod.f90` contain Fortran 90 code *and* additional specially formatted commented code that is used to guide the preprocess program in constructing obs_def_mod.f90 and obs_kind_mod.f90. The specially formatted comments are most conveniently placed at the beginning of the module and comprise seven sections, each beginning and ending with a special F90 comment line that must be included *verbatim*.

The seven sections and their specific instances for the 1d_raw_state_mod are:

1. A list of all observation types defined by this module and their associated generic quantities (see *PROGRAM preprocess* for details on quantity files). The header line is followed by lines that have the observation type name (an all caps Fortran 90 identifier) and their associated generic quantity identifier. If there is no special processing needed for an observation type, and no additional data needed beyond the standard contents of an observation then a third word on the line, `COMMON_CODE`, will instruct the preprocess program to automatically generate all stubs and code needed for this type. For observation types needing special code or additional data, this word should not be specified and the user must supply the code manually.

```
! BEGIN DART PREPROCESS KIND LIST
! RAW_STATE_VARIABLE,     QTY_STATE_VARIABLE,    COMMON_CODE
! RAW_STATE_1D_INTEGRAL, QTY_1D_INTEGRAL
! END DART PREPROCESS KIND LIST
```

2. A list of all the use statements that the completed obs_def_mod.f90 must have in order to use the public interfaces provided by this special obs_def module. This section is optional if there are no external interfaces.

```
! BEGIN DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE
!   use obs_def_1d_state_mod, only : write_1d_integral, read_1d_integral,  &
!                                    interactive_1d_integral, get_expected_1d_
↪integral, &
!                                    set_1d_integral
! END DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE
```

3. Case statement entries for each observation type defined by this special obs_def module stating how to compute the forward observation operator. There must be a case statement entry for each type of observation, *except* for observation types defined with COMMON_CODE.

```
! BEGIN DART PREPROCESS GET_EXPECTED_OBS_FROM_DEF
!         case(RAW_STATE_1D_INTEGRAL)
```

(continues on next page)

(continued from previous page)

```
!           call get_expected_1d_integral(state, location, obs_def%key, obs_val,␣
↪istatus)
! END DART PREPROCESS GET_EXPECTED_OBS_FROM_DEF
```

4. Case statement entries for each observation type defined by this special obs_def module stating how to read any extra required information from an obs sequence file. There must be a case statement entry for each type of observation, *except* for observation types defined with COMMON_CODE. If no special action is required put a `continue` statement as the body of the case instead of a subroutine call.

```
! BEGIN DART PREPROCESS READ_OBS_DEF
!      case(RAW_STATE_1D_INTEGRAL)
!         call read_1d_integral(obs_def%key, ifile, fform)
! END DART PREPROCESS READ_OBS_DEF
```

5. Case statement entries for each observation type defined by this special obs_def module stating how to write any extra required information to an obs sequence file. There must be a case statement entry for each type of observation, *except* for observation types defined with COMMON_CODE. If no special action is required put a `continue` statement as the body of the case instead of a subroutine call.

```
! BEGIN DART PREPROCESS WRITE_OBS_DEF
!      case(RAW_STATE_1D_INTEGRAL)
!         call write_1d_integral(obs_def%key, ifile, fform)
! END DART PREPROCESS WRITE_OBS_DEF
```

6. Case statement entries for each observation type defined by this special obs_def module stating how to interactively create any extra required information. There must be a case statement entry for each type of observation, *except* for observation types defined with COMMON_CODE. If no special action is required put a `continue` statement as the body of the case instead of a subroutine call.

```
! BEGIN DART PREPROCESS INTERACTIVE_OBS_DEF
!      case(RAW_STATE_1D_INTEGRAL)
!         call interactive_1d_integral(obs_def%key)
! END DART PREPROCESS INTERACTIVE_OBS_DEF
```

7. Any executable F90 module code must be tagged with the following comments. All lines between these markers will be copied, verbatim, to obs_def_mod.f90. This section is not required if there are no observation-specific subroutines.

```
! BEGIN DART PREPROCESS MODULE CODE
module obs_def_1d_state_mod

... (module executable code)

end module obs_def_1d_state_mod
! END DART PREPROCESS MODULE CODE
```

6.147.2 Other modules used

```
types_mod
utilities_mod
location_mod (1d_location_mod_only)
time_manager_mod
assim_model_mod
cov_cutoff_mod
```

6.147.3 Public interfaces

use obs_def_mod, only :	write_1d_integral
	read_1d_integral
	interactive_1d_integral
	get_expected_1d_integral
	set_1d_integral
	write_power
	read_power
	interactive_power
	get_expected_power
	set_power

call write_1d_integral(igrkey, ifile, fform)

```
integer,           intent(in) :: igrkey
integer,           intent(in) :: ifile
character(len=*), intent(in) :: fform
```

Writes out the extra information for observation with unique identifier key for a 1d_integral observation type. This includes the half-width, localization type and number of quadrature points for this observation.

igrkey	Unique integer key associated with the 1d integral observation being processed. This is not the same as the key that all types of observations have and uniquely distinguishes all observations from each other; this is a key that is only set and retrieved by this code for 1d integral observations. It is stored in the obs_def derived type, not in the main obs_type definition.
ifile	Unit number on which observation sequence file is open
fform	String noting whether file is opened for 'formatted' or 'unformatted' IO.

call read_1d_integral(igrkey, ifile, fform)

```
integer,           intent(out) :: igrkey
integer,           intent(in)  :: ifile
character(len=*), intent(in)  :: fform
```

Reads the extra information for observation with unique identifier key for a 1d_integral observation type. This information includes the half-width, localization type and number of quadrature points for this observation. The key that is returned is uniquely associated with the definition that has been created and is used by this module to keep track of the associated parameters for this observation.

igrkey	Unique integer key associated with the observation being processed.
ifile	Unit number on which observation sequence file is open
fform	String noting whether file is opened for 'formatted' or 'unformatted' IO.

call interactive_1d_integral(igrkey)

```
integer, intent(out) :: igrkey
```

Uses input from standard in to define the characteristics of a 1D integral observation. The key that is returned is uniquely associated with the definition that has been created and can be used by this module to keep track of the associated parameters (half_width, localization option, number of quadrature points) for this key.

igrkey	Unique identifier associated with the created observation definition in the obs sequence.

call get_expected_1d_integral(state, location, igrkey, val, istatus)

```
real(r8), intent(in)               :: state
type(location_type), intent(in)  :: location
integer, intent(in)                :: igrkey
real(r8), intent(out)              :: val
integer, intent(out)               :: istatus
```

Computes the forward observation operator for a 1d integral observation. Calls the `interpolate()` routine multiple times to invoke the forward operator code in whatever model this has been compiled with.

`state`	Model state vector (or extended state vector).
`location`	Location of this observation.
`igrkey`	Unique integer key associated with this observation.
`val`	Returned value of forward observation operator.
`istatus`	Returns 0 if forward operator was successfully computed, else returns a positive value. (Negative values are reserved for system use.)

call set_1d_integral(integral_half_width, num_eval_pts, localize_type, igrkey, istatus)

```
real(r8), intent(in)   :: integral_half_width
integer,  intent(in)   :: num_eval_pts
integer,  intent(in)   :: localize_type
integer,  intent(out) :: igrkey
integer,  intent(out) :: istatus
```

Available for use by programs that create observations to set the additional metadata for these observation types. This information includes the integral half-width, localization type and number of quadrature points for this observation. The key that is returned is uniquely associated with the definition that has been created and should be set in the obs_def structure by calling `set_obs_def_key()`. This key is different from the main observation key which all observation types have. This key is unique to this observation type and is used when reading in the observation sequence to match the corresponding metadata with each observation of this type.

`integral_half_width`	Real value setting the half-width of the integral.
`num_eval_pts`	Integer, number of evaluation points. 5-20 recommended.
`localize_type`	Integer localization type: 1=Gaspari-Cohn; 2=Boxcar; 3=Ramped Boxcar
`igrkey`	Unique integer key associated with the observation being processed.
`istatus`	Return code. 0 means success, any other value is an error

call write_power(powkey, ifile, fform)

```
integer,           intent(in) :: powkey
integer,           intent(in) :: ifile
character(len=*), intent(in) :: fform
```

Writes out the extra information, the power, for observation with unique identifier key for a power observation type.

powkey	Unique integer key associated with the power observation being processed. This is not the same as the key that all types of observations have and uniquely distinguishes all observations from each other; this is a key that is only set and retrieved by this code for power observations. It is stored in the obs_def derived type, not in the main obs_type definition.
ifile	Unit number on which observation sequence file is open
fform	String noting whether file is opened for 'formatted' or 'unformatted' IO.

call read_power(powkey, ifile, fform)

```
integer,          intent(out) :: powkey
integer,          intent(in)  :: ifile
character(len=*), intent(in)  :: fform
```

Reads the extra information, the power, for observation with unique identifier key for a power observation type. The key that is returned is uniquely associated with the definition that has been created and is used by this module to keep track of the associated parameters for this observation.

powkey	Unique integer key associated with the observation being processed.
ifile	Unit number on which observation sequence file is open
fform	String noting whether file is opened for 'formatted' or 'unformatted' IO.

call interactive_power(powkey)

```
integer, intent(out) :: powkey
```

Uses input from standard in to define the characteristics of a power observation. The key that is returned is uniquely associated with the definition that has been created and can be used by this module to keep track of the associated parameter, the power, for this key.

powkey	Unique identifier associated with the created observation definition in the obs sequence.

call get_expected_power(state, location, powkey, val, istatus)

```
real(r8), intent(in)                :: state
type(location_type), intent(in)     :: location
integer, intent(in)                 :: powkey
real(r8), intent(out)               :: val
integer, intent(out)                :: istatus
```

Computes the forward observation operator for a power observation. Calls the `interpolate()` routine to invoke the forward operator code in whatever model this has been compiled with, then raises the result to the specified power associated with this powkey.

`state`	Model state vector (or extended state vector).
`location`	Location of this observation.
`powkey`	Unique integer key associated with this observation.
`val`	Returned value of forward observation operator.
`istatus`	Returns 0 if forward operator was successfully computed, else returns a positive value. (Negative values are reserved for system use.)

call set_power(power_in, powkey, istatus)

```
real(r8), intent(in)  :: power_in
integer,  intent(out) :: powkey
integer,  intent(out) :: istatus
```

Available for use by programs that create observations to set the additional metadata for these observation types. This information includes the power to which to raise the state variable. The key that is returned is uniquely associated with the definition that has been created and should be set in the obs_def structure by calling `set_obs_def_key()`. This key is different from the main observation key which all observation types have. This key is unique to this observation type and is used when reading in the observation sequence to match the corresponding metadata with each observation of this type.

`power_in`	Real value setting the power.
`powkey`	Unique integer key associated with the observation being processed.
`istatus`	Return code. 0 means success, any other value is an error

6.147.4 Namelist

This module has no namelist.

6.147.5 Files

- NONE

6.147.6 References

1. none

6.148 MODULE `obs_def_radar_mod`

6.148.1 Overview

DART radar observation module, including the observation operators for the two primary radar-observation types – Doppler velocity and reflectivity – plus other utility subroutines and functions. A number of simplifications are employed for the observation operators. Most notably, the model state is mapped to a "point" observation, whereas a real radar observation is a volumetric sample. The implications of this approximation have not been investigated fully, so in the future it might be worth developing and testing more sophisticated observation operators that produce volumetric power- weighted samples.

This module is able to compute reflectivity and precipitation fall speed (needed for computing Doppler radial velocity) from the prognostic model fields only for simple single-moment microphysics schemes such as the Kessler and Lin schemes. If a more complicated microphysics scheme is used, then reflectivity and fall speed must be accessible instead as diagnostic fields in the model state.

Author and Contact information:

- Radar Science: David Dowell, david.dowell at noaa.gov, Glen Romine, romine at ucar.edu
- DART Code: Nancy Collins, nancy at ucar.edu
- Original DART/Radar work: Alain Caya

Backward compatibility note

For users of previous versions of the radar obs_def code, here are a list of changes beginning with subversion revision 3616 which are not backward compatible:

- The namelist has changed quite a bit; some items were removed, some added, and some renamed. See the namelist documention in this file for the current item names and default values.
- Some constants which depend on the microphysics scheme have been added to the namelist to make it easier to change the values for different schemes, but the defaults have also changed. Verify they are appropriate for the scheme being used.
- The interactive create routine prompts for the beam direction differently now. It takes azimuth and elevation, and then does the trigonometry to compute the three internal values which are stored in the file. The previous version prompted for the internal values directly.
- The get_expected routines try to call the model interpolate routine for `QTY_POWER_WEIGHTED_FALL_SPEED` and `QTY_RADAR_REFLECTIVITY` values. If they are not available then the code calls the model interpolate routines for several other quantities and computes these quantities. However, this requires that the model_mod interpolate code returns gracefully if the quantity is unknown or unsupported. The previous version of the WRF model_mod code used to print an error message and stop if the quantity was unknown. The updated version in the repository which went in with this radar code has been changed to return an error status code but continue if the quantity is unknown.
- The value for gravity is currently hardcoded in this module. Previous versions of this code used the gravity constant in the DART types_mod.f90 code, but in reality the code should be using whatever value of gravity is being used in the model code. For now, the value is at least separated so users can change the value in this code if necessary.

6.148.2 Other modules used

```
types_mod
utilities_mod
location_mod (threed_sphere)
assim_model_mod
obs_kind_mod
```

6.148.3 Public interfaces

use obs_def_radar_mod, only :	read_radar_ref
	get_expected_radar_ref
	read_radial_vel
	write_radial_vel
	interactive_radial_vel
	get_expected_radial_vel
	get_obs_def_radial_vel
	set_radial_vel

Namelist interface `&obs_def_radar_mod_nml` is read from file `input.nml`.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

call read_radar_ref(obsvalue, refkey)

```
real(r8),                    intent(inout) :: obsvalue
integer,                     intent(out)   :: refkey
```

Reflectivity observations have no auxiliary data to read or write, but there are namelist options that can alter the observation value at runtime. This routine tests the observation value and alters it if required.

`obsvalue`	Observation value.
`refkey`	Set to 0 to avoid uninitialized values, but otherwise unused.

call get_expected_radar_ref(state_vector, location, ref, istatus)

```
real(r8),            intent(in)  :: state_vector(:)
type(location_type), intent(in)  :: location
real(r8),            intent(out) :: ref
integer,             intent(out) :: istatus
```

Given a location and the state vector from one of the ensemble members, compute the model-predicted radar reflectivity that would be observed at that location. The returned value is in dBZ.

If `apply_ref_limit_to_fwd_op` is .TRUE. in the namelist, reflectivity values less than `reflectivity_limit_fwd_op` will be set to `lowest_reflectivity_fwd_op`.

`state_vector`	A one dimensional representation of the model state vector
`location`	Location of this observation
`ref`	The returned radar reflectivity value
`istatus`	Returned integer status code describing problems with applying forward operator. 0 is a good value; any positive value indicates an error; negative values are reserved for internal DART use only.

call read_radial_vel(velkey, ifile [, fform])

```
integer,                      intent(out) :: velkey
integer,                      intent(in)  :: ifile
character(len=*), optional,   intent(in)  :: fform
```

Reads the additional auxiliary information associated with a radial velocity observation. This includes the location of the radar source, the beam direction, and the nyquist velocity.

`velkey`	Unique identifier associated with this radial velocity observation. In this code it is an integer index into module local arrays which hold the additional data. This routine increments it and returns the new value.
`ifile`	File unit descriptor for input file
fform	File format specifier: FORMATTED or UNFORMATTED; default FORMATTED

call write_radial_vel(velkey, ifile [, fform])

```
integer,                      intent(in) :: velkey
integer,                      intent(in) :: ifile
character(len=*), optional,   intent(in) :: fform
```

Writes the additional auxiliary information associated with a radial velocity observation. This includes the location of the radar source, the beam direction, and the nyquist velocity.

velkey	Unique identifier associated with this radial velocity observation. In this code it is an integer index into module local arrays which hold the additional data. This routine uses the value to select the appropriate data to write for this observation.
ifile	File unit descriptor for output file
fform	File format specifier: FORMATTED or UNFORMATTED; default FORMATTED

call get_obs_def_radial_vel(velkey, radar_location, beam_direction, nyquist_velocity)

```
integer,             intent(in)  :: velkey
type(location_type), intent(out) :: radar_location
real(r8),            intent(out) :: beam_direction(3)
real(r8),            intent(out) :: nyquist_velocity
```

Returns the auxiliary information associated with a given radial velocity observation.

velkey	Unique identifier associated with this radial velocity observation. In this code it is an integer index into module local arrays which hold the additional data. This routine uses the value to select the appropriate data to return.
radar_location	Location of the radar.
beam_orientation	Orientation of the radar beam at the observation location. The three values are: sin(azimuth)*cos(elevation), cos(azimuth)*cos(elevation), and sin(elevation).
nyquist_velocity	Nyquist velocity at the observation point in meters/second.

call set_radial_vel(velkey, radar_location, beam_direction, nyquist_velocity)

```
integer,             intent(out) :: velkey
type(location_type), intent(in)  :: radar_location
real(r8),            intent(in)  :: beam_direction(3)
real(r8),            intent(in)  :: nyquist_velocity
```

Sets the auxiliary information associated with a radial velocity observation. This routine increments and returns the new key associated with these values.

velkey	Unique identifier associated with this radial velocity observation. In this code it is an integer index into module local arrays which hold the additional data. This routine returns the incremented value associated with this data.
radar_location	Location of the radar.
beam_orientation	Orientation of the radar beam at the observation location. The three values are: sin(azimuth)*cos(elevation), cos(azimuth)*cos(elevation), and sin(elevation).
nyquist_velocity	Nyquist velocity at the observation point in meters/second.

call interactive_radial_vel(velkey)

```
integer, intent(out) :: velkey
```

Prompts the user for the auxiliary information needed for a radial velocity observation, and returns the new key associated with this data.

velkey	Unique identifier associated with this radial velocity observation. In this code it is an integer index into module local arrays which hold the additional data. This routine returns the incremented value associated with this data.

call get_expected_radial_vel(state_vector, location, velkey, radial_vel, istatus)

```
real(r8),            intent(in)  :: state_vector(:)
type(location_type), intent(in)  :: location
integer,             intent(in)  :: velkey
real(r8),            intent(out) :: radial_vel
integer,             intent(out) :: istatus
```

Given a location and the state vector from one of the ensemble members, compute the model-predicted radial velocity in meters/second that would be observed at that location. `velkey` is the unique index for this particular radial velocity observation. The value is returned in `radial_vel`, `istatus` is the return code.

The along-beam component of the 3-d air velocity is computed from the u, v, and w fields plus the beam_direction. The along-beam component of power-weighted precipitation fall velocity is added to the result.

state_vector	A one dimensional representation of the model state vector
location	Location of this observation
velkey	Unique identifier associated with this radial velocity observation
radial_vel	The returned radial velocity value in meters/second
istatus	Returned integer status code describing problems with applying forward operator. 0 is a good value; any positive value indicates an error; negative values are reserved for internal DART use only.

6.148.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_def_radar_mod_nml
   apply_ref_limit_to_obs      =   .false.,
   reflectivity_limit_obs      =     -10.0,
   lowest_reflectivity_obs     =     -10.0,
```

(continues on next page)

(continued from previous page)

```
   apply_ref_limit_to_fwd_op   =   .false.,
   reflectivity_limit_fwd_op   =     -10.0,
   lowest_reflectivity_fwd_op  =     -10.0,
   max_radial_vel_obs          =   1000000,
   allow_wet_graupel           =   .false.,
   microphysics_type           =       2  ,
   allow_dbztowt_conv          =   .false.,
   dielectric_factor           =     0.224,
   n0_rain                     =     8.0e6,
   n0_graupel                  =     4.0e6,
   n0_snow                     =     3.0e6,
   rho_rain                    =    1000.0,
   rho_graupel                 =     400.0,
   rho_snow                    =     100.0
   /
```

Item	Type	Description
apply_ref_limit_to_obs	logical	If .TRUE. replace all reflectivity values less than "reflectivity_limit_obs" with "lowest_reflectivity_obs" value. If .FALSE. leave all values as-is.
reflectivity_limit_obs	real(r8)	The threshold value. Observed reflectivity values less than this threshold will be set to the "lowest_reflectivity_obs" value. Units are dBZ.
lowest_reflectivity_obs	real(r8)	The 'set-to' value. Observed reflectivity values less than the threshold will be set to this value. Units are dBZ.
apply_ref_limit_to_fwd_op	logical	Same as "apply_ref_limit_to_obs", but for the forward operator.
reflectivity_limit_fwd_op	real(r8)	Same as "reflectivity_limit_obs", but for the forward operator values.
lowest_reflectivity_fwd_op	real(r8)	Same as "lowest_reflectivity_obs", but for the forward operator values.
max_radial_vel_obs	integer	Maximum number of observations of this type to support at run time. This is combined total of all obs_seq files, for example the observation diagnostic program potentially opens multiple obs_seq.final files, or the obs merge program can also open multiple obs files.
allow_wet_graupel	logical	It is difficult to predict/diagnose whether graupel/hail has a wet or dry surface. Even when the temperature is above freezing, evaporation and/or absorption can still result in a dry surface. This issue is important because the reflectivity from graupel with a wet surface is significantly greater than that from graupel with a dry surface. Currently, the user has two options for how to compute graupel reflectivity. If allow_wet_graupel is .false. (the default), then graupel is always assumed to be dry. If allow_wet_graupel is .true., then graupel is assumed to be wet (dry) when the temperature is above (below) freezing. A consequence is that a sharp gradient in reflectivity will be produced at the freezing level. In the future, it might be better to provide the option of having a transition layer.
microphysics_type	integer	If the state vector contains the reflectivity or the power weighted fall speed, interpolate directly from those regardless of the setting of this item. If the state vector does not contain the fields, this value should

6.148.5 Files

- A DART observation sequence file containing Radar obs.

6.148.6 References

- Battan, L. J., 1973: *Radar Observation of the Atmosphere.* Univ. of Chicago Press, 324 pp.
- Caya, A. *Radar Observations in Dart.* DART Subversion repository.
- Doviak, R. J., and D. S. Zrnic, 1993: *Doppler Radar and Weather Observations.* Academic Press, 562 pp.
- Ferrier, B. S., 1994: A double-moment multiple-phase four-class bulk ice scheme. Part I: Description. *J. Atmos. Sci.*, **51**, 249-280.
- Lin, Y.-L., Farley R. D., and H. D. Orville, 1983: Bulk parameterization of the snow field in a cloud model. *J. Climate Appl. Meteor.*, **22**, 1065-1092.
- Smith, P. L. Jr., 1984: Equivalent radar reflectivity factors for snow and ice particles. *J. Climate Appl. Meteor.*, 23, 1258-1260.
- Smith, P. L. Jr., Myers C. G., and H. D. Orville, 1975: Radar reflectivity factor calculations in numerical cloud models using bulk parameterization of precipitation. *J. Appl. Meteor.*, **14**, 1156-1165.

6.148.7 Private components

use obs_def_radar_mod, only :	initialize_module
	read_beam_direction
	read_nyquist_velocity
	write_beam_direction
	write_nyquist_velocity
	interactive_beam_direction
	interactive_nyquist_velocity
	get_reflectivity
	get_precip_fall_speed
	initialize_constants
	print_constants
	pr_con
	velkey_out_of_range
	check_namelist_limits
	ascii_file_format

call initialize_module()

Reads the namelist, allocates space for the auxiliary data associated wtih radial velocity observations, initializes the constants used in subsequent computations (possibly altered by values in the namelist), and prints out the list of constants and the values in use. These may need to change depending on which microphysics scheme is being used.

beam_direction = read_beam_direction(ifile, is_asciiformat)

```
real(r8), dimension(3)            :: read_beam_direction
integer,                intent(in) :: ifile
logical,                intent(in) :: is_asciiformat
```

Reads the beam direction at the observation location. Auxiliary data for doppler radial velocity observations.

read_beam_direction	Returns three real values for the radar beam orientation
ifile	File unit descriptor for input file
is_asciiformat	File format specifier: .TRUE. if file is formatted/ascii, or .FALSE. if unformatted/binary. Default .TRUE.

nyquist_velocity = read_nyquist_velocity(ifile, is_asciiformat)

```
real(r8),            :: read_nyquist_velocity
integer,  intent(in) :: ifile
logical,  intent(in) :: is_asciiformat
```

Reads nyquist velocity for a doppler radial velocity observation.

read_nyquist_velocity	Returns a real value for the nyquist velocity value
ifile	File unit descriptor for input file
is_asciiformat	File format specifier: .TRUE. if file is formatted/ascii, or .FALSE. if unformatted/binary. Default .TRUE.

call write_beam_direction(ifile, beam_direction, is_asciiformat)

```
integer,                 intent(in) :: ifile
real(r8), dimension(3),  intent(in) :: beam_direction
logical,                 intent(in) :: is_asciiformat
```

Writes the beam direction at the observation location. Auxiliary data for doppler radial velocity observations.

ifile	File unit descriptor for output file
beam_direction	Three components of the radar beam orientation
is_asciiformat	File format specifier: .TRUE. if file is formatted/ascii, or .FALSE. if unformatted/binary. Default .TRUE.

call write_nyquist_velocity(ifile, nyquist_velocity, is_asciiformat)

```
integer,  intent(in) :: ifile
real(r8), intent(in) :: nyquist_velocity
logical,  intent(in) :: is_asciiformat
```

Writes nyquist velocity for a doppler radial velocity observation.

`ifile`	File unit descriptor for output file
`nyquist_velocity`	The nyquist velocity value for this observation
`is_asciiformat`	File format specifier: .TRUE. if file is formatted/ascii, or .FALSE. if unformatted/binary. Default .TRUE.

call interactive_beam_direction(beam_direction)

```
real(r8), dimension(3), intent(out) :: beam_direction
```

Prompts the user for input for the azimuth and elevation of the radar beam at the observation location. Will be converted to the three values actually stored in the observation sequence file.

`beam_direction`	Three components of the radar beam orientation

call interactive_nyquist_velocity(nyquist_velocity)

```
real(r8), intent(out) :: nyquist_velocity
```

Prompts the user for input for the nyquist velocity value associated with a doppler radial velocity observation.

`nyquist_velocity`	Nyquist velocity value for the observation.

call get_reflectivity(qr, qg, qs, rho, temp, ref)

```
real(r8), intent(in)  :: qr
real(r8), intent(in)  :: qg
real(r8), intent(in)  :: qs
real(r8), intent(in)  :: rho
real(r8), intent(in)  :: temp
real(r8), intent(out) :: ref
```

Computes the equivalent radar reflectivity factor in mm^6 m^{-3} for simple single-moment microphysics schemes such as Kessler and Lin, et al. See the references for more details.

`qr`	Rain water content (kg kg^{-1})
`qg`	Graupel/hail content (kg kg^{-1})
`qs`	Snow content (kg kg^{-1})
`rho`	Air density (kg m^{-3})
`temp`	Air temperature (K)
`ref`	The returned radar reflectivity value

call get_precip_fall_speed(qr, qg, qs, rho, temp, precip_fall_speed)

```
real(r8), intent(in)  :: qr
real(r8), intent(in)  :: qg
real(r8), intent(in)  :: qs
real(r8), intent(in)  :: rho
real(r8), intent(in)  :: temp
real(r8), intent(out) :: precip_fall_speed
```

Computes power-weighted precipitation fall speed in m s^{-1} for simple single-moment microphysics schemes such as Kessler and Lin, et al. See the references for more details.

qr	Rain water content (kg kg^{-1})
qg	Graupel/hail content (kg kg^{-1})
qs	Snow content (kg kg^{-1})
rho	Air density (kg m^{-3})
temp	Air temperature (K)
precip_fall_speed	The returned precipitation vall speed

call initialize_constants()

Set values for a collection of constants used throughout the module during the various calculations. These are set once in this routine and are unchanged throughout the rest of the execution. They cannot be true Fortran `parameters` because some of the values can be overwritten by namelist entries, but once they are set they are treated as read-only parameters.

call print_constants()

Print out the names and values of all constant parameters used by this module. The error handler message facility is used to print the message, which by default goes to both the DART log file and to the standard output of the program.

call pr_con(c_val, c_str)

```
real(r8),         intent(in)  :: c_val
character(len=*), intent(in)  :: c_str
```

Calls the DART error handler routine to print out a string label and a real value to both the log file and to the standard output.

`Value of constant`	A real value.
`Name of constant`	A character string.

call velkey_out_of_range(velkey)

```
integer, intent(in)  :: velkey
```

Range check key and trigger a fatal error if larger than the allocated array for observation auxiliary data.

`velkey`	Integer key into a local array of auxiliary observation data.

call check_namelist_limits(apply_ref_limit_to_obs, reflectivity_limit_obs, lowest_reflectivity_obs, apply_ref_limit_to_fwd_op, reflectivity_limit_fwd_op, lowest_reflectivity_fwd_op)

```
logical,  intent(in) :: apply_ref_limit_to_obs
real(r8), intent(in) :: reflectivity_limit_obs
real(r8), intent(in) :: lowest_reflectivity_obs
logical,  intent(in) :: apply_ref_limit_to_fwd_op
real(r8), intent(in) :: reflectivity_limit_fwd_op
real(r8), intent(in) :: lowest_reflectivity_fwd_op
```

Check the values set in the namelist for consistency. Print out a message if the limits and set-to values are different; this may be intentional but is not generally expected to be the case. In all cases below, see the namelist documentation for a fuller explanation of each value.

`apply_ref_limit_to_obs`	Logical. See namelist.
`reflectivity_limit_obs`	Real value. See namelist.
`lowest_reflectivity_obs`	Real value. See namelist.
`apply_ref_limit_to_fwd_op`	Logical. See namelist.
`reflectivity_limit_fwd_op`	Real value. See namelist.
`lowest_reflectivity_fwd_op`	Real value. See namelist.

is_asciifile = ascii_file_format(fform)

```
logical                                   :: ascii_file_format
character(len=*), intent(in), optional :: fform
```

Should be moved to DART utility module at some point. Returns .TRUE. if the optional argument is missing or if it is not one of the following values: `"unformatted", "UNFORMATTED", "unf", "UNF"`.

ascii_file_format	Return value. Logical. Default is .TRUE.
fform	Character string file format.

6.149 MODULE DEFAULT_obs_def_mod

6.149.1 Overview

DEFAULT_obs_def.F90 is a template used by the program preprocess to create obs_def_mod.f90.

To read more detailed instructions on how to add new observation types, see the documentation for *MODULE obs_def_mod*. obs_def_*_mod.f90 files are specified as input to the preprocess program by namelist, and a new obs_def_mod.f90 file is generated which contains all the selected observation types.

Information from zero or more special obs_def modules, such as obs_def_1d_state_mod.f90 or obs_def_reanalyis_bufr_mod.f90, (also documented in this directory) are incorporated into the DEFAULT_obs_def_mod.F90 template by preprocess. If no special obs_def files are included in the preprocess namelist, a minimal obs_def_mod.f90 is created which can only support identity forward observation operators. Any identity observations on the obs_seq.out file will be assimilated, regardless of the obs types specified in assimilate_these_obs_types.

The documentation below describes the special formatting that is included in the DEFAULT_obs_def_mod.F90 in order to guide the preprocess program.

Up to seven sections of code are inserted into DEFAULT_obs_def_mod.F90 from each of the special obs_def_*_mod.f90 files. The insertion point for each section is denoted by a special comment line that must be included *verbatim* in DEFAULT_obs_def_mod.F90. These special comment lines and their significance are:

1. ! DART PREPROCESS MODULE CODE INSERTED HERE Some special observation definition modules (see for instance obs_def_1d_state_mod.f90) contain code for evaluating forward observation operators, reading or writing special information about an observation definition to an obs sequence file, or for interactive definition of an observation. The entire module code section is inserted here, so the resulting output file will be completely self-contained. Fortran 90 allows multiple modules to be defined in a single source file, and subsequent module code can use previously defined modules, so this statement must preceed the rest of the other comment lines.

2. ! DART PREPROCESS USE FOR OBS_QTY_MOD INSERTED HERE The quantities available to DART are defined by passing quantity files from DART/assimilation_code/modules/observations to preprocess. Unique integer values for each quantity are assigned by preprocess and the use statements for these entries are inserted here.

3. ! DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE INSERTED HERE Some special observation definition modules (see for instance obs_def_1d_state_mod.f90) contain code for evaluating forward observation operators, reading or writing special information about an observation definition to an obs sequence file, or for interactive definition of an observation. The use statements for these routines from the special observation definition modules are inserted here.

4. ! DART PREPROCESS GET_EXPECTED_OBS_FROM_DEF INSERTED HERE Special observation definition modules must contain case statement code saying what to do to evaluate a forward observation operator for each observation type that they define. This code is inserted here.

5. `! DART PREPROCESS READ_OBS_DEF INSERTED HERE` Special observation definition modules must contain case statement code saying what to do to read any additional information required for each observation type that they define from an observation sequence file. This code is inserted here.
6. `! DART PREPROCESS WRITE_OBS_DEF INSERTED HERE` Special observation definition modules must contain case statement code saying what to do to write any additional information required for each observation type that they define to an observation sequence file. This code is inserted here.
7. `! DART PREPROCESS INTERACTIVE_OBS_DEF INSERTED HERE` Special observation definition modules must contain case statement code saying what to do to interactively create any additional information required for each observation type that they define. This code is inserted here.

6.150 MODULE obs_def_mod

6.150.1 Overview

The DART Fortran90 derived type `obs_def` provide an abstraction of the definition of an observation. An observation sequence `obs_seq` at a higher level is composed of observation definitions associated with observed values. For now, the basic operations required to implement an observation definition are an ability to compute a forward operator given the model state vector, the ability to read/write the observation definition from/to a file, and a capability to do a standard input driven interactive definition of the observation definition.

DART makes a distinction between specific `observation types` and generic `observation quantities`. The role of the various obs_def input files is to define the mapping between the types and quantities, and optionally to provide type-specific processing routines.

A single obs_def output module is created by the program `preprocess` from two kinds of input files. First, a DEFAULT obs_def module (normally called `DEFAULT_obs_def_mod.F90` and documented in this directory) is used as a template into which the preprocessor incorporates information from zero or more special obs_def modules (such as `obs_def_1d_state_mod.f90` or `obs_def_reanalysis_bufr_mod.f90`, also documented in this directory). If no special obs_def files are included in the preprocessor namelist, a minimal `obs_def_mod.f90` is created which can only support identity forward observation operators.

To add a new observation type which does not fit into any of the already-defined obs_def files, a new file should be created in the `obs_def` directory. These files are usually named according the the pattern `obs_def_X_mod.f90`, where the X is either an instrument name, a data source, or a class of observations. See the existing filenames in that directory for ideas. Then this new filename must be listed in the `input.nml` namelist for the model, in the `&preprocess_nml` section, in the `obs_type_files` variable. This variable is a string list type which can contain multiple filenames. Running the `preprocess` program will then use the contents of the new file to generate the needed output files for use in linking to the rest of the DART system.

Simple observations

If the new observation type can be directly interpolated by a model_mod interpolation routine, and has no additional observation-specific code for reading, writing, or initializing the observation, then the entire contents of the new file is:

```
! BEGIN DART PREPROCESS TYPE DEFINITIONS
! type, quantity, COMMON_CODE
! (repeat lines for each type)
! END DART PREPROCESS TYPE DEFINITIONS
```

DART will automatically generate all interface code needed for these new observation types. For example, here is a real list:

```
! BEGIN DART PREPROCESS TYPE DEFINITIONS
!VELOCITY,                     QTY_VELOCITY,               COMMON_CODE
!TRACER_CONCENTRATION,         QTY_TRACER_CONCENTRATION,   COMMON_CODE
!TRACER_SOURCE,                QTY_TRACER_SOURCE,          COMMON_CODE
!MEAN_SOURCE,                  QTY_MEAN_SOURCE,            COMMON_CODE
!SOURCE_PHASE,                 QTY_SOURCE_PHASE,           COMMON_CODE
! END DART PREPROCESS TYPE DEFINITIONS
```

The first column is the specific observation `type` and should be unique. The second column is the generic observation `quantity`. The quantities available to DART are defined at compile time by *preprocess* via the option 'quantity_files' in the *preprocess_nml* namelist. The third column must be the keyword `COMMON_CODE` which tells the `preprocess` program to automatically generate all necessary interface code for this type.

Observations needing special handling

For observation types which have observation-specific routines, must interpolate using a combination of other generic quantities, or require additional observation-specific data to be stored, the following format is used:

```
! BEGIN DART PREPROCESS TYPE DEFINITIONS
! type, quantity
! (repeat lines for each type/quantity pair)
! END DART PREPROCESS TYPE DEFINITIONS
```

DART will need user-supplied interface code for each of the listed types. For example, here is a real list:

```
! BEGIN DART PREPROCESS TYPE DEFINITIONS
! DOPPLER_RADIAL_VELOCITY, QTY_VELOCITY
! RADAR_REFLECTIVITY,      QTY_RADAR_REFLECTIVITY
! END DART PREPROCESS TYPE DEFINITIONS
```

In this case, DART needs additional information for how to process these types. They include code sections delimited by precisely formatted comments, and possibly module code sections:

1.
   ```
   ! BEGIN DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE
   ! END DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE
   ```

   Any fortran use statements for public subroutines or variables from other modules should be placed between these lines, with comment characters in the first column.
   For example, if the forward operator code includes a module with public routines then a "use" statement like:

   ```
   use obs_def_1d_state_mod, only : write_1d_integral, read_1d_integral, &
                                    interactive_1d_integral, get_expected_1d_integral
   ```

   needs to be added to the obs_def_mod so the listed subroutines are available to be called. This would look like:

   ```
   ! BEGIN DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE
   ! use obs_def_1d_state_mod, only : write_1d_integral, read_1d_integral, &
   !                                  interactive_1d_integral, get_expected_1d_
   ↪integral
   ! END DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE
   ```

2.
```
! BEGIN DART PREPROCESS GET_EXPECTED_OBS_FROM_DEF
! END DART PREPROCESS GET_EXPECTED_OBS_FROM_DEF
```

These comments must enclose a case statement for each defined type that returns the expected observation value based on the current values of the state vector. The code must be in comments, with the comment character in the first column.

The variables available to be passed to subroutines or used in this section of code are:

`state`	the entire model state vector
`state_time`	the time of the state data
`ens_index`	the ensemble member number
`location`	the observation location
`obs_kind_ind`	the index of the specific observation type
`obs_time`	the time of the observation
`error_val`	the observation error variance

The routine must fill in the values of these variables:

`obs_val`	the computed forward operator value
`istatus`	return code: 0=ok, >0 is error, <0 reserved for system use

To call a model_mod interpolate routine directly, the argument list must match exactly:

```
interpolate(state, location, QTY_xxx, obs_val, istatus)
```

This can be useful if the forward operator needs to retrieve values for fields which are typically found in a model and then compute a derived value from them.

3.
```
! BEGIN DART PREPROCESS READ_OBS_DEF
! END DART PREPROCESS READ_OBS_DEF
```

These comments must enclose a case statement for each defined type that reads any additional data associated with a single observation. If there is no information beyond that for the basic obs_def type, the case statement must still be provided, but the code can simply be `continue`. The code must be in comments, with the comment character in the first column.

The variables available to be passed to subroutines or used in this section of code are:

`ifile`	the open unit number positioned ready to read, read-only
`obs_def`	the rest of the obs_def derived type for this obs, read-write
`key`	the index observation number in this sequence, read-only
`obs_val`	the observation value, if needed. in general should not be changed
`is_ascii`	logical to indicate how the file was opened, formatted or unformatted

The usual use of this routine is to read in additional metadata per observation and to set the private key in the `obs_def` to indicate which index to use for this observation to look up the corresponding metadata in arrays or derived types. Do not confuse the key in the obs_def with the key argument to this routine; the latter is the global observation sequence number for this observation.

4.
```
! BEGIN DART PREPROCESS WRITE_OBS_DEF
! END DART PREPROCESS WRITE_OBS_DEF
```

These comments must enclose a case statement for each defined type that writes any additional data associated with a single observation. If there is no information beyond that for the basic obs_def type, the case statement must still be provided, but the code can simply be `continue`. The code must be in comments, with the comment character in the first column.

The variables available to be passed to subroutines or used in this section of code are:

`ifile`	the open unit number positioned ready to write, read-only
`obs_def`	the rest of the obs_def derived type for this obs, read-only
`key`	the index observation number in this sequence, read-only
`is_ascii`	logical to indicate how the file was opened, formatted or unformatted

The usual use of this routine is to write the additional metadata for this observation based on the private key in the `obs_def`. Do not confuse this with the key in the subroutine call which is the observation number relative to the entire observation sequence file.

5.
```
! BEGIN DART PREPROCESS INTERACTIVE_OBS_DEF
! END DART PREPROCESS INTERACTIVE_OBS_DEF
```

These comments must enclose a case statement for each defined type that prompts the user for any additional data associated with a single observation. If there is no information beyond that for the basic obs_def type, the case statement must still be provided, but the code can simply be `continue`. The code must be in comments, with the comment character in the first column.

The variables available to be passed to subroutines or used in this section of code are:

obs_def	the rest of the obs_def derived type for this obs, read-write
key	the index observation number in this sequence, read-only

The DART code will prompt for the rest of the obs_def values (location, type, value, error) but any additional metadata needed by this observation type should be prompted to, and read from, the console (e.g. `write(*,*)`, and `read(*, *)`). The code will generally set the `obs_def%key` value as part of setting the metadata.

6.
```
! BEGIN DART PREPROCESS MODULE CODE
! END DART PREPROCESS MODULE CODE
```

If the code to process this observation requires module data and/or subroutines, then these comments must surround the module definitions. Unlike all the other sections, this comment pair is optional, and if used, the code must not be in comments; it will be copied verbatim over to the output file.

Generally the code for a forward operator should be defined inside a module, to keep module variables and other private subroutines from colliding with unrelated routines and variables in other forward operator files.

It is possible to mix automatic code types and user-supplied code types in the same list. Simply add the COMMON_CODE keyword on the lines which need no special data or interfaces. For example, here is an extract from the 1d state obs_def module, where the raw state variable needs only autogenerated code, but the 1d integral has user-supplied processing code:

```
! BEGIN DART PREPROCESS TYPE LIST
! RAW_STATE_VARIABLE,    QTY_STATE_VARIABLE, COMMON_CODE
! RAW_STATE_1D_INTEGRAL, QTY_1D_INTEGRAL
! END DART PREPROCESS TYPE LIST


! BEGIN DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE
!   use obs_def_1d_state_mod, only : write_1d_integral, read_1d_integral, &
!                                    interactive_1d_integral, get_expected_1d_integral
! END DART PREPROCESS USE OF SPECIAL OBS_DEF MODULE

! BEGIN DART PREPROCESS GET_EXPECTED_OBS_FROM_DEF
!         case(RAW_STATE_1D_INTEGRAL)
!            call get_expected_1d_integral(state, location, obs_def%key, obs_val,␣
↪istatus)
! END DART PREPROCESS GET_EXPECTED_OBS_FROM_DEF

! BEGIN DART PREPROCESS READ_OBS_DEF
!      case(RAW_STATE_1D_INTEGRAL)
!         call read_1d_integral(obs_def%key, ifile, fileformat)
! END DART PREPROCESS READ_OBS_DEF

! BEGIN DART PREPROCESS WRITE_OBS_DEF
!      case(RAW_STATE_1D_INTEGRAL)
!         call write_1d_integral(obs_def%key, ifile, fileformat)
! END DART PREPROCESS WRITE_OBS_DEF
```

(continues on next page)

(continued from previous page)

```
! BEGIN DART PREPROCESS INTERACTIVE_OBS_DEF
!      case(RAW_STATE_1D_INTEGRAL)
!         call interactive_1d_integral(obs_def%key)
! END DART PREPROCESS INTERACTIVE_OBS_DEF

! BEGIN DART PREPROCESS MODULE CODE
module obs_def_1d_state_mod

use        types_mod, only : r8
use    utilities_mod, only : register_module, error_handler, E_ERR, E_MSG
use     location_mod, only : location_type, set_location, get_location
use  assim_model_mod, only : interpolate
use   cov_cutoff_mod, only : comp_cov_factor

implicit none

public :: write_1d_integral, read_1d_integral, interactive_1d_integral, &
          get_expected_1d_integral

...  (module code here)

end module obs_def_1d_state_mod
! END DART PREPROCESS MODULE CODE
```

See the *MODULE obs_def_1d_state_mod* documentation for more details and examples of each section. Also see obs_def_wind_speed_mod.f90 for an example of a 3D geophysical forward operator.

In addition to collecting and managing any additional observation type-specific code, this module provides the definition of the obs_def_type derived type, and a collection of subroutines for creating, accessing, and updating this type. The remainder of this document describes the subroutines provided by this module.

6.150.2 Other modules used

```
types_mod
utilities_mod
location_mod (depends on model choice)
time_manager_mod
assim_model_mod
obs_kind_mod
Other special obs_def_kind modules as required
```

6.150.3 Public interfaces

use obs_def_mod, only :	obs_def_type
	init_obs_def
	get_obs_def_location
	get_obs_def_type_of_obs
	get_obs_def_time
	get_obs_def_error_variance
	get_obs_def_key
	set_obs_def_location
	set_obs_def_type_of_obs
	set_obs_def_time
	set_obs_def_error_variance
	set_obs_def_key
	interactive_obs_def
	write_obs_def
	read_obs_def
	get_expected_obs_from_def
	destroy_obs_def
	copy_obs_def
	assignment(=)
	get_name_for_type_of_obs

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

```
type obs_def_type
   private
   type(location_type)  :: location
```

(continues on next page)

(continued from previous page)

```
   integer                 :: kind
   type(time_type)         :: time
   real(r8)                :: error_variance
   integer                 :: key
end type obs_def_type
```

Models all that is known about an observation except for actual values. Includes a location, type, time and error variance.

Component	Description
location	Location of the observation.
kind	Despite the name, the specific type of the observation.
time	Time of the observation.
error_variance	Error variance of the observation.
key	Unique identifier for observations of a particular type.

call init_obs_def(obs_def, location, kind, time, error_variance)

```
type(obs_def_type),  intent(out) :: obs_def
type(location_type), intent(in)  :: location
integer,             intent(in)  :: kind
type(time_type),     intent(in)  :: time
real(r8),            intent(in)  :: error_variance
```

Creates an obs_def type with location, type, time and error_variance specified.

`obs_def`	The obs_def that is created
`location`	Location for this obs_def
`kind`	Observation type for obs_def
`time`	Time for obs_def
`error_variance`	Error variance of this observation

call copy_obs_def(obs_def1, obs_def2)

```
type(obs_def_type), intent(out) :: obs_def1
type(obs_def_type), intent(in)  :: obs_def2
```

Copies obs_def2 to obs_def1, overloaded as assignment (=).

`obs_def1`	obs_def to be copied into
`obs_def2`	obs_def to be copied from

var = get_obs_def_key(obs_def)

```
integer                          :: get_obs_def_key
type(obs_def_type), intent(in)   :: obs_def
```

Returns key from an observation definition.

`var`	Returns key from an obs_def
`obs_def`	An obs_def

var = get_obs_def_error_variance(obs_def)

```
real(r8)                         :: get_obs_def_error_variance
type(obs_def_type), intent(in)   :: obs_def
```

Returns error variance from an observation definition.

`var`	Error variance from an obs_def
`obs_def`	An obs_def

var = get_obs_def_location(obs_def)

```
type(location_type)                :: get_obs_def_location
type(obs_def_type), intent(in)     :: obs_def
```

Returns the location from an observation definition.

`var`	Returns location from an obs_def
`obs_def`	An obs_def

var = get_obs_def_type_of_obs(obs_def)

```
integer                           :: get_obs_def_type_of_obs
type(obs_def_type),  intent(in)   :: obs_def
```

Returns an observation type from an observation definition.

`var`	Returns the observation type from an obs_def
`obs_def`	An obs_def

var = get_obs_def_time(obs_def)

```
type(time_type)                :: get_obs_def_time
type(obs_def_type), intent(in) :: obs_def
```

Returns time from an observation definition.

`var`	Returns time from an obs_def
`obs_def`	An obs_def

obs_name = get_name_for_type_of_obs(obs_kind_ind)

```
character(len = 32)            :: get_name_for_type_of_obs
integer, intent(in)            :: obs_kind_ind
```

Returns an observation name from an observation type.

`var`	Returns name from an observation type
`obs_kind_ind`	An observation type

call set_obs_def_location(obs_def, location)

```
type(obs_def_type),  intent(inout) :: obs_def
type(location_type), intent(in)    :: location
```

Set the location in an observation definition.

`obs_def`	An obs_def
`location`	A location

call set_obs_def_error_variance(obs_def, error_variance)

```
type(obs_def_type), intent(inout) :: obs_def
real(r8), intent(in)              :: error_variance
```

Set error variance for an observation definition.

`obs_def`	An obs_def
`error_variance`	Error variance

call set_obs_def_key(obs_def, key)

```
type(obs_def_type), intent(inout) :: obs_def
integer,            intent(in)    :: key
```

Set the key for an observation definition.

obs_def	An obs_def
key	Unique identifier for this observation

call set_obs_def_type_of_obs(obs_def, kind)

```
type(obs_def_type), intent(inout) :: obs_def
integer,            intent(in)    :: kind
```

Set the type of observation in an observation definition.

obs_def	An obs_def
kind	An integer observation type

call set_obs_def_time(obs_def, time)

```
type(obs_def_type), intent(inout) :: obs_def
type(time_type), intent(in)       :: time
```

Sets time for an observation definition.

obs_def	An obs_def
time	Time to set

call get_expected_obs_from_def(key, obs_def, obs_kind_ind, ens_index, state, state_time, obs_val, istatus, assimilate_this_ob, evaluate_this_ob)

```
integer,            intent(in)  :: key
type(obs_def_type), intent(in)  :: obs_def
integer,            intent(in)  :: obs_kind_ind
integer,            intent(in)  :: ens_index
real(r8),           intent(in)  :: state(:)
```

(continues on next page)

(continued from previous page)

```
type(time_type),     intent(in)  :: state_time
real(r8),            intent(out) :: obs_val
integer,             intent(out) :: istatus
logical,             intent(out) :: assimilate_this_ob
logical,             intent(out) :: evaluate_this_ob
```

Compute the observation (forward) operator for a particular obs definition.

`key`	descriptor for observation type
`obs_def`	The input obs_def
`obs_kind_ind`	The obs type
`ens_index`	The ensemble member number of this state vector
`state`	Model state vector
`state_time`	Time of the data in the model state vector
`istatus`	Returned integer describing problems with applying forward operator (0 == OK, >0 == error, <0 reserved for sys use).
`assimilate_this_ob`	Indicates whether to assimilate this obs or not
`evaluate_this_ob`	Indicates whether to evaluate this obs or not

call read_obs_def(ifile, obs_def, key, obs_val [,fform])

```
integer,                        intent(in)    :: ifile
type(obs_def_type),             intent(inout) :: obs_def
integer,                        intent(in)    :: key
real(r8),                       intent(inout) :: obs_val
character(len=*), optional,     intent(in)    :: fform
```

Reads an obs_def from file open on channel ifile. Uses format specified in fform or FORMATTED if fform is not present.

`ifile`	File unit open to output file
`obs_def`	Observation definition to be read
`key`	Present if unique identifier key is needed by some obs type. Unused by default code.
`obs_val`	Present if needed to perform operations based on value. Unused by default code.
`fform`	File format specifier: FORMATTED or UNFORMATTED; default FORMATTED (FORMATTED in this case is the human readable/text option as opposed to UNFORMATTED which is binary.)

call interactive_obs_def(obs_def, key)

```
type(obs_def_type), intent(inout) :: obs_def
integer,            intent(in)    :: key
```

Creates an obs_def via input from standard in.

`obs_def`	An obs_def to be created
`key`	Present if unique identifier key is needed by some obs type. Unused by default code.

call write_obs_def(ifile, obs_def, key [,fform])

```
integer,                      intent(in) :: ifile
type(obs_def_type),           intent(in) :: obs_def
integer,                      intent(in) :: key
character(len=*), optional, intent(in) :: fform
```

Writes an obs_def to file open on channel ifile. Uses format specified in fform or FORMATTED if fform is not present.

`ifile`	File unit open to output file
`obs_def`	Observation definition to be written
`key`	Present if unique identifier key is needed by some obs type. Unused by default code.
`fform`	File format specifier: FORMATTED or UNFORMATTED; default FORMATTED

call destroy_obs_def(obs_def)

```
type(obs_def_type), intent(inout) :: obs_def
```

Releases all storage associated with an obs_def and its subcomponents.

`obs_def`	An obs_def to be released.

6.150.4 Files

- The read_obs_def() and write_obs_def() routines are passed an already-opened file channel/descriptor and read to or write from it.

6.150.5 References

- none

6.150.6 Private components

N/A

6.151 MODULE `obs_def_rttov_mod`

6.151.1 Overview

DART RTTOV observation module, including the observation operators for the two primary RTTOV-observation types – visible/infrared radiances and microwave radiances/brightness temperatures.

This module acts as a pass-through for RTTOV version 12.3. For more information, see the RTTOV site.

DART supports both RTTOV-direct for visible/infrared/microwave as well as RTTOV-scatt for microwave computations. The code, in principle, supports all features of version 12.3 as a pass-through from the model to RTTOV, includes aerosols, trace gases, clouds, and atmospheric variables. The code also includes directly specifying scattering properties.

However, a model may not have all of the variables necessary for these functions depending on your model's setup.

For example, DART can use any of the RTTOV clw or ice schemes, but the WRF model is not directly compatible with the IR default cloud classification of marine/continental stratus/cumulus clean/dirty. We also offer a simple classification based on maximum vertical velocity in the column and land type, but due to lack of aerosol information, WRF/DART cannot differentiate between clean and dirty cumulus. This may have some impact on the forward calculations - but in experience the difference in cloud phase (ice versus water) makes a much larger difference.

Trace gases and aerosols may be important for actual observation system experiments using visible/infrared; this may depend on the precise frequencies you wish to use. Although a model may not have the necessary inputs by itself, although the defaults in RTTOV based on climatology can be used. The impact on the quality of the results should be investigated.

Known issues:

- DART does not yet provide any type of bias correction
- Cross-channel error correlations are not yet supported. A principal component approach has been discussed. For now, the best bet is to use a subset of channels that are nearly independent of one another.
- Vertical localization will need to be tuned. Turning off vertical localization may work well if you have a large number of ensemble members. Using the maximum peak of the weighting function or the cloud-top may be appropriate. There are also other potential approaches being investigated.

Author and Contact information:

- DART Code: Jeff Steward, jsteward at ucar.edu
- Original DART/RTTOV work: Nancy Collins, Johnny Hendricks

Backward compatibility note

6.151.2 Other modules used

```
types_mod
utilities_mod
location_mod (threed_sphere)
assim_model_mod
obs_def_utilitie_mod
ensemble_manager_mod
utilities_mod
parkind1 (from RTTOV)
rttov_types (from RTTOV)
obs_kind_mod
```

6.151.3 Public interfaces

use obs_def_rttov_mod, only :	set_visir_metadata
	set_mw_metadata
	get_expected_radiance
	get_rttov_option_logical

Namelist interface `&obs_def_rttov_mod_nml` is read from file `input.nml`.

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

call set_visir_metadata(key, sat_az, sat_ze, sun_az, sun_ze, & platform_id, sat_id, sensor_id, channel, specularity)

```
integer,  intent(out) :: key
real(r8), intent(in)  :: sat_az
real(r8), intent(in)  :: sat_ze
real(r8), intent(in)  :: sun_az
real(r8), intent(in)  :: sun_ze
integer,  intent(in)  :: platform_id, sat_id, sensor_id, channel
real(r8), intent(in)  :: specularity
```

Visible / infrared observations have several auxillary metadata variables. Other than the key, which is standard DART fare, the RTTOV satellite azimuth and satellite zenith angle must be specified. See the RTTOV user guide for more information (in particular, see figure 4). If the `addsolar` namelist value is set to true, then the solar azimuth and solar zenith angles must be specified - again see the RTTOV user guide. In addition to the platform/satellite/ sensor ID numbers, which are the RTTOV unique identifiers, the channel specifies the chanenl number in the RTTOV coefficient file. Finally, if `do_lambertian` is true, specularity must be specified here. Again, see the RTTOV user guide for more information.

key	The DART observation key.
sat_az	The satellite azimuth angle.
sat_ze	The satellite zenith angle.
sun_az	The solar azimuth angle. Only relevant if addsolar is true.
sun_ze	The solar zenith angle. Only relevant if addsolar is true.
platform_id	The RTTOV platform ID.
sat_id	The RTTOV satellite ID.
sensor_id	The RTTOV sensor ID.
channel	The RTTOV channel number.
specularity	The surface specularity. Only relevant if do_lambertian is true.

call set_mw_metadata(key, sat_az, sat_ze, platform_id, sat_id, sensor_id, channel, mag_field, cosbk, fastem_p1, fastem_p2, fastem_p3, fastem_p4, fastem_p5)

```
integer,  intent(out) :: key
real(r8), intent(in)  :: sat_az
real(r8), intent(in)  :: sat_ze
integer,  intent(in)  :: platform_id, sat_id, sensor_id, channel
real(r8), intent(in)  :: mag_field
real(r8), intent(in)  :: cosbk
real(r8), intent(in)  :: fastem_p[1-5]
```

Microwave observations have several auxillary metadata variables. Other than the key, which is standard DART fare, the RTTOV satellite azimuth and satellite zenith angle must be specified. See the RTTOV user guide for more information (in particular, see figure 4). In addition to the platform/satellite/ sensor ID numbers, which are the RTTOV unique identifiers, the channel specifies the chanenl number in the RTTOV coefficient file. In addition, if `use_zeeman` is true, the magnetic field and cosine of the angle between the magnetic field and angle of propagation must be specified. See the RTTOV user guide for more information. Finally, the fastem parameters for land must be specified here. This may be difficult for observations to set, so default values (see table 21 in the RTTOV user guide) can be used until a better solution is devised.

key	The DART observation key.
sat_az	The satellite azimuth angle.
sat_ze	The satellite zenith angle.
platform_id	The RTTOV platform ID.
sat_id	The RTTOV satellite ID.
sensor_id	The RTTOV sensor ID.
channel	The RTTOV channel number.
mag_field	The strength of the magnetic field. Only relevant if add_zeeman is true.
cosbk	The cosine of the angle between the magnetic field and direction of EM propagation. Only relevant if add_zeeman is true.
fastem_p[1-5]	The five parameters used for fastem land/sea ice emissivities. For ocean emissivities, an internal model is used based on the value of fastem_version.

call get_expected_radiance(obs_kind_ind, state_handle, ens_size, location, key, val, istatus)

```
integer,              intent(in)  :: obs_kind_ind
type(ensemble_type),  intent(in)  :: state_handle
integer,              intent(in)  :: ens_size
type(location_type),  intent(in)  :: location
integer,              intent(in)  :: key
real(r8),             intent(out) :: val(ens_size)
integer,              intent(out) :: istatus(ens_size)
```

Given a location and the state vector from one of the ensemble members, compute the model-predicted satellite observation. This can be either in units of radiance (mW/cm-1/sr/sq.m) or a brightness temperature (in K), depending on if this is a visible/infrared observation or a microwave observation.

`obs_kind_ind`	The index of the observation kind; since many observation kinds are handled by this module, this can be used to determine precisely which observation kind is being used.
`state_handle`	The ensemble of model states to be used for the observation operator calculations.
`location`	Location of this observation
`key`	Unique identifier associated with this satellite observation
`val`	The returned observation in units of either radiance or brightness temperature.
`istatus`	Returned integer status code describing problems with applying forward operator. 0 is a good value; any positive value indicates an error; negative values are reserved for internal DART use only.

p = get_rttov_option_logical(field_name)

```
character(len=*),            intent(in)   :: field_name
logical,                     result       :: p
```

Return the logical value of the RTTOV parameter associated with the field_name.

`field_name`	The name of the RTTOV parameter from the namelist.
`p`	The logical return value associated with the parameter.

6.151.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&obs_def_rttov_nml
   rttov_sensor_db_file   = 'rttov_sensor_db.csv'
   first_lvl_is_sfc       = .true.
   mw_clear_sky_only      = .false.
   interp_mode            = 1
   do_checkinput          = .true.
   apply_reg_limits       = .true.
   verbose                = .true.
```

(continues on next page)

(continued from previous page)

```
   fix_hgpl                = .false.
   do_lambertian           = .false.
   lambertian_fixed_angle  = .true.
   rad_down_lin_tau        = .true.
   use_q2m                 = .true.
   use_uv10m               = .true.
   use_wfetch              = .false.
   use_water_type          = .false.
   addrefrac               = .false.
   plane_parallel          = .false.
   use_salinity            = .false.
   apply_band_correction   = .true.
   cfrac_data              = .true.
   clw_data                = .true.
   rain_data               = .true.
   ciw_data                = .true.
   snow_data               = .true.
   graupel_data            = .true.
   hail_data               = .false.
   w_data                  = .true.
   clw_scheme              = 1
   clw_cloud_top           = 322.
   fastem_version          = 6
   supply_foam_fraction    = .false.
   use_totalice            = .true.
   use_zeeman              = .false.
   cc_threshold            = 0.05
   ozone_data              = .false.
   co2_data                = .false.
   n2o_data                = .false.
   co_data                 = .false.
   ch4_data                = .false.
   so2_data                = .false.
   addsolar                = .false.
   rayleigh_single_scatt   = .true.
   do_nlte_correction      = .false.
   solar_sea_brdf_model    = 2
   ir_sea_emis_model       = 2
   use_sfc_snow_frac       = .false.
   add_aerosl              = .false.
   aerosl_type             = 1
   add_clouds              = .true.
   ice_scheme              = 1
   use_icede               = .false.
   idg_scheme              = 2
   user_aer_opt_param      = .false.
   user_cld_opt_param      = .false.
   grid_box_avg_cloud      = .true.
   cldstr_threshold        = -1.0
   cldstr_simple           = .false.
   cldstr_low_cloud_top    = 750.0
   ir_scatt_model          = 2
   vis_scatt_model         = 1
   dom_nstreams            = 8
   dom_accuracy            = 0.0
   dom_opdep_threshold     = 0.0
   addpc                   = .false.
```

(continues on next page)

(continued from previous page)

```
   npcscores              = -1
   addradrec              = .false.
   ipcreg                 = 1
   use_htfrtc             = .false.
   htfrtc_n_pc            = -1
   htfrtc_simple_cloud    = .false.
   htfrtc_overcast        = .false.
/
```

1

Item	Type	Description
rttov_sensor_db_file	character(len=512)	The location of the RTTOV sensor database. The format for the database is a comma-
first_lvl_is_sfc	logical	Whether the first level of the model represents the surface (true) or the top of the atm
mw_clear_sky_only	logical	If microwave calculations should be “clear-sky” only (although cloud-liquid water ab
interp_mode	integer	The interpolation mode (see the RTTOV user guide).
do_checkinput	logical	Whether to check the input for reasonableness (see the RTTOV user guide).
apply_reg_limits	logical	Whether to clamp the atmospheric values to the RTTOV bounds (see the RTTOV use
verbose	logical	Whether to output lots of additional output (see the RTTOV user guide).
fix_hgpl	logical	Whether the surface pressure represents the surface or the 2 meter value (see the RTT
do_lambertian	logical	Whether to include the effects of surface specularity (see the RTTOV user guide).
lambertian_fixed_angle	logical	Whether to include a fixed angle for the lambertian effect (see the RTTOV user guide
rad_down_lin_tau	logical	Whether to use the linear-in-tau approximation (see the RTTOV user guide).
use_q2m	logical	Whether to use 2m humidity information (see the RTTOV user guide). If true, the QT
use_q2m	logical	Whether to use 2m humidity information (see the RTTOV user guide). If true, the QT
use_uv10m	logical	Whether to use 10m wind speed information (see the RTTOV user guide). If true, the
use_wfetch	logical	Whether to use wind fetch information (see the RTTOV user guide). If true, the QTY
use_water_type	logical	Whether to use water-type information (0 = fresh, 1 = ocean; see the RTTOV user gu
addrefrac	logical	Whether to enable atmospheric refraction (see the RTTOV user guide).
plane_parallel	logical	Whether to treat the atmosphere as plane parallel (see the RTTOV user guide).
use_salinity	logical	Whether to use salinity (see the RTTOV user guide). If true, the QTY_SALINITY wi
apply_band_correction	logical	Whether to apply band correction from the coefficient field for microwave data (see t
cfrac_data	logical	Whether to use the cloud fraction from 0 to 1 (see the RTTOV user guide). If true, the
clw_data	logical	Whether to use cloud-liquid water data (see the RTTOV user guide). If true, the QTY
rain_data	logical	Whether to use precipitating water data (see the RTTOV user guide). If true, the QTY
ciw_data	logical	Whether to use non-precipiting ice information (see the RTTOV user guide). If true, t
snow_data	logical	Whether to use precipitating fluffy ice (see the RTTOV user guide). If true, the QTY_
graupel_data	logical	Whether to use precipting small, hard ice (see the RTTOV user guide). If true, the QT
hail_data	logical	Whether to use precipitating large, hard ice (see the RTTOV user guide). If true, the Q
w_data	logical	Whether to use vertical velocity information. This will be used to crudely classify if a
clw_scheme	integer	The clw_scheme to use (see the RTTOV user guide).
clw_cloud_top	real(r8)	Lower hPa limit for clw calculations (see the RTTOV user guide).
fastem_version	integer	Which FASTEM version to use (see the RTTOV user guide).
supply_foam_fraction	logical	Whether to use sea-surface foam fraction (see the RTTOV user guide). If true, the QT
use_totalice	logical	Whether to use totalice instead of precip/non-precip ice for microwave (see the RTTO
use_zeeman	logical	Whether to use the Zeeman effect (see the RTTOV user guide). If true, the magnetic f

Item	Type	Description
cc_threshold	real(r8)	Cloud-fraction value to treat as clear-sky (see the RTTOV user guide).
ozone_data	logical	Whether to use ozone (O3) profiles (see the RTTOV user guide). If true, the QTY_O
co2_data	logical	Whether to use carbon dioxide (CO2) profiles (see the RTTOV user guide). If true, th
n2o_data	logical	Whether to use nitrous oxide (N2O) profiles (see the RTTOV user guide). If true, the
co_data	logical	Whether to use carbon monoxide (CO) profiles (see the RTTOV user guide). If true, t
ch4_data	logical	Whether to use methane (CH4) profiles (see the RTTOV user guide). If true, the QTY
so2_data	logical	Whether to use sulfur dioxide (SO2) (see the RTTOV user guide). If true, the QTY_S
addsolar	logical	Whether to use solar angles (see the RTTOV user guide). If true, the sun_ze and sun_
rayleigh_single_scatt	logical	Whether to use only single scattering for Rayleigh scattering for visible calculations (
do_nlte_correction	logical	Whether to include non-LTE bias correction for HI-RES sounder (see the RTTOV us
solar_sea_brdf_model	integer	The solar sea BRDF model to use (see the RTTOV user guide).
ir_sea_emis_model	logical	The infrared sea emissivity model to use (see the RTTOV user guide).
use_sfc_snow_frac	logical	Whether to use the surface snow fraction (see the RTTOV user guide). If true, the QT
add_aerosl	logical	Whether to use aerosols (see the RTTOV user guide).
aerosl_type	integer	Whether to use OPAC or CAMS aerosols (see the RTTOV user guide).
add_clouds	logical	Whether to enable cloud scattering for visible/infrared (see the RTTOV user guide).
ice_scheme	integer	The ice scheme to use (see the RTTOV user guide).
use_icede	logical	Whether to use the ice effective diameter for visible/infrared (see the RTTOV user gu
idg_scheme	integer	The ice water effective diameter scheme to use (see the RTTOV user guide).
user_aer_opt_param	logical	Whether to directly specify aerosol scattering properties (see the RTTOV user guide).
user_cld_opt_param	logical	Whether to directly specify cloud scattering properties (see the RTTOV user guide).
grid_box_avg_cloud	logical	Whether to cloud concentrations are grid box averages (see the RTTOV user guide).
cldstr_threshold	real(r8)	Threshold for cloud stream weights for scattering (see the RTTOV user guide).
cldstr_simple	logical	Whether to use one clear and one cloudy column (see the RTTOV user guide).
cldstr_low_cloud_top	real(r8)	Cloud fraction maximum in layers from the top of the atmosphere down to the specifi
ir_scatt_model	integer	Which infrared scattering method to use (see the RTTOV user guide).
vis_scatt_model	integer	Which visible scattering method to use (see the RTTOV user guide).
dom_nstreams	integer	The number of streams to use with DOM (see the RTTOV user guide).
dom_accuracy	real(r8)	The convergence criteria for DOM (see the RTTOV user guide).
dom_opdep_threshold	real(r8)	Ignore layers below this optical depth (see the RTTOV user guide).
addpc	logical	Whether to do principal component calculations (see the RTTOV user guide).
npcscores	integer	Number of principal components to use for addpc (see the RTTOV user guide).
addradrec	logical	Reconstruct the radiances using addpc (see the RTTOV user guide).
ipcreg	integer	Number of predictors to use with addpc (see the RTTOV user guide).
use_htfrtc	logical	Whether to use HTFRTC (see the RTTOV user guide).
htfrtc_n_pc	integer	Number of PCs to use with HTFRTC (see the RTTOV user guide).
htfrtc_simple_cloud	logical	Whether to use simple cloud scattering with htfrtc (see the RTTOV user guide).
htfrtc_overcast	logical	Whether to calculate overcast radiances with HTFRTC (see the RTTOV user guide).

6.151.5 Files

- A DART observation sequence file containing Radar obs.

6.151.6 References

- RTTOV user guide

6.151.7 Private components

use obs_def_rttov_mod, only :	initialize_module
	initialize_rttov_sensor_runtime
	initialize_rttov_sensor_runtime

call initialize_module()

Reads the namelist, allocates space for the auxiliary data associated wtih satellite observations, initializes the constants used in subsequent computations (possibly altered by values in the namelist), and prints out the list of constants and the values in use.

call initialize_rttov_sensor_runtime(sensor,ens_size,nlevels)

```
type(rttov_sensor_type), pointer    :: sensor
integer,                 intent(in) :: ens_size
integer,                 intent(in) :: nlevels
```

Initialize a RTTOV sensor runtime. A rttov_sensor_type instance contains information such as options and coefficients that are initialized in a “lazy” fashion only when it will be used for the first time.

`sensor`	The sensor type to be initialized
`ens_size`	The size of the ensemble
`nlevels`	The number of vertical levels in the atmosphere

6.152 SSEC Data Center

6.152.1 Overview

The program in this directory takes satellite wind data from the University of Wisconsin-Madison Space Science and Engineering Center, and converts it into DART format observation sequence files, for use in assimilating with the DART filter program.

6.152.2 Data sources

The Space Science and Engineering Center (SSEC) at University of Wisconsin-Madison has an online data center with both real-time and archival weather satellite data.

The last 2 day's worth of data is available from ftp://cyclone.ssec.wisc.edu/pub/fnoc.

There is a second satellite wind DART converter in the *MADIS Data Ingest System* directory which converts wind observations which originate from NESDIS. The data from this converter is processed at the SSEC and the observations will be different from the ones distributed by MADIS.

6.152.3 Programs

Conversion program `convert_ssec_satwnd` converts the ascii data in the input files into a DART observation sequence file. Go into the `work` directory and run the `quickbuild.csh` script to compile the necessary files.

The program reads standard input for the data time range, which types of observations to convert, and then, if quality control information is found in the input file, what type of quality control algorithm to use when deciding whether the observation is of good quality or not. See the references below.

6.152.4 References

- RF method: Velden, C. S., T. L. Olander, and S. Wanzong, 1998: The impact of multispectral GOES-8 wind information on Atlantic tropical cyclone track forecasts in 1995. Part I: Dataset methodology, description, and case analysis. Mon. Wea. Rev., 126, 1202-1218.
- QI method: Holmlund, K., 1998: The utilization of statistical properties of satellite-derived atmospheric motion vectors to derive quality indicators. Wea. Forecasting, 13, 1093-1104.
- Comparison of two methods: Holmlund, K., C.S. Velden, and M. Rohn, 2001: Enhanced Automated Quality Control Applied to High-Density Satellite-Derived Winds. Mon. Wea. Rev., 129, 517-529.

6.153 GTSPP Observations

6.153.1 Overview

GTSPP (Global Temperature-Salinity Profile Program) data measures vertical profiles of ocean temperature and salinity. The GTPSS home page has detailed information about the repository, observations, and datasets. The programs in this directory convert from the netcdf files found in the repository into DART observation sequence (obs_seq) file format.

6.153.2 Data sources

Data from the GTSPP can be downloaded interactively from here. It is delivered in netCDF file format, one vertical profile per netCDF file.

Currently each vertical profile is stored in a separate file, so converting a months's worth of observations involves downloading many individual files. The converter program can take a list of input files, so it is easy to collect a month of observations together into a single output file with one execution of the converter program.

The units in the source file are degrees C for temperature, g/kg for salinity, and so far we have not found any error information (not quality control, but observation instrument error values). There is probably instrument source information encoded in these files, but so far we don't have the key. The quality control values are read and only those with a QC of 1 are retained.

6.153.3 Programs

The data is distributed in netCDF file format. DART requires all observations to be in a proprietary format often called DART "obs_seq" format. The files in this directory, a combination of C shell scripts and a Fortran source executable, do this data conversion.

6.153.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&gtspp_to_obs_nml
   gtspp_netcdf_file      = '1234567.nc'
   gtspp_netcdf_filelist = 'gtspp_to_obs_filelist'
   gtspp_out_file         = 'obs_seq.gtspp'
   avg_obs_per_file       = 500
   debug                  = .false.
/
```

Item	Type	Description
gtspp_netcdf_file	character(len=128)	The input filename when converting a single profile. Only one of the two file or filelist items can have a valid value, so to use the single filename set the list name 'gtspp_netcdf_filelist' to the empty string (' ').
gtspp_netcdf_filelist	character(len=128)	To convert a series of profiles in a single execution create a text file which contains each input file, in ascii, one filename per line. Set this item to the name of that file, and set 'gtspp_netcdf_file' to the empty string (' ').
gtspp_out_file	character(len=128)	The output file to be created. To be compatible with earlier versions of this program, if this file already exists it will be read in and the new data will be inserted into that file.
avg_obs_per_file	integer	The code needs an upper limit on the number of observations generated by this program. It can be larger than the actual number of observations converted. The total number of obs is computed by multiplying this number by the number of input files. If you get an error because there is no more room to add observations to the output file, increase this number.
debug	logical	If true, output more debugging messages.

6.153.5 Modules used

```
types_mod
time_manager_mod
utilities_mod
location_mod
obs_sequence_mod
obs_def_mod
obs_def_ocean_mod
obs_kind_mod
netcdf
```

6.154 GPS Observations

6.154.1 Overview

The COSMIC project provides data from a series of satellites. There are two forms of the data that are used by DART: GPS Radio Occultation data and Electron Density. The programs in this directory extract the data from the distribution files and put them into DART observation sequence (obs_seq) file format.

Radio occultation

The COSMIC satellites measure the phase delay caused by deviation of the straight-line path of the GPS satellite signal as it passes through the Earth's atmosphere when the GPS and COSMIC satellites rise and set relative to each other. This deviation results from changes in the angle of refraction of light as it passes through regions of varying density of atmosphere. These changes are a result of variations in the temperature, pressure, and moisture content. Vertical profiles of temperature and moisture can be derived as the signal passes through more and more atmosphere until it is obscured by the earth's horizon. There are thousands of observations each day distributed around the globe, including in areas which previously were poorly observed. These data are converted with the `convert_cosmic_gps_cdf.f90` program and create DART observations of GPSRO_REFRACTIVITY.

Electron density

The COSMIC satellites also provide ionospheric profiles of electron density. The accuracy is generally about 10^{-4} 10^{-5} cm^{-3}. These data are converted with the `convert_cosmic_ionosphere.f90` program and create DART observations tagged as COSMIC_ELECTRON_DENSITY.

6.154.2 Data sources

Data from the COSMIC Program are available by signing up on the data access web page. We prefer delivery in netCDF file format.

Radio occultation

The files we use as input to these conversion programs are the Level 2 data, Atmospheric Profiles (filenames include the string 'atmPrf').

Each vertical profile is stored in a separate netCDF file, and there are between 1000-3000 profiles/day, so converting a day's worth of observations used to involve downloading many individual files. There are now daily tar files available which makes it simpler to download the raw data all in a single file and then untar it to get the individual profiles.

The scripts in the `shell_scripts` directory can now download profiles from any of the available satellites that return GPS RO data to the CDAAC web site. See the `gpsro_to_obsseq.csh` or `convert_many_gpsro.csh` script for where to specify the satellites to be included.

Electron density

The files we have used as input to these conversion programs are from the COSMIC 2013 Mission and have a data type of 'ionPrf'.

The file naming convention and file format are described by COSMIC here and there can be more than 1000 profiles/day. Like the GPS radio occultation data, the profiles are now available in a single daily tar file which can be downloaded then be unpacked into the individual files. COSMIC has instructions on ways to download the data at http://cdaac-www.cosmic.ucar.edu/cdaac/tar/rest.html

6.154.3 Programs

Convert_cosmic_gps_cdf

The data are distributed in netCDF file format. DART requires all observations to be in a proprietary format often called DART "obs_seq" format. The files in this directory (a combination of C shell scripts and a Fortran source executable) do this data conversion.

The shell_scripts directory contains several example scripts, including one which downloads the raw data files a day at a time (`download_script.csh`), and one which executes the conversion program (`convert_script.csh`). These scripts make 6 hour files by default, but have options for other times. Each profile is stored in a separate netcdf file and there are usually between 1000-3000 files/day, so the download process can be lengthy. You probably want to download as a separate preprocess step and do not use the script options to automatically delete the input files. Keep the files around until you are sure you are satisified with the output files and then delete them by hand.

The conversion executable `convert_cosmic_gps_cdf`, reads the namelist `&convert_cosmic_gps_nml` from the file `input.nml`.

The namelist lets you select from one of two different forward operators. The 'local' forward operator computes the expected observation value at a single point: the requested height at the tangent point of the ray between satellites. The 'non-local' operator computes values along the ray-path and does an integration to get the expected value. The length of the integration segments and height at which to end the integration are given in the namelist. In some experiments the difference between the two types of operators was negligible. This choice is made at the time of the conversion, and the type of operator is stored in the observation, so at runtime the corresponding forward operator will be used to compute the expected observation value.

The namelist also lets you specify at what heights you want observations to be extracted. The raw data is very dense in the vertical; using all values would not results in a set of independent observations. The current source code no longer does an intermediate interpolation; the original profiles appear to be smooth enough that this is not needed. The requested vertical output heights are interpolated directly from the full profile.

Convert_cosmic_ionosphere

Each profile is interpolated to a set of desired levels that are specified at run time. During the conversion process, each profile is checked for negative values of electron density above the minimum desired level. If negative values are found, the entire profile is discarded. If an observation sequence file already exists, the converter will simply add the new observations to it. Multiple profiles may be converted in a single execution, so it is easy to consolidate all the profiles for a single day into a single observation sequence file, for example. `convert_cosmic_ionosphere` reads the namelist `&convert_cosmic_ionosphere_nml` from the file `input.nml`. The original observation times are preserved in the conversion process. If it is desired to subset the observation sequence file such that observations too far away from desired assimilation times are rejected, a separate post-processing step using the *program obs_sequence_tool* is required. A script will be necessary to take a start date, an end date, an assimilation time step, and a desired time 'window' - and strip out the unwanted observations from a series of observation sequence files. There are multiple ways of specifying the observation error variance at run time. They are implemented in a routine named `electron_density_error()` and are selected by the namelist variable `observation_error_method`.

'constant'	a scalar value for all observations
'scaled'	the electron density is multiplied by a scalar value
'lookup'	a lookup table is read
'scaled_lookup'	the lookup table value is multiplied by a scalar value and the electron density value

> I-Te Lee: " … the original idea for error of ionospheric observation is 1%. Thus, I put the code as "oerr = 0.01_r8 * obsval". Liu et. al and Yue et al investigated the Abel inversion error of COSMIC ionosphere profile, both of them figure out the large error would appear at the lower altitude and push model toward wrong direction at the lower ionosphere while assimilating these profiles. On the other hand, the Abel

> inversion error depends on the ionospheric electron density structure, which is a function of local time, altitude and geomagnetic latitude. To simplify the procedure to define observation error of profiles, Xinan Yue help me to estimate an error matrix and saved in the file which named 'f3coerr.nc'. . . . The number in the matrix is error percentage (%), which calculated by OSSE. Here are two reference papers. In the end, the observation error consists of instrumentation error (10%) and Abel error."
>
> - X. Yue, W.S. Schreiner, J. Lei, S.V. Sokolovskiy, C. Rocken, D.C. Hunt, and Y.-H. Kuo (2010), Error analysis of Abel retrieved electron density profiles from radio occultation measurements. *Annales Geophysicae: Atmospheres, Hydrospheres and Space Sciences*. **28** No. 1, pp 217-222, doi:10.5194/angeo-28-217-2010
> - J.Y. Liu, C.Y. Lin, C.H. Lin, H.F. Tsai, S.C. Solomon, Y.Y. Sun, I.T. Lee, W.S. Schreiner, and Y.H. Kuo (2010), Artificial plasma cave in the low-latitude ionosphere results from the radio occultation inversion of the FORMOSAT-3/COSMIC}, *Journal of Geophysical Research: Space Physics*. **115** No. A7, pp 2156-2202, doi:10.1029/2009JA015079

It is possible to create observation sequence files for perfect model experiments that have realistic observation sampling patterns and observation error variances that **do not have any actual electron densities**. The COSMIC data files are read, but the electron density information is not written. Keep in mind that some methods of specifying the observation error variance require knowledge of the observation value. If the observation value is bad or the entire profile is bad, no observation locations are created for the profile.

6.154.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&convert_cosmic_gps_nml
   obs_levels              = -1.0
   use_original_kuo_error = .false.
   local_operator          = .true.
   ray_ds                  = 5000.0
   ray_htop                = 15000.0
   gpsro_netcdf_file       = 'cosmic_gps_input.nc'
   gpsro_netcdf_filelist   = ''
   gpsro_out_file          = 'obs_seq.gpsro'
 /
```

Item	Type	Description
obs_levels	integer(200)	A series of heights, in kilometers, where observations from this profile should be interpolated. (Note that the other distances and heights in the namelist are specified in meters.) The values should be listed in increasing height order.
use_original_kuo_error	logical	If .true. use the observation error variances for a refractivity observation that come from a Kuo paper and were implied to be used for the CONUS domain. If .false. use observation error variances similar to what is used in GSI.
local_operator	logical	If .true. compute the observation using a method which assumes all effects occur at the tangent point. If .false. integrate along the tangent line and do ray-path reconstruction.
ray_ds	real(r8)	For the non-local operator only, the delta stepsize, in meters, to use for the along-path integration in each direction out from the tangent point.
ray_htop	real(r8)	For the non-local operator only, stop the integration when one of the endpoints of the next integration step goes above this height. Specify in meters.
gpsro_netcdf_file	character(len=128)	The input filename when converting a single profile. Only one of the file or filelist items can have a valid value, so to use the single filename set the list name ‘gpsro_netcdf_filelist’ to the empty string (‘ ‘).
gpsro_netcdf_filelist	character(len=128)	To convert a series of profiles in a single execution create a text file which contains each input file, in ascii, one filename per line. Set this item to the name of that file, and set ‘gpsro_netcdf_file’ to the empty string (‘ ‘).
gpsro_out_file	character(len=128)	The output file to be created. To be compatible with earlier versions of this program, if this file already exists it will be read in and the new data will be appended to that file.

A more useful example follows:

```
&convert_cosmic_gps_nml
  gpsro_netcdf_file      = ''
  gpsro_netcdf_filelist  = 'flist'
  gpsro_out_file         = 'obs_seq.gpsro'
  local_operator         = .true.
  use_original_kuo_error = .false.
  ray_ds                 = 5000.0
  ray_htop               = 13000.1
  obs_levels =        0.2,  0.4,  0.6,  0.8,
                1.0,  1.2,  1.4,  1.6,  1.8,
                2.0,  2.2,  2.4,  2.6,  2.8,
                3.0,  3.2,  3.4,  3.6,  3.8,
                4.0,  4.2,  4.4,  4.6,  4.8,
                5.0,  5.2,  5.4,  5.6,  5.8,
                6.0,  6.2,  6.4,  6.6,  6.8,
                7.0,  7.2,  7.4,  7.6,  7.8,
                8.0,  8.2,  8.4,  8.6,  8.8,
                9.0,  9.2,  9.4,  9.6,  9.8,
               10.0, 10.2, 10.4, 10.6, 10.8,
               11.0, 11.2, 11.4, 11.6, 11.8,
               12.0, 12.2, 12.4, 12.6, 12.8,
               13.0, 13.2, 13.4, 13.6, 13.8,
               14.0, 14.2, 14.4, 14.6, 14.8,
               15.0, 15.2, 15.4, 15.6, 15.8,
               16.0, 16.2, 16.4, 16.6, 16.8,
               17.0, 17.2, 17.4, 17.6, 17.8,
               18.0, 19.0, 20.0, 21.0, 22.0,
               23.0, 24.0, 25.0, 26.0, 27.0,
               28.0, 29.0, 30.0, 31.0, 32.0,
               33.0, 34.0, 35.0, 36.0, 37.0,
               38.0, 39.0, 40.0, 41.0, 42.0,
```

(continues on next page)

(continued from previous page)

```
               43.0, 44.0, 45.0, 46.0, 47.0,
               48.0, 49.0, 50.0, 51.0, 52.0,
               53.0, 54.0, 55.0, 56.0, 57.0,
               58.0, 59.0, 60.0,
 /
```

```
&convert_cosmic_ionosphere_nml
  input_file                 = ''
  input_file_list            = 'input_file_list.txt'
  output_file                = 'obs_seq.out'
  observation_error_file     = 'none'
  observation_error_method = 'scaled_lookup'
  locations_only             = .false.
  obs_error_factor           = 1.0
  verbose                    = 0
  obs_levels                 = -1.0
 /
```

Item	Type	Description
input_file	character(len=256)	The input filename when converting a single profile. Only one of the `input_file` or `input_file_list` items can have a valid value, so to use a single filename set `input_file_list = ''`
input_file_list	character(len=256)	To convert a series of profiles in a single execution create a text file which contains one filename per line. Set this item to the name of that file, and set `input_file = ''`
output_file	character(len=256)	The output file to be created. If this file already exists the new data will be added to that file. DART observation sequences are linked lists. When the list is traversed, the observations are in ascending time order. The order they appear in the file is completely irrelevant.
observation_error_file	character(len=256)	This specifies a lookup table. The table created by I-Te Lee and Xinan Yue is called `f3coerr.nc`.
observation_error_method	character(len=128)	There are multiple ways of specifying the observation error variance. This character string allows you to select the method. The selection is not case-sensitive. Allowable values are: 'constant', 'scaled', 'lookup', or 'scaled_lookup'. Anything else will result in an error. Look in the `electron_density_error()` routine for specifics.
locations_only	logical	If `locations_only = .true.` then the actual observation values are not written to the output observation sequence file. This is useful for designing an OSSE that has a realistic observation sampling pattern. Keep in mind that some methods of specifying the observation error variance require knowledge of the observation value. If the observation value is bad or the entire profile is bad, this profile is rejected - even if `locations_only = .true.`
obs_error_factor	real(r8)	This is the scalar that is used in several of the methods specifying the observation error variance.
verbose	integer	controls the amount of run-time output echoed to the screen. 0 is nearly silent, higher values write out more. The filenames of the profiles that are skipped are ALWAYS printed.
obs_levels	integer(200)	A series of heights, in kilometers, where observations from this profile should be interpolated. (Note that the other distances and heights in the namelist are specified in meters.) The values must be listed in increasing height order.

A more useful example follows:

```
&convert_cosmic_ionosphere_nml
   input_file                 = ''
   input_file_list            = 'file_list.txt'
   output_file                = 'obs_seq.out'
   observation_error_file     = 'f3coeff.dat'
   observation_error_method = 'scaled'
   locations_only             = .false.
   obs_error_factor           = 0.01
   verbose                    = 1
   obs_levels = 160.0, 170.0, 180.0, 190.0, 200.0,
                210.0, 220.0, 230.0, 240.0, 250.0,
                260.0, 270.0, 280.0, 290.0, 300.0,
                310.0, 320.0, 330.0, 340.0, 350.0,
                360.0, 370.0, 380.0, 390.0, 400.0,
                410.0, 420.0, 430.0, 440.0, 450.0
  /
```

6.154.5 Workflow for batch conversions

If you are converting only a day or two of observations you can download the files by hand and call the converter programs from the command line. However if you are going convert many days/months/years of data you need an automated script, possibly submitted to a batch queue on a large machine. The following instructions describe shell scripts we provide as a guide in the `shell_scripts` directory. You will have to adapt them for your own system unless you are running on an NCAR superscomputer.

Making DART Observations from Radio Occultation atmPrf Profiles:

```
Description of the scripts provided to process the COSMIC and
CHAMP GPS radio occultation data.

Summary of workflow:
1) cd to the ../work directory and run ./quickbuild.csh to compile everything.
2) Edit ./gpsro_to_obsseq.csh once to set the directory where the DART
    code is installed, and your CDAAC web site user name and password.
3) Edit ./convert_many_gpsro.csh to set the days of data to download/convert/remove.
4) Run ./convert_many_gpsro.csh either on the command line or submit to a batch␣
 ↪system.


More details:

1) quickbuild.csh:

Make sure your $DART/mkmf/mkmf.template is one that matches the
platform and compiler for your system.  It should be the same as
how you have it set to build the other DART executables.

Run quickbuild.csh and it should compile all the executables needed
to do the GPS conversion into DART obs_sequence files.
```

(continues on next page)

(continued from previous page)

```
2) gpsro_to_obsseq.csh:

Edit gpsro_to_obsseq.csh once to set the DART_DIR to where you have
downloaded the DART distribution.  (There are a few additional options
in this script, but the distribution version should be good for most users.)
If you are downloading data from the CDAAC web site, set your
web site user name and password.  After this you should be able to
ignore this script.


3) convert_many_gpsro.csh:

A wrapper script that calls the converter script a day at a time.
Set the days of data you want to download/convert/remove.  See the
comments at the top of this script for the various options to set.
Rerun this script for all data you need.  This script depends on
the advance_time executable, which should automatically be built
in the ../work directory, but you may have to copy or link to a
version from this dir.  you also need a minimal input.nml here:

&utilities_nml
 /

is all the contents it needs.


It can be risky to use the automatic delete/cleanup option - if there are
any errors in the script or conversion (file system full, bad file format,
etc) and the script doesn't exit, it can delete the input files before
the conversion has succeeded.  But if you have file quota concerns
this allows you to keep the total disk usage lower.
```

Making DART Observations from Ionospheric ionPrf Profiles:

```
0) run quickbuild.csh as described above

1) iono_to_obsseq.csh

set the start and stop days.  downloads from the CDAAC and
untars into 100s of files per day.  runs the converter to
create a single obs_seq.ion.YYYYMMDD file per day.

2) split_obs_seq.csh

split the daily files into X minute/hour files - set the
window times at the top of the file before running.
```

Notes on already converted observations on the NCAR supercomputers

GPS Radio Occultation Data:

```
See /glade/p/image/Observations/GPS

These are DART observation sequence files that contain
radio-occultation measurements from the COSMIC
(and other) satellites.

Uses temperature/moisture bending of the signals as they
pass through the atmosphere between GPS source satellites
and low-earth-orbit receiving satellites to compute the
delay in the arrival of data. the files also contain the
bending angle data, but we are not using that currently.


the subdirectories include:

local -- original processed files, single obs at nadir
local-cosmic2013 -- reprocessed by CDAAC in 2013
local-test2013 -- 2013 data, denser in vertical, diff errors
local-complete2013 - all satellites available for that time,
 new errors (from lydia c), 2013 cosmic reprocessed data
nonlocal -- original processed files, ray-path integrated
rawdata -- netcdf data files downloaded from the CDAAC

local: the ob is at a single location (the tangent point
of the ray and earth) and the entire effect is assumed
to be impacting the state at that point.

non-local: computes the ob value by doing a line integral
along the ray path to accumulate the total effect.

(in our experiments we have compared both and did not see
a large difference between the two methods, and so have
mistly used the local version because it's faster to run.)


some directories contain only the gps obs and must be
merged (with the obs_sequence_tool) with the rest of
the conventional obs before assimilation.

some directories contain both the gps-only files and
the obs merged with NCEP and ACARS data.


if a directory exists but is empty, the files are
likely archived on the HPSS.  see the README files
in the next level directory down for more info on
where they might be.

nsc
jan 2016
```

Ionosphere Data:

```
See /glade/p/image/Observation/ionosphere

These are COSMIC 'ionPrf' ionospheric profile observations.

They are downloaded from the CDAAC website as daily tar files
and unpacked into the 'raw' directory.  They distribute these
observations with one profile per netcdf file.  Each profile has
data at ~500-1000 different levels.

Our converter has a fixed number of levels in the namelist
and we interpolate between the two closest levels to get the
data for that level.  If you give the converter a list of
input netcdf files it will convert all of them into a
single output file.

The 'daily' directory is a collection of all the profiles for
that day.

The 'convert' directory has the executables and scripting
for breaking up the daily files into 10 minute files which
are put in the '10min' directory.  Change the 'split_obs_seq.csh'
script to change the width of this window, or the names of
the output files.

The 'verify.csh' script prints out any missing files, which
happens if there are no profiles in the given window.

Our convention is to make a 0 length file for missing intervals
and we expect the filter run script to look at the file size
and loop if there is a file but with no contents.  This will
allow us to distinguish between a time where we haven't converted
the observations and a time where there are no observations.
In that case the script should add time to the next model
advance request and loop to the next interval.
```

6.154.6 Modules used

`convert_cosmic_gps_cdf` and `convert_cosmic_ionosphere` use the same set of modules.

```
assimilation_code/location/threed_sphere/location_mod.f90
assimilation_code/modules/assimilation/adaptive_inflate_mod.f90
assimilation_code/modules/assimilation/assim_model_mod.f90
assimilation_code/modules/io/dart_time_io_mod.f90
assimilation_code/modules/io/direct_netcdf_mod.f90
assimilation_code/modules/io/io_filenames_mod.f90
assimilation_code/modules/io/state_structure_mod.f90
assimilation_code/modules/io/state_vector_io_mod.f90
assimilation_code/modules/observations/obs_kind_mod.f90
assimilation_code/modules/observations/obs_sequence_mod.f90
```

(continues on next page)

(continued from previous page)

```
assimilation_code/modules/utilities/distributed_state_mod.f90
assimilation_code/modules/utilities/ensemble_manager_mod.f90
assimilation_code/modules/utilities/netcdf_utilities_mod.f90
assimilation_code/modules/utilities/null_mpi_utilities_mod.f90
assimilation_code/modules/utilities/null_win_mod.f90
assimilation_code/modules/utilities/options_mod.f90
assimilation_code/modules/utilities/random_seq_mod.f90
assimilation_code/modules/utilities/sort_mod.f90
assimilation_code/modules/utilities/time_manager_mod.f90
assimilation_code/modules/utilities/types_mod.f90
assimilation_code/modules/utilities/utilities_mod.f90
models/template/model_mod.f90
models/utilities/default_model_mod.f90
observations/forward_operators/obs_def_mod.f90
observations/forward_operators/obs_def_utilities_mod.f90
observations/obs_converters/utilities/obs_utilities_mod.f90
```

6.154.7 Errors

The converters have a parameter declaring the maximum number of desired levels as 200. If more than 200 levels are entered as input (to `obs_levels`), a rather uninformative run-time error is generated:

```
ERROR FROM:
 routine: check_namelist_read
 message:  INVALID NAMELIST ENTRY:  / in namelist convert_cosmic_ionosphere_nml
```

Your error may be different if `obs_levels` is not the last namelist item before the slash '/'

6.155 GSI2DART

6.155.1 Overview

The GSI2DART converter was contributed by **Craig Schwartz** and **Jamie Bresch** of the Mesoscale & Microscale Meteorology Lab at NCAR. *Thanks Craig and Jamie!*

This converter is designed to convert observation files created by the Gridpoint Statistical Interpolation (GSI) system maintained by the National Oceanic and Atmospheric Administration (NOAA) into DART observation sequence files. The files created by GSI are 'BIG_ENDIAN' and have filenames such as:

- diag_amsua_metop-a_ges.ensmean
- diag_amsua_metop-a_ges.mem001
- diag_amsua_metop-a_ges.mem002
- diag_amsua_n18_ges.ensmean
- diag_amsua_n18_ges.mem001
- diag_amsua_n18_ges.mem002
- diag_amsua_n19_ges.ensmean
- diag_amsua_n19_ges.mem001
- diag_amsua_n19_ges.mem002

- diag_conv_ges.ensmean
- diag_conv_ges.mem001
- diag_conv_ges.mem002

The DART converter uses routines from the GSI system that use the Message Passing Interface (MPI) to process observations in parallel (even when converting a small amount of observations) so MPI is required to execute this observation converter.

Due to these prerequisites, we provide a detailed description of this directory to guide the user.

This directory contains copies of several source code files from GSI. The GSI source code is available via a Github repository managed by NOAA's Environmental Modeling Center (EMC):

https://github.com/NOAA-EMC/GSI

To differentiate between the sets of code, we refer to the root directory of the NOAA-EMC repository as `GSI` and refer to the root directory of this observation converter as `GSI2DART`.

`GSI2DART/enkf` copies seven files from `GSI/src` mostly without modification:

1. `GSI2DART/enkf/constants.f90` from `GSI/src/gsi/constants.f90`
2. `GSI2DART/enkf/kinds.F90` from `GSI/src/gsi/kinds.F90`
3. `GSI2DART/enkf/mpi_readobs.f90` from `GSI/src/enkf/mpi_readobs.f90`
4. `GSI2DART/enkf/readconvobs.f90` from `GSI/src/enkf/readconvobs.f90`
5. `GSI2DART/enkf/read_diag.f90` from `GSI/src/gsi/read_diag.f90`
6. `GSI2DART/enkf/readozobs.f90` from `GSI/enkf/readozobs.f90`
7. `GSI2DART/enkf/readsatobs.f90` from `GSI/enkf/readsatobs.f90`

Note that within `GSI` the source file `kinds.F90` has an upper-case `F90` suffix. Within the `GSI2DART` observation converter, it gets preprocessed into `mykinds.f90` with a lower-case `f90` suffix. Case-insensitive filesystems should be banned ... until then, it is more robust to implement some name change during preprocessing. The path name specified in `GSI2DART/work/path_names_gsi_to_dart` reflects this processed filename.

The following three files had their open() statements modified to read 'BIG_ENDIAN' files without the need to compile EVERYTHING with the `-convert big_endian` compiler option. Using the DART open_file() routine also provides some nice error handling.

- original: `open(iunit,form="unformatted",file=obsfile,iostat=ios)`
- modified: `iunit = open_file(obsfile,form='unformatted',action='read', convert='BIG_ENDIAN')`

1. `GSI2DART/enkf/readconvobs.f90`
2. `GSI2DART/enkf/readozobs.f90`
3. `GSI2DART/enkf/readsatobs.f90`

6.155.2 DART Modifications

Within GSI2DART

The source files within GSI2DART are:

1. gsi_to_dart.f90: the main program.
2. dart_obs_seq_mod.f90: the DART obs_seq output subroutine.
3. params.f90: the same module name as GSI/src/enkf/params.f90 but with different content. This version is used to avoid modifying GSI2DART/enkf/read*.f90.
4. radinfo.f90: the same module name as GSI/src/gsi/radinfo.f90 but with different content. This version is used to avoid modifying GSI2DART/enkf/read*.f90.
5. mpisetup.f90: the same module name as GSI/src/enkf/mpisetup.f90 but with different content. This version is used to avoid dependency on GSI.

Elsewhere in the repository

This observation converter required modifying two files and adding a module for radiance observation types.

- Modified ../../forward_operators/DEFAULT_obs_def_mod.F90
- Modified ../../DEFAULT_obs_kind_mod.F90
- Added ../../forward_operators/obs_def_radiance_mod.f90 which has radiance observation types

Compiler notes

When using ifort, the Intel Fortran compiler, you may need to add the compiler flag -nostdinc to avoid inserting the standard C include files which have incompatible comment characters for Fortran. You can add this compiler flag in the the GSI2DART/work/mkmf_gsi_to_dart file by adding it to the “-c” string contents.

Please note: this was NOT needed for ifort version 19.0.5.281.

Additional files and directories

1. satinfo is a file read by radinfo.f90 and must exist in the GSI2DART/work directory.
2. datapath specifies the directory containing the data to be converted – it is specified in the gsi_to_dart_nml namelist in GSI2DART/work/input.nml.
3. submit.csh is contained in GSI2DART/work/ – it runs the gsi_to_dart converter once it has been compiled. Again, since GSI requires MPI, multiple processors must be requested to run the gsi_to_dart executable.

6.155.3 Issues

1. The converter requires an ensemble size greater than one and will MPI_Abort() if only one ensemble member is requested.

The following are issues previously recorded in the README:

1. Radiance and surface pressure bias correction
2. Surface pressure altimeter adjustment?
3. Specific humidity obs are transformed to relative humidity. What to do? [Just run EnSRF with psuedo_rh=.false. and assimilate RH obs]
4. DART must use W and PH as control variables [okay, EnSRF can do this too (nvars=6 for WRF-ARW)]
5. Does DART not do vertical localization for surface obs?

```
! If which_vert has no vertical definition for either location do only horizontal
if(loc1%which_vert == VERTISUNDEF .or. loc2%which_vert == VERTISUNDEF) comp_h_only = .
↪true.
! If both verts are surface, do only horizontal
if(loc1%which_vert == VERTISSURFACE .and. loc2%which_vert == VERTISSURFACE) comp_h_
↪only = .true.
```

Running with 32 bit reals

The converter has been tested with 64-bit reals as well as 32-bit reals (i.e. r8=r4 and -D_REAL_4). The answers are different only at the roundoff level.

This requires changes in two places:

1. `DART/assimilation_code/modules/utilities/types_mod.f90` change required: r8 = r4
2. `GSI2DART/work/mkmf_gsi_to_dart` change required: `-D_REAL4_`

If these are not set in a compatible fashion, you will fail to compile with the following error (or something similar):

```
../../../../observations/obs_converters/GSI2DART/dart_obs_seq_mod.f90(213): error
↪#6284:
There is no matching specific function for this generic function reference.   [SET_
↪LOCATION]
location = set_location(lon, lat, vloc, which_vert)
-----------------^
```

6.156 WOD Observations

6.156.1 Overview

The WOD (World Ocean Database) data is a collection of data from various sources, combined into a single format with uniform treatment. The WOD 2009 page has detailed information about the repository, observations, and datasets. The programs in this directory convert from the packed ASCII files found in the repository into DART observation sequence (obs_seq) file format.

There are 2 sets of available files - the raw observations, and the observations binned onto standard levels. The recommended datasets are the ones on standard levels. The raw data can be very dense in the vertical and are not truly independent observations. This leads to too much certainty in the updated values during the assimilation.

6.156.2 Data sources

Data from the WOD09 can be downloaded interactively from links on this page. One suggestion is to pick the 'sorted by year link' and download the files (you can select multiple years at once) for each data type for the years of interest. Make sure to select the standard level versions of each dataset.

UCAR/NCAR users with access to the DSS data repository can download WOD09 files from here. A UCAR DSS userid is required to access this page. The files to use are named "yearly_*_STD.tar".

Requested citation if you use this data:

```
Johnson, D.R., T.P. Boyer, H.E. Garcia, R.A. Locarnini, O.K. Baranova, and M.M. Zweng,
2009. World Ocean Database 2009 Documentation. Edited by Sydney Levitus. NODC
Internal Report 20, NOAA Printing Office, Silver Spring, MD, 175 pp.
Available at http://www.nodc.noaa.gov/OC5/WOD09/pr_wod09.html.
```

6.156.3 Programs

The data is distributed in a specialized packed ASCII format. In this directory is a program called `wodFOR.f` which is an example reader program to print out data values from the files. The program `wod_to_obs` converts these packed ASCII files into DART obs_sequence files.

As with most other DART directories, the `work` directory contains a `quickbuild.csh` script to build all necessary executables.

6.156.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&wod_to_obs_nml
   wod_input_file       =  'XBTS2005',
   wod_input_filelist   =  '',
   wod_out_file         =  'obs_seq.wod',
   avg_obs_per_file     =  500000,
   debug                =  .false.,
   timedebug            =  .false.,
   print_qc_summary     =  .true.,
   max_casts            =  -1,
   no_output_file       =  .false.,
   print_every_nth_cast =  -1,
   temperature_error    =  0.5,
   salinity_error       =  0.5,
 /
! temperature error is in degrees C, salinity error in g/kg.
```

Item	Type	Description
wod_input_file	character(len=128)	The input filename when converting a single file. Only one of the two namelist items that specify input files can have a valid value, so to use a single filename set the list name 'wod_input_filelist' to the empty string (' ').
wod_input_filelist	character(len=128)	To convert one or more files in a single execution create a text file which contains each input filename, in ascii, one filename per line. Set this item to the name of that file, and set 'wod_input_file' to the empty string (' ').
wod_out_file	character(len=128)	The output file to be created. Note that unlike earlier versions of some converters, this program will overwrite an existing output file instead of appending to it. The risk of replicated observations, which are difficult to detect since most of the contents are floating point numbers, outweighed the possible utility.
avg_obs_per_file	integer	The code needs an upper limit on the number of observations generated by this program. It can be larger than the actual number of observations converted. The total number of obs is computed by multiplying this number by the number of input files. If you get an error because there is no more room to add observations to the output file, increase this number. Do not make this an unreasonably huge number, however, since the code does preallocate space and will be slow if the number of obs becomes very large.
print_every_nth_cast	integer	If a value greater than 0, the program will print a message after processing every N casts. This allows the user to monitor the progress of the conversion.
print_qc_summary	logical	If .TRUE. the program will print out a summary of the number of casts which had a non-zero quality control values (current files appear to use values of 1-9).
debug	logical	If .TRUE. the program will print out debugging information.
timedebug	logical	If .TRUE. the program will print out specialized time-related debugging information.
max_casts	integer	If a value greater than 0 the program will only convert at most this number of casts from each input file. Generally only expected to be useful for debugging. A negative value will convert all data from the input file.
no_output_file	logical	If .TRUE. the converter will do all the work needed to convert the observations, count the number of each category of QC values, etc, but will not create the final obs_seq file. Can be useful if checking an input file for problems, or for getting QC statistics without waiting for a full output file to be constructed, which can be slow for large numbers of obs. Only expected to be useful for debugging.
temperature_error	real(r8)	The combined expected error of temperature observations from all sources, including instrument error, model bias, and representativeness error (e.g. larger or smaller grid box sizes affecting expected accuracy), in degrees Centigrade. Values in output file are error variance, which will be this value squared.
salinity_error	real(r8)	The combined expected error of salinity observations from all sources, including instrument error, model bias, and representativeness error (e.g. larger or smaller grid box sizes affecting expected accuracy) in g/kg (psu). Values in output file are error variance, and use units of msu (kg/kg), so the numbers will be this value / 1000.0, squared.

6.156.5 Modules used

```
types_mod
time_manager_mod
utilities_mod
location_mod
obs_sequence_mod
obs_def_mod
obs_def_ocean_mod
obs_kind_mod
```

6.156.6 Errors and known bugs

The code for setting observation error variances is using fixed values, and we are not certain if they are correct. Incoming QC values larger than 0 are suspect, but it is not clear if they really signal unusable values or whether there are some codes we should accept.

6.157 Total Precipitable Water Observations

6.157.1 Overview

Several satellites contain instruments that return observations of integrated Total Precipitable Water (TPW). There are two MODIS Spectroradiometers, one aboard the TERRA satellite, and the other aboard the AQUA satellite. There is also an AMSR-E instrument on the AQUA satellite.

These instruments produce a variety of data products which are generally distributed in HDF format using the HDF-EOS libraries. The converter code in this directory IS NOT USING THESE FILES AS INPUT. The code is expecting to read ASCII TEXT files, which contain one line per observation, with the latitude, longitude, TPW data value, and the observation time. The Fortran read line is:

```
read(iunit, '(f11.6, f13.5, f10.4, 4x, i4, 4i3, f7.3)') &
          lat, lon, tpw, iyear, imonth, iday, ihour, imin, seconds
```

No program to convert between the HDF and text files is currently provided. Contact dart@ucar.edu for more information if you are interested in using this converter.

6.157.2 Data sources

This converter reads files produced as part of a data research effort. Contact dart@ucar.edu for more information if you are interested in this data.

Alternatively, if you can read HDF-EOS files and output a text line per observation in the format listed above, then you can use this converter on TPW data from any MODIS file.

6.157.3 Programs

The programs in the DART/observations/tpw directory extract data from the distribution text files and create DART observation sequence (obs_seq) files. Build them in the work directory by running the ./quickbuild.csh script. In addition to the converters, several other general observation sequence file utilities will be built.

Generally the input data comes in daily files, with the string YYYYMMDD (year, month, day) as part of the name. This converter has the option to loop over multiple days within the same month and create an output file per day.

Like many kinds of satellite data, the TWP data is dense and generally needs to be subsampled or averaged (super-ob'd) before being used for data assimilation. This converter will average in both space and time. There are 4 namelist items (see the namelist section below) which set the centers and widths of time bins for each day. All observations within a single time bin are eligible to be averaged together. The next available observation in the bin is selected and any other remaining observations in that bin that are within delta latitude and delta longitude of it are averaged in both time and space. Then all observations which were averaged are removed from the bin, so each observation is only averaged into one output observation. Observations that are within delta longitude of the prime meridian are handled correctly by averaging observations on both sides of the boundary.

It is possible to restrict the output observation sequence to contain data from a region of interest using namelist settings. If your region spans the Prime Meridian min_lon can be a larger number than max_lon. For example, a region from 300 E to 40 E and 60 S to 30 S (some of the South Atlantic), specify *min_lon = 300, max_lon = 40, min_lat = -60, max_lat = -30*. So 'min_lon' sets the western boundary, 'max_lon' the eastern.

The specific type of observation created in the output observation sequence file can be select by namelist. "MODIS_TOTAL_PRECIPITABLE_WATER" is the most general term, or a more satellite-specific name can be chosen. The choice of which observations to assimilate or evaluate are made using this name. The observation-space diagnostics also aggregate statistics based on this name.

6.157.4 Namelist

This namelist is read from the file input.nml. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&convert_tpw_nml
   start_year          = 2008
   start_month         = 1
   start_day           = 1
   total_days          = 31
   max_obs             = 150000
   time_bin_start      = 0.0
   time_bin_interval   = 0.50
   time_bin_half_width = 0.25
   time_bin_end        = 24.0
   delta_lat_box       = 1.0
   delta_lon_box       = 1.0
   min_lon             =   0.0
   max_lon             = 360.0
   min_lat             = -90.0
   max_lat             =  90.0
   ObsBase             = '../data'
   InfilePrefix        = 'datafile.'
   InfileSuffix        = '.txt'
   OutfilePrefix       = 'obs_seq.'
   OutfileSuffix       = ''
   observation_name    = 'MODIS_TOTAL_PRECIPITABLE_WATER'
 /
```

Item	Type	Description
start_year	integer	The year for the first day to be converted. (The converter will optionally loop over multiple days in the same month.)
start_month	integer	The month number for the first day to be converted. (The converter will optionally loop over multiple days in the same month.)
start_day	integer	The day number for the first day to be converted. (The converter will optionally loop over multiple days in the same month.)
total_days	integer	The number of days to be converted. (The converter will optionally loop over multiple days in the same month.) The observations for each day will be created in a separate output file which will include the YYYYMMDD date as part of the output filename.
max_obs	integer	The largest number of obs in the output file. If you get an error, increase this number and run again.
time_bin_start	real(r8)	The next four namelist values define a series of time intervals that define time bins which are used for averaging. The input data from the satellite is very dense and generally the data values need to be subsetted in some way before assimilating. All observations in the same time bin are eligible to be averaged in space if they are within the latitude/longitude box. The input files are distributed as daily files, so use care when defining the first and last bins of the day. The units are in hours. This item defines the midpoint of the first bin.
time_bin_interval	real(r8)	Increment added the time_bin_start to compute the center of the next time bin. The units are in hours.
time_bin_half_width	real(r8)	The amount of time added to and subtracted from the time bin center to define the full bin. The units are in hours.
time_bin_end	real(r8)	The center of the last bin of the day. The units are in hours.
delta_lat_box	real(r8)	For all observations in the same time bin, the next available observation is selected. All other observations in that bin that are within delta latitude or longitude of it are averaged together and a single observation is output. Observations which are averaged with others are removed from the bin and so only contribute to the output data once. The units

6.158 ROMS observations to DART observation sequences

6.158.1 Overview

The relationship between ROMS and DART is slightly different than most other models. ROMS has the ability to apply its own forward operator as the model is advancing (a capability needed for variational assimilation) which produces something the ROMS community calls '*verification*' observations. The observation file that is input to ROMS is specified by the `s4dvar.in`:`OBSname` variable. The verification obs are written out to a netcdf file whose name is specified by the `s4dvar.in`:`MODname` variable. Since each ROMS model is advancing independently, a set of verification observation files are created during a DART/ROMS assimilation cycle. This set of files can be converted using `convert_roms_obs` to produce a DART observation sequence file that has precomputed forward operators (FOs). `convert_roms_obs` can also convert `s4dvar.in`:`OBSname`,`MODname` files to a DART observation sequence file that does not have the precomputed FOs.

The ROMS verification observation files **must** contain the **obs_provenance as a global attribute** and the following variables:

- *obs_lat, obs_lon, obs_depth*
- *obs_value*
- *obs_error*
- *obs_time*
- *NLmodel_value*
- *obs_scale*
- *obs_provenance*

Note that the *obs_provenance:flag_values*, and *obs_provenance:flag_meanings* attributes are totally ignored - those relationships are specified by the global attribute **obs_provenance**.

Locations only specified by *obs_Xgrid, obs_Ygrid, obs_depth* are **not** supported.

The conversion of a (set of) ROMS verification observations requires metadata to coordinate the relationship of the ROMS observation provenance to a DART observation TYPE. ROMS provides significant flexibility when specifying the observation provenance and it is simply impractical for DART to try to support all of them. An example of the current practice is described in the PROGRAMS section below.

Important: `filter` and `perfect_model_obs` must also be informed which DART observation types use precomputed forward operators. This is done by setting the `input.nml&obs_kind_nml` namelist. An example is shown at the end of the PROGRAMS section below.

6.158.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&convert_roms_obs_nml
   ens_size               = 1
   roms_mod_obs_files     = ''
   roms_mod_obs_filelist  = 'filelist.txt'
   dart_output_obs_file   = 'obs_seq.out'
   append_to_existing     = .false.
   use_precomputed_values = .true.
   add_random_noise       = .false.
   pert_amplitude         = 0.01
   verbose                = 0
   type_translations      = 'NULL'
  /
```

Item	Type	Description
ens_size	integer	Number of ensemble members which are expected to be found when creating the expected obs values. This must match the number of ROMS "mod" files listed in either the 'roms_mod_obs_files' or 'roms_mod_obs_filelist' namelist items. It is an error if they are not the same length.
roms_mod_obs_files	character(len=256), dimension(100)	List of filenames, one per ensemble member, that contain the observation values for each ensemble member. These are output from the ROMS program. If listing the files explicitly in this list, 'roms_mod_obs_filelist' must be ' ' (null).
roms_mod_obs_filelist	character(len=256)	The name of an ASCII file which contains, one per line, a list of filenames, one per ensemble member, that contain the expected obs values for each ensemble member. The filenames should NOT be quoted. These are output from the ROMS program. If using a filelist, then 'roms_mod_obs_files' must be ' ' (null).
dart_output_obs_file	character(len=256)	The name of the DART obs_seq file to create. If a file already exists with this name, it is either appended to or overwritten depending on the 'append_to_existing' setting below.
append_to_existing	logical	If an existing 'dart_output_obs_file' is found, this namelist item controls how it is handled. If .true. the new observations are appended to the existing file. If .false. the new observations overwrite the existing file.
use_precomputed_values	logical	flag to indicate that the output DART observation sequence file should include the verification observation values from all of the ROMS observation files. If `.true.` this will result in the DART file having the precomputed FOs to be used in the DART assimilation. If `.false.` this will result in DART files having the instrument values only.
add_random_noise	logical	Almost always should be .false. . The exception is the first cycle of an assimilation if all the ROMS input files are identical (no ensemble currently exists). To create differences in the forward operator values (since they are computed by ROMS), we can add gaussian noise here to give them perturbed values. This should be set as well as the "perturb_from_single_instance = .true." namelist in the `&filter_nml` namelist. After the first cycle, both these should be set back to .false. .
pert_amplitude	real(r8)	Ignored unless 'add_random_noise' is .true. . Controls the range of random values added to the expected obs values. Sets the width of a gaussian.
verbose	integer	If greater than 0, prints more information during the conversion.
type_translations	character(256), dimension(2, 100)	A set of strings which control the mapping of ROMS observation types to DART observation types. These should be specified in pairs. The first column should be a string that occurs in the global attribute '`obs_provenance`'. Note that the `obs_provenance:flag_values` and `obs_provenance:flag_meanings` attributes are ignored. The second column should be a DART specific obs type that is found in `DART/assimi lation_code/modules/observations/obs_kind_mod.f90`, which is created by the DART `preprocess` program.

6.158.3 Data sources

The origin of the input observation files used by ROMS are completely unknown to me.

6.158.4 Programs

- `convert_roms_obs`
- *PROGRAM obs_seq_to_netcdf*
- *program obs_sequence_tool*
- *PROGRAM preprocess*
- *PROGRAM advance_time*

Only `convert_roms_obs` will be discussed here.

The **global attribute** `obs_provenance` is used to relate the observation provenance to DART observation TYPES. The ROMS 'MODname' netCDF file(s) must have both the `obs_provenance` variable and a `obs_provenance` **global attribute**. The **exact** strings must be repeated in the DART `convert_roms_obs_nml:type_translations` variable to be able to convert from the integer value of the obs_provenance to th DART type in the following example:

`ncdump -h roms_mod_obs.nc` (the output has been pruned for clarity)

```
netcdf roms_mod_obs {
dimensions:
        record = 2 ;
        survey = 5376 ;
        state_var = 8 ;
        datum = 2407217 ;
variables:
        {snip}
        int obs_provenance(datum) ;
                obs_provenance:long_name = "observation origin" ;
                obs_provenance:flag_values = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ;
        double obs_time(datum) ;
                obs_time:long_name = "time of observation" ;
                obs_time:units = "days since 1900-01-01 00:00:00 GMT" ;
                obs_time:calendar = "gregorian" ;
        double obs_lon(datum) ;
                obs_lon:long_name = "observation longitude" ;
                obs_lon:units = "degrees_east" ;
        double obs_lat(datum) ;
                obs_lat:long_name = "observation latitude" ;
                obs_lat:units = "degrees_north" ;
        double obs_depth(datum) ;
                obs_depth:long_name = "ROMS internal depth of observation variable" ;
                obs_depth:units = "meters or fractional z-levels" ;
                obs_depth:negative_value = "downwards" ;
                obs_depth:missing_value = 1.e+37 ;
        double obs_error(datum) ;
                obs_error:long_name = "observation error covariance" ;
        double obs_value(datum) ;
                obs_value:long_name = "observation value" ;
        double obs_scale(datum) ;
                obs_scale:long_name = "observation screening/normalization scale" ;
                obs_scale:_FillValue = 0. ;
```

(continues on next page)

(continued from previous page)

```
        double NLmodel_value(datum) ;
                NLmodel_value:long_name = "nonlinear model at observation locations" ;
                NLmodel_value:_FillValue = 1.e+37 ;
        {snip}
     :obs_provenance = "\n",
             "1: gridded AVISO sea level anomaly (zeta)\n",
             "2: gridded Aquarius SSS (salinity)\n",
             "3: XBT from Met Office (temperature)\n",
             "4: CTD from Met Office (temperature)\n",
             "5: CTD from Met Office (salinity)\n",
             "6: ARGO floats (temperature)\n",
             "7: ARGO floats (salinity)\n",
             "8: glider UCSD (temperature)\n",
             "9: glider UCSD (salinity)\n",
             "10: blended satellite SST (temperature)" ;
        {snip}
```

Note the integer values that start the obs_provenance strings are used to interpret the integer contents of the obs_provenance variable. They need not be consecutive, nor in any particular order, but they must not appear more than once.

The following is the relevent section of the DART `input.nml`:

```
&convert_roms_obs_nml
   ens_size               = 32
   roms_mod_obs_filelist  = 'precomputed_files.txt'
   dart_output_obs_file   = 'obs_seq.out'
   append_to_existing     = .false.
   use_precomputed_values = .true.
   add_random_noise       = .false.
   verbose                = 1
   type_translations = "gridded AVISO sea level anomaly (zeta)", "SATELLITE_SSH",
                       "gridded Aquarius SSS (salinity)",        "SATELLITE_SSS",
                       "XBT from Met Office (temperature)",      "XBT_TEMPERATURE",
                       "CTD from Met Office (temperature)",      "CTD_TEMPERATURE",
                       "CTD from Met Office (salinity)",         "CTD_SALINITY",
                       "ARGO floats (temperature)",              "ARGO_TEMPERATURE",
                       "ARGO floats (salinity)",                 "ARGO_SALINITY",
                       "glider UCSD (temperature)",              "GLIDER_TEMPERATURE",
                       "glider UCSD (salinity)",                 "GLIDER_SALINITY",
                       "blended satellite SST (temperature)",    "SATELLITE_BLENDED_
↪SST"
  /
```

A complete list of DART observation TYPES is available in obs_def_ocean_mod.f90

Any or all of the DART observation types that appear in the second column of `type_translations` must also be designated as observations that have precomputed forward operators. This is done by setting the `input.nml&obs_kind_nml` namelist as follows:

```
&obs_kind_nml
   assimilate_these_obs_types =          'SATELLITE_SSH',
                                          'SATELLITE_SSS',
                                          'XBT_TEMPERATURE',
```

(continues on next page)

(continued from previous page)

```
                                          'CTD_TEMPERATURE',
                                          'CTD_SALINITY',
                                          'ARGO_TEMPERATURE',
                                          'ARGO_SALINITY',
                                          'GLIDER_TEMPERATURE',
                                          'GLIDER_SALINITY',
                                          'SATELLITE_BLENDED_SST'
   use_precomputed_FOs_these_obs_types = 'SATELLITE_SSH',
                                          'SATELLITE_SSS',
                                          'XBT_TEMPERATURE',
                                          'CTD_TEMPERATURE',
                                          'CTD_SALINITY',
                                          'ARGO_TEMPERATURE',
                                          'ARGO_SALINITY',
                                          'GLIDER_TEMPERATURE',
                                          'GLIDER_SALINITY',
                                          'SATELLITE_BLENDED_SST'
  /
```

6.159 PROGRAM COSMOS_to_obs

6.159.1 Overview

COSMOS "level 2" text file to DART converter

COSMOS is an NSF supported project to measure soil moisture on the horizontal scale of hectometers and depths of decimeters using cosmic-ray neutrons. The data for each station is available from the COSMOS data portal with several levels of processing. The metadata for each station (location, height, etc) is also available from the data portal. The **Level 2 Data** is most suited for use with DART.

Since each site has a separate input data file, and the metadata for each site must essentially be hand-input to the converter program, it is generally easiest to convert the observations for each site separately and then use the *program obs_sequence_tool* to combine the observations from multiple sites and restrict the DART observation sequence file to contain just the observations of the timeframe of interest.

FYI - in DART, the soil moisture profile is converted to expected neutron counts using the **CO**smic-ray **S**oil **M**oisture **I**nteraction **C**ode (COSMIC), developed at the University of Arizona by Rafael Rosolem and Jim Shuttleworth.

The workflow is usually:

1. get the site metadata and enter it in the `input.nml` *&COSMOS_to_obs_nml*
2. download the Level 2 Data and prefix the filename with the station name (or else they all get named `corcounts.txt`) and enter the filename into *&COSMOS_to_obs_nml*
3. make sure the station soil parameters and COSMIC parameters are contained in the `observations/COSMOS/data/COSMIC_parlist.nc` (more on this in the section on COSMIC parameters)
4. run `COSMOS_to_obs` to generate a DART observation sequence file for the station and rename the output file if necessary (you can explicity name the output file via the namelist).
5. repeat steps 1-4 for this converter to generate a DART observation sequence file for each station.
6. use the *program obs_sequence_tool* to combine the observations from multiple sites

6.159.2 Data sources

The COSMOS data portal can be found at: http://cosmos.hwr.arizona.edu/Probes/probemap.php The data for each station is available from the data portal with several levels of processing. The metadata for each station (location, height, etc) is also available from the data portal. The **Level 2 Data** is most suited for use with DART. An example of the Level 2 Data follows:

```
YYYY-MM-DD HH:MM  MOD PROBE PRESS  SCALE SANPE INTEN OTHER CORR ERR
2009-10-23 18:34 5996 0.800 1.087 06.901 2.486 1.062 1.000 1768 022
2009-10-23 19:34 5885 0.800 1.080 06.901 2.486 1.059 1.000 1729 022
2009-10-23 20:34 6085 0.800 1.072 06.901 2.486 1.059 1.000 1774 022
2009-10-23 21:34 6339 0.800 1.068 06.901 2.486 1.059 1.000 1843 023
...
```

6.159.3 Programs

The `COSMOS_to_obs.f90` file is the source code for the main converter program. At present there is an uncomfortable assumption that the order of the columns in the Level 2 data is fixed. I hope to relax that requirement in the near future. `COSMOS_to_obs` reads each text line into a character buffer and then reads from that buffer to parse up the data items. The items are then combined with the COSMIC parameters for that site and written to a DART-format observation sequence file. The DART format allows for the additional COSMIC parameters to be contained as metadata for each observation.

To compile and test, go into the `COSMOS/work` subdirectory and run the `quickbuild.csh` script to build the converter and a couple of general purpose utilities. The *program obs_sequence_tool* manipulates (i.e. combines, subsets) DART observation files once they have been created. The default observations supported are those defined in DART/observations/forward_operators/obs_def_land_mod.f90 and DART/observations/forward_operators/obs_def_COSMOS_mod.f90. If you need additional observation types, you will have to add the appropriate `obs_def_XXX_mod.f90` file to the `input.nml` `&preprocess_nml:input_files` variable and run `quickbuild.csh` again. It rebuilds the table of supported observation types before compiling the source code.

Guidance on COSMIC parameters

Additional information is needed by DART to convert soil moisture profiles to neutron counts. Each COSMOS instrument has site-specific parameters describing soil properties etc. Those parameters have been inserted into the observation file as metadata for each observation to simplify the DART observation operator. It is a bit redundant as currently implemented, but it is convenient.

`COSMOS_to_obs` reads the site name from the input namelist and the known station information from `COSMIC_parlist.nc`. The simplest way to add a new station to `COSMIC_parlist.nc` is probably to:

1. manually enter the information into the "data" section of `COSMIC_parlist_station.txt`
2. then use `ncgen` to convert `COSMIC_parlist_station.txt` to a netCDF file.
3. That netCDF file can be concatenated onto `COSMIC_parlist.nc` with a simple `ncrcat` command.

Listing the sites already supported is easy:

```
observations/COSMOS/data % ncdump -v sitenames COSMIC_parlist.nc
netcdf COSMIC_parlist {
dimensions:
        nsites = UNLIMITED ; // (42 currently)
        strlength = 21 ;
```

(continues on next page)

(continued from previous page)

```
variables:
        char sitenames(nsites, strlength) ;
                sitenames:long_name = "COSMOS Site Names" ;
        double longitude(nsites) ;
                longitude:long_name = "Longitude" ;
                longitude:units = "degrees" ;
        double latitude(nsites) ;
                latitude:long_name = "Latitude" ;
                latitude:units = "degrees" ;
        double elevation(nsites) ;
                elevation:long_name = "Elevation" ;
                elevation:units = "m" ;
        double bd(nsites) ;
                bd:long_name = "Dry Soil Bulk Density" ;
                bd:units = "g cm{-3}" ;
        double lattwat(nsites) ;
                lattwat:long_name = "Lattice Water Content" ;
                lattwat:units = "m{3} m{-3}" ;
        double N(nsites) ;
                N:long_name = "High Energy Neutron Intensity" ;
                N:units = "relative counts" ;
        double alpha(nsites) ;
                alpha:long_name = "Ratio of Fast Neutron Creation Factor (Soil to␣
↪Water)" ;
                alpha:units = "-" ;
        double L1(nsites) ;
                L1:long_name = "High Energy Soil Attenuation Length" ;
                L1:units = "g cm{-2}" ;
        double L2(nsites) ;
                L2:long_name = "High Energy Water Attenuation Length" ;
                L2:units = "g cm{-2}" ;
        double L3(nsites) ;
                L3:long_name = "Fast Neutron Soil Attenuation Length" ;
                L3:units = "g cm{-2}" ;
        double L4(nsites) ;
                L4:long_name = "Fast Neutron Water Attenuation Length" ;
                L4:units = "g cm{-2}" ;

// global attributes:
                :website = "COsmic-ray Soil Moisture Observing System (COSMOS) -
                            http://cosmos.hwr.arizona.edu" ;
data:

 sitenames =
  "ARM-1                ",
  "Austin_Cary          ",
  "Bondville            ",
  "Brookings            ",
  "Chestnut_Ridge_NOAA  ",
  "Coastal_Sage_UCI     ",
  "Daniel_Forest        ",
  "Desert_Chaparral_UCI ",
  "Fort_Peck            ",
  "Harvard_Forest       ",
  "Hauser_Farm_North    ",
  "Hauser_Farm_South    ",
  "Howland              ",
```

(continues on next page)

(continued from previous page)

```
  "Iowa_Validation_Site ",
  "Island_Dairy         ",
  "JERC                 ",
  "Kendall              ",
  "KLEE                 ",
  "Manitou_Forest_Ground",
  "Metolius             ",
  "Morgan_Monroe        ",
  "Mozark               ",
  "Mpala_North          ",
  "Neb_Field_3          ",
  "P301                 ",
  "Park_Falls           ",
  "Pe-de-Gigante        ",
  "Rancho_No_Tengo      ",
  "Reynolds_Creek       ",
  "Rietholzbach         ",
  "Rosemount            ",
  "San_Pedro_2          ",
  "Santa_Rita_Creosote  ",
  "Savannah_River       ",
  "Silver_Sword         ",
  "SMAP-OK              ",
  "Soaproot             ",
  "Sterling             ",
  "Tonzi_Ranch          ",
  "UMBS                 ",
  "UVA                  ",
  "Wind_River           " ;
}
```

The observation sequence files will look something like the following, the attributes in yellow are the information from `COSMIC_parlist.nc`:

```
 obs_sequence
obs_kind_definitions
           1
          20 COSMOS_NEUTRON_INTENSITY
  num_copies:            1  num_qc:            1
  num_obs:          3840  max_num_obs:          3840
observation
COSMOS QC
  first:            1  last:         3840
 OBS            1
   1048.0000000000000
   1.0000000000000000
          -1           2          -1
obdef
loc3d
     4.154723123116714        0.7997185899100618         0.000000000000000     -1
kind
          20
```

cosmic 0.88500000000000001 5.84099999999999966E-002 336.95696938999998 0.31918025877000000 161.98621864285701 129.14558984999999 55.311849408000000 3.8086191933000002 1

```
77340     150034
  1225.0000000000000
  ...
```

6.159.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&COSMOS_to_obs_nml
   site_metadata_file = 'COSMIC_parlist.nc'
   text_input_file    = 'textdata.input',
   obs_out_file       = 'obs_seq.out',
   sitename           = 'missing',
   maxgoodqc          =  3,
   verbose            = .false.
   /
```

Contents	Type	Description
site_metadata_file	character(len=256)	The netCDF file containing the parameter values for each site.
text_input_file	character(len=128)	The text file containing the raw observations for each site.
obs_out_file	character(len=128)	The output observation sequence file for DART.
sitename	character(len=128)	The name of the site. Must match one of the site names in the `site_metadata_file`. Case-insensitive match, trailing blanks ignored. Use *ncdump -v sitenames COSMIC_parlist.nc*
maxgoodqc	integer	left for future implementation.
verbose	logical	A switch to specify the amount of run-time output. `.true.` the most amount of output. `.false.` the least amount of output.

Cosmos_to_obs namelist

```
&COSMOS_to_obs_nml
   site_metadata_file = 'COSMIC_parlist.nc',
   text_input_file    = 'SantaRita_corcounts.txt',
   obs_out_file       = 'SantaRita_obs_seq.out',
   sitename           = 'Santa_Rita_Creosote',
```

6.159.5 References

- The COSMOS web page.
- Franz, T.E, M. Zreda, T.P.A. Ferre, R. Rosolem, C. Zweck, S. Stillman, X. Zeng and W.J. Shuttleworth, 2012: Measurement depth of the cosmic-ray soil moisture probe affected by hydrogen from various sources. Water Resources Research 48, W08515, doi:10.1029/2012WR011871
- Franz, T.E, M. Zreda, R. Rosolem, T.P.A. Ferre, 2012: Field validation of cosmic-ray soil moisture probe using a distributed sensor network. Vadose Zone Journal (in press), doi:10.2136/vzj2012.0046

6.160 PROGRAM `COSMOS_development`

6.160.1 Overview

Trial COSMOS text file to DART converter

COSMOS is an NSF supported project to measure soil moisture on the horizontal scale of hectometers and depths of decimeters using cosmic-ray neutrons. The data for each station is available from the COSMOS data portal with several levels of processing. The metadata for each station (location, height, etc) is also available from the data portal. The **Level 2 Data** is most suited for use with DART, but does not currently have a correction for the amount of hydrogen in the atmospheric volume near the probe. To this end, Rafael Rosolem has a separate data stream. `COSMOS_development` reads Rafaels data streams and converts them to DART observation sequence files. **Since these data streams are not widespread, we recommend using :doc:`./COSMOS_to_obs`.**
The workflow is usually:

1. get the site metadata and enter it in the `input.nml` *&COSMOS_development_nml*
2. acquire the development observation data and prefix the filename with the station name (or else they all get named `corcounts.txt`) and enter the filename into *&COSMOS_development_nml*
3. make sure the station soil parameters and COSMIC parameters are contained in the `observations/COSMOS/data/COSMIC_parlist.nc` (more on this in the section on COSMIC parameters)
4. run `COSMOS_development` to generate a DART observation sequence file for the station and rename the output file if necessary (you can explicity name the output file via the namelist).
5. repeat steps 1-4 for this converter to generate a DART observation sequence file for each station.
6. use the *program obs_sequence_tool* to combine the observations from multiple sites

6.160.2 Data sources

The COSMOS data portal can be found at: http://cosmos.hwr.arizona.edu/Probes/probemap.php The development observation data for each station is generally not available. The metadata for each station (location, height, etc) is also available from the data portal. The **Level 2 Data** is most suited for use with DART. **We recommend using :doc:`./COSMOS_to_obs`.** An example of the development observation data follows:

```
month,day,hour,doy,neutron_fluxAVE,neutron_fluxSTD,neutron_fluxQC
 1,  1,  0,   1,-9999,9999,3
 1,  1,  1,   1,-9999,9999,3
 1,  1,  2,   1,-9999,9999,3
 1,  1,  3,   1,-9999,9999,3
...
```

6.160.3 Programs

The COSMOS_development.f90 file is the source code for the main converter program. At present there is an uncomfortable assumption that the order of the columns in the Level 2 data is fixed. I hope to relax that requirement in the near future. COSMOS_development reads each text line into a character buffer and then reads from that buffer to parse up the data items. The items are then combined with the COSMIC parameters for that site and written to a DART-format observation sequence file. The DART format allows for the additional COSMIC parameters to be contained as metadata for each observation.

To compile and test, go into the COSMOS/work subdirectory and run the quickbuild.csh script to build the converter and a couple of general purpose utilities. The *program obs_sequence_tool* manipulates (i.e. combines, subsets) DART observation files once they have been created. The default observations supported are those defined in observations/forward_operators/obs_def_land_mod.f90 and observations/forward_operators/obs_def_COSMOS_mod.f90. If you need additional observation types, you will have to add the appropriate obs_def_XXX_mod.f90 file to the input.nml &preprocess_nml:input_files variable and run quickbuild.csh again. It rebuilds the table of supported observation types before compiling the source code.

COSMIC parameters

Additional information is needed by DART to convert soil moisture profiles to neutron counts. Each COSMOS instrument has site-specific parameters describing soil properties etc. Those parameters have been inserted into the observation file as metadata for each observation to simplify the DART observation operator. It is a bit redundant as currently implemented, but it is convenient.

COSMOS_development reads the site name from the input namelist and the known station information from COSMIC_parlist.nc. The simplest way to add a new station to COSMIC_parlist.nc is probably to:

1. manually enter the information into the "data" section of COSMIC_parlist_station.txt
2. then use ncgen to convert COSMIC_parlist_station.txt to a netCDF file.
3. That netCDF file can be concatenated onto COSMIC_parlist.nc with a simple ncrcat command.

Listing the sites already supported is easy:

```
observations/COSMOS/data % ncdump -v sitenames COSMIC_parlist.nc
netcdf COSMIC_parlist {
dimensions:
        nsites = UNLIMITED ; // (42 currently)
        strlength = 21 ;
variables:
        char sitenames(nsites, strlength) ;
                sitenames:long_name = "COSMOS Site Names" ;
        double longitude(nsites) ;
                longitude:long_name = "Longitude" ;
                longitude:units = "degrees" ;
        double latitude(nsites) ;
                latitude:long_name = "Latitude" ;
                latitude:units = "degrees" ;
        double elevation(nsites) ;
                elevation:long_name = "Elevation" ;
                elevation:units = "m" ;
        double bd(nsites) ;
                bd:long_name = "Dry Soil Bulk Density" ;
                bd:units = "g cm{-3}" ;
        double lattwat(nsites) ;
                lattwat:long_name = "Lattice Water Content" ;
```

(continues on next page)

(continued from previous page)

```
                lattwat:units = "m{3} m{-3}" ;
        double N(nsites) ;
                N:long_name = "High Energy Neutron Intensity" ;
                N:units = "relative counts" ;
        double alpha(nsites) ;
                alpha:long_name = "Ratio of Fast Neutron Creation Factor (Soil to␣
↪Water)" ;
                alpha:units = "-" ;
        double L1(nsites) ;
                L1:long_name = "High Energy Soil Attenuation Length" ;
                L1:units = "g cm{-2}" ;
        double L2(nsites) ;
                L2:long_name = "High Energy Water Attenuation Length" ;
                L2:units = "g cm{-2}" ;
        double L3(nsites) ;
                L3:long_name = "Fast Neutron Soil Attenuation Length" ;
                L3:units = "g cm{-2}" ;
        double L4(nsites) ;
                L4:long_name = "Fast Neutron Water Attenuation Length" ;
                L4:units = "g cm{-2}" ;

// global attributes:
                :website = "COsmic-ray Soil Moisture Observing System (COSMOS) -
                            http://cosmos.hwr.arizona.edu" ;
data:

 sitenames =
  "ARM-1                ",
  "Austin_Cary          ",
  "Bondville            ",
  "Brookings            ",
  "Chestnut_Ridge_NOAA  ",
  "Coastal_Sage_UCI     ",
  "Daniel_Forest        ",
  "Desert_Chaparral_UCI ",
  "Fort_Peck            ",
  "Harvard_Forest       ",
  "Hauser_Farm_North    ",
  "Hauser_Farm_South    ",
  "Howland              ",
  "Iowa_Validation_Site ",
  "Island_Dairy         ",
  "JERC                 ",
  "Kendall              ",
  "KLEE                 ",
  "Manitou_Forest_Ground",
  "Metolius             ",
  "Morgan_Monroe        ",
  "Mozark               ",
  "Mpala_North          ",
  "Neb_Field_3          ",
  "P301                 ",
  "Park_Falls           ",
  "Pe-de-Gigante        ",
  "Rancho_No_Tengo      ",
  "Reynolds_Creek       ",
  "Rietholzbach         ",
```

(continues on next page)

(continued from previous page)

```
  "Rosemount              ",
  "San_Pedro_2            ",
  "Santa_Rita_Creosote    ",
  "Savannah_River         ",
  "Silver_Sword           ",
  "SMAP-OK                ",
  "Soaproot               ",
  "Sterling               ",
  "Tonzi_Ranch            ",
  "UMBS                   ",
  "UVA                    ",
  "Wind_River             " ;
}
```

The observation sequence files will look something like the following, the attributes in yellow are the information from `COSMIC_parlist.nc`:

```
 obs_sequence
obs_kind_definitions
           1
          20 COSMOS_NEUTRON_INTENSITY
  num_copies:            1  num_qc:            1
  num_obs:          3840  max_num_obs:         3840
observation
COSMOS QC
  first:            1  last:         3840
 OBS              1
   1048.0000000000000
   1.0000000000000000
          -1           2          -1
obdef
loc3d
     4.154723123116714        0.7997185899100618         0.000000000000000     -1
kind
          20
```

cosmic 0.88500000000000001 5.84099999999999966E-002 336.95696938999998 0.31918025877000000 161.98621864285701 129.14558984999999 55.311849408000000 3.8086191933000002 1

```
77340     150034
  1225.0000000000000
  ...
```

6.160.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&COSMOS_development_nml
   site_metadata_file = 'COSMIC_parlist.nc'
   text_input_file    = 'textdata.input',
   obs_out_file       = 'obs_seq.out',
   sitename           = 'missing',
```

(continues on next page)

(continued from previous page)

```
   year                 = -1
   maxgoodqc            =  3,
   verbose              = .false.
   /
```

Contents	Type	Description
site_metadata_file	character(len=256)	The netCDF file containing the parameter values for each site.
text_input_file	character(len=128)	The text file containing the raw observations for each site.
obs_out_file	character(len=128)	The output observation sequence file for DART.
sitename	character(len=128)	The name of the site. Must match one of the site names in the `site_metadata_file`. Case-insensitive match, trailing blanks ignored. Use *ncdump -v sitenames COSMIC_parlist.nc*
year	integer	The year of the data.
maxgoodqc	integer	left for future implementation.
verbose	logical	A switch to specify the amount of run-time output. `.true.` the most amount of output. `.false.` the least amount of output.

COSMOS development namelist

```
&COSMOS_development_nml
   site_metadata_file = '../data/COSMIC_parlist.nc',
   text_input_file    = 'SantaRita_corcounts.txt',
   obs_out_file       = 'SantaRita_obs_seq.out',
   sitename           = 'Santa_Rita_Creosote',
```

6.160.5 References

- The COSMOS web page.
- Franz, T.E, M. Zreda, T.P.A. Ferre, R. Rosolem, C. Zweck, S. Stillman, X. Zeng and W.J. Shuttleworth, 2012: Measurement depth of the cosmic-ray soil moisture probe affected by hydrogen from various sources. Water Resources Research 48, W08515, doi:10.1029/2012WR011871
- Franz, T.E, M. Zreda, R. Rosolem, T.P.A. Ferre, 2012: Field validation of cosmic-ray soil moisture probe using a distributed sensor network. Vadose Zone Journal (in press), doi:10.2136/vzj2012.0046

6.161 PROGRAM `littler_tf_dart`

6.161.1 Overview

Programs to convert littler data files into DART observation sequence files, and vice versa. The capability of the program is limited to wind and temperature from radiosondes.

The littler data files do not contain observation errors. The observation errors are in a separate file called `obserr.txt`. The littler file generated here has to be preprocessed by the program `3dvar_obs.exe` before beeing ingested in the WRF 3D-Var system.

6.161.2 Modules used

```
types_mod
obs_sequence_mod
obs_def_mod
obs_kind_mod
location/threed_sphere/location_mod
time_manager_mod
utilities_mod
```

6.161.3 Modules indirectly used

```
assim_model_mod
models/wrf/model_mod
models/wrf/module_map_utils
random_seq_mod
```

6.161.4 Namelist

The program does not have its own namelist. However, an `input.nml` file is required for the modules used by the program.

6.161.5 Files

- input namelist ; `input.nml`
- Input - output observation files; `obs_seq.out` and `little-r.dat`
- Input - output littler observation error files ; `obserr.txt`

File formats

If there are no observation error at a particular pressure level, the default value of -1 is written in `obserr.txt`.

6.161.6 References

- 3DVAR GROUP PAGE

6.161.7 Private components

call set_str_date(timestring, dart_time)

```
type(time_type),    intent(in)  ::  dart_time
character(len=20), intent(out) ::  timestring
```

Given a dart_time (seconds, days), returns date as bbbbbbyyyymmddhhmmss, where b is a blank space.

call set_dart_time(tstring, dart_time)

```
character(len=20), intent(in)  ::  tstring
type(time_type),    intent(out) ::  dart_time
```

Given a date as bbbbbbyyyymmddhhmmss, where b is a blank space, returns the dart_time (seconds, days).

call StoreObsErr(obs_err_var, pres, plevel, nlev, obs_err_std)

```
integer,  intent(in)     ::  nlev, pres
real(r8), intent(in)     ::  obs_err_var
integer,  intent(in)     ::  plevel(nlev)
real(r8), intent(inout) ::  obs_err_std(nlev)
```

If the incoming pres corresponds exactly to a pressure level in plevel, then transfers the incoming obs_err_var into the array obs_err_std at the corresponding level.

level_index = GetClosestLevel(ilev, vlev, nlev)

```
integer,  intent(in) ::  nlev, ilev
integer,  intent(in) ::  vlev(nlev)
```

Returns the index of the closest level in vlev to the incoming ilev.

call READ_OBSERR(filein, platform, sensor_name, err, nlevels)

```
CHARACTER (LEN=80), intent(in)  ::  filein
CHARACTER (LEN=80), intent(in)  ::  platform
CHARACTER (LEN=80), intent(in   ::  sensor_name
INTEGER,            intent(in)  ::  nlevels
REAL(r8),           intent(out) ::  err(nlevels)
```

Read observational error on pressure levels (in hPa) from the incoming filein and store the result in the array err. It is assumed that filein has the same format as WRF 3D-Var `obserr.txt` file. It reads observational error for a specific platform (e.g. RAOBS) and a specific sensor (e.g. WIND SENSOR ERRORS).

f_obstype = obstype(line)

```
CHARACTER (LEN= 80), intent(in) ::  line
```

Read in a line the string present after keyword 'BOGUS', which should be the sensor name.

f_sensor = sensor(line)

```
CHARACTER (LEN= 80), intent(in) ::  line
```

Read in a line the string present after numbers, which should be the platform name.

val = intplin(x,xx,yy)

```
INTEGER,  DIMENSION (:), intent(in) ::  xx
REAL(r8), DIMENSION (:), intent(in) ::  yy
REAL(r8),                intent(in) ::  x
```

Do a linear interpolation.

val = intplog(x,xx,yy)

```
INTEGER,  DIMENSION (:), intent(in) ::  xx
REAL(r8), DIMENSION (:), intent(in) ::  yy
REAL(r8),                intent(in) ::  x
```

Do a log-linear interpolation.

index = locate(x,xx)

```
INTEGER, DIMENSION (:), intent(in) ::  xx
REAL(r8),               intent(in) ::  x
```

Return the index in xx such that xx(index) < x < xx(index+1).

6.162 PROGRAM `rad_3dvar_to_dart`

6.162.1 Overview

Programs to convert MM5 3D-VAR 2.0 Radar data files into DART observation sequence files. The capability of the program is limited to DOPPLER_RADIAL_VELOCITY and RADAR_REFLECTIVITY.

6.162.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&rad_3dvar_to_dart_nml
   var_file = 'qc_radr_3dvar_2002083100.dat',
   obs_seq_out_file_name = 'obs_seq.out',
   calendar_type = 3
/
```

Item	Type	Description
var_file	character(len=129)	This is the name of the file containing MM5 3D-VAR 2.0 Radar observations.
obs_seq_out_file_name	character(len=129)	File name for output observation sequence file.
calendar_type	integer	Calendar type. We recommend using 3 (GREGORIAN).

6.162.3 Modules directly used

```
types_mod
obs_sequence_mod
obs_def_mod
obs_def/obs_def_radar_mod
obs_kind_mod
location/threed_sphere/location_mod
time_manager_mod
utilities_mod
```

6.162.4 Modules indirectly used

```
assim_model_mod
models/wrf/model_mod
models/wrf/module_map_utils
random_seq_mod
```

6.162.5 Files

- input namelist ; `input.nml`
- Input observation file; `qc_radr_3dvar_2002083100.dat`
- Output observation file; `obs_seq.out`

File formats

`input.nml` and `qc_radr_3dvar_2002083100.dat` are ASCII files. `obs_seq.out` is either ASCII or binary, depending on the logical write_binary_obs_sequence, which is the namelist entry for obs_sequence_mod.

6.162.6 References

- 3DVAR GROUP PAGE

6.163 3DVAR/4DVAR Observation Converters

6.163.1 Overview

The programs in this directory help convert data which is formatted for input into the 3DVAR/4DVAR programs into DART obs_seq observation files.

See the README file in this directory for more information.

6.163.2 Data sources

6.163.3 Programs

6.164 PROGRAM `tc_to_obs`

6.165 Tropical Cyclone ATCF File to DART Converter

6.165.1 Overview

Tropical Cyclone data created by the 'Automated Tropical Cyclone Forecast (ATCF) System' can be converted into DART observations of the storm center location, minimum sea level pressure, and maximum wind speed. Several of the options can be customized at runtime by setting values in a Fortran namelist. See the namelist section below for more details. In the current release of DART only the *WRF* has forward operator code to generate expected obs values for these vortex observations.

This webpage documents many things about the ATCF system and the various file formats that are used for storm track data and other characteristics.

The converter in this directory is only configured to read the packed "b-deck" format (as described on the webpage referenced above). There are sections in the fortran code which can be filled in to read other format variants. This should mostly be a matter of changing the read format string to match the data in the file.

6.165.2 Data sources

A collection of past storm ATCF information can be found here. For each observation you will need a location, a data value, a type, a time, and some kind of error estimate. The error estimates will need to be hardcoded or computed in the converter since they are not available in the input data. See below for more details on selecting an appropriate error value.

6.165.3 Programs

The tc_to_obs.f90 file is the source for the main converter program. Look at the source code where it reads the example data file. Given the variety of formatting details in different files, you may quite possibly need to change the "read" statement to match your data format. There is a 'select case' section which is intended to let you add more formats and select them at runtime via namelist.

To compile and test, go into the work subdirectory and run the quickbuild.csh script to build the converter and a couple of general purpose utilities. advance_time helps with calendar and time computations, and the obs_sequence_tool manipulates DART observation files once they have been created.

This converter creates observation types defined in the DART/observations/forward_operators/obs_def_vortex_mod.f90 file. This file must be listed in the input.nml namelist file, in the &preprocess_nml namelist, in the 'input_files' variable, for any programs which are going to process these observations. If you have to change the &preprocess_nml namelist you will have to run quickbuild.csh again to build and execute the preprocess program before compiling other executables. It remakes the table of supported observation types before trying to recompile other source code.

There is an example b-deck data file in the data directory. This format is what is supported in the code as distributed. There are other variants of this format which have more spaces so the columns line up, and variants which have many more fields than what is read here.

6.165.4 Specifying expected error

The ATCF files DO NOT include any estimated error values. The source code currently has hardcoded values for location, sea level pressure, and max wind errors. These may need to be adjusted as needed if they do not give the expected results.

6.165.5 Namelist

This namelist is read from the file input.nml. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&tc_to_obs_nml
   input_atcf_file         = 'input.txt'
   fileformat              = 'b-deck'
   obs_out_file            = 'obs_seq.out'
   append_to_existing_file = .false.
   debug                   = .false.
 /
```

Item	Type	Description
input_atcf_file	character(len=256)	Name of the input ascii text file in ATCF format.
fileformat	character(len=128)	Currently only supports 'b-deck' but if other format strings are added, can switch at runtime between reading different varieties of ATCF file formats.
obs_out_file	character(len=256)	Name of the output observation sequence file to create.
append_to_existing_file	logical	If .false., this program will overwrite an existing file. If .true. and if a file already exists with the same name the newly converted observations will be appended to that file. Useful if you have multiple small input files that you want to concatenate into a single output file. However, there is no code to check for duplicated observations. If this is .true. and you run the converter twice you will get duplicate observations in the file which is bad. (It will affect the quality of your assimilation results.) Use with care. You can concatenate multiple obs sequence files as a postprocessing step with the :doc :*../../../assimilation_code/programs /obs_sequence_tool/obs_sequence_tool* which comes with DART and is in fact built by the quickbuild.csh script in the TC converter work directory.
debug	logical	Set to .true. to print out more details during the conversion process.

6.166 PROGRAM `level4_to_obs`

6.166.1 Overview

AmeriFlux level 4 data to DART observation sequence converter

This routine is designed to convert the flux tower Level 4 data from the AmeriFlux network of observations from micrometeorological tower sites. AmeriFlux is part of FLUXNET and the converter is hoped to be a suitable starting point for the conversion of observations from FLUXNET. As of May 2012, I have not yet tried to work with any other observations from FLUXNET.

The AmeriFlux Level 4 products are recorded using the local time. DART observation sequence files use GMT. For more information about AmeriFlux data products, go to http://ameriflux.lbl.gov.

> **Warning:** There was a pretty severe bug in the converter that swapped latent heat flux and sensible heat flux. The bug was present through revision 7200. It has been corrected in all subsequent versions.

The workflow is usually:

1. download the Level 4 data for the towers and years in question (see DATA SOURCES below)
2. record the TIME ZONE, latitude, longitude, and elevation for each tower
3. build the DART executables with support for the tower observations. This is done by running `preprocess` with `obs_def_tower_mod.f90` in the list of `input_files` for `preprocess_nml`.

4. provide basic tower information via the `level4_to_obs_nml` namelist since this information is not contained in the Level 4 data file
5. convert each Level 4 data file individually using `level4_to_obs`
6. combine all output files for the region and timeframe of interest into one file using *program obs_sequence_tool*

For some models (CLM, for example), it is required to reorganize the observation sequence files into a series of files that contains ONLY the observations for each assimilation. This can be achieved with the makedaily.sh script.

6.166.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&level4_to_obs_nml
   text_input_file = 'textdata.input',
   obs_out_file    = 'obs_seq.out',
   year            = -1,
   timezoneoffset  = -1,
   latitude        = -1.0,
   longitude       = -1.0,
   elevation       = -1.0,
   flux_height     = -1.0,
   maxgoodqc       = 3,
   verbose         = .false.
   /
```

Contents	Type	Description
text_input_file	character(len=128)	Name of the Level 4 ASCII file of comma-separated values. This may be a relative or absolute filename.
obs_out_file	character(len=128)	Name of the output observation sequence file.
year	integer	The year of the observations in the Level 4 text file.
timezoneoffset	real	the time zone offset (in hours) of the station. The tower observation times are local time, we need to convert them to GMT.
latitude	real	Latitude (in degrees N) of the tower.
longitude	real	Longitude (in degrees E) of the tower. For internal consistency, DART uses longitudes in the range [0,360]. An input value of -90 will be converted to 270, for example.
elevation	real	surface elevation (in meters) of the tower.
flux_height	real	height (in meters) of the flux instrument on the tower.
maxgoodqc	real	maximum value of any observation quality control flag to pass through to the output observation sequence. Keep in mind that `filter` has the ability to discriminate on the value, so there is really little to be gained by rejecting them during the conversion.
verbose	logical	Print extra information during the `level4_to_obs` execution.

6.166.3 Data sources

The data was acquired from http://cdiac.ornl.gov/ftp/ameriflux/data/Level4/Sites_ByName

and have names like `USBar2004_L4_h.txt, USHa12004_L4_h.txt, USNR12004_L4_h.txt, USSP32004_L4_h.txt, USSRM2004_L4_h.txt, USWCr2004_L4_h.txt, USWrc2004_L4_h.txt, ...`

The Level 4 products in question are ASCII files of comma-separated values taken every 30 minutes for an entire year. The first line is a comma-separated list of column descriptors, all subsequent lines are comma-separated numerical values. The converter presently searches for the columns pertaining to *NEE_or_fMDS*, *H_f*, *LE_f*, their corresponding quality control fields, and those columns pertaining to the time of the observation. These values are mapped as follows:

Level 4 units

Level 4 variable

description

DART type

DART kind

DART units

W/m^2

LE_f

Latent Heat Flux

TOWER_LATENT_HEAT_FLUX

QTY_LATENT_HEAT_FLUX

W/m^2

[0-3]

LE_fqc

QC for LE_f

N/A

N/A

same

W/m^2

H_f

Sensible Heat Flux

TOWER_SENSIBLE_HEAT_FLUX

QTY_SENSIBLE_HEAT_FLUX

W/m^2

[0-3]

H_fqc

QC for H_f

N/A

N/A

same

umolCO2/m^2/s

NEE_or_fMDS

Net Ecosystem Production

TOWER_NETC_ECO_EXCHANGE

QTY_NET_CARBON_PRODUCTION

gC/m^2/s

[0-3]

NEE_or_fMDSqc

QC for NEE_or_fMDS

N/A

N/A

same

The `LE_fqc`, `H_fqc`, and `NEE_or_fMDSqc` variables use the following convention:

> 0 = original, 1 = category A (most reliable), 2 = category B (medium), 3 = category C (least reliable). (Refer to Reichstein et al. 2005 Global Change Biology for more information)

I am repeating the AmeriFlux Data Fair-Use Policy because I believe it is important to be a good scientific citizen:

> "The AmeriFlux data provided on this site are freely available and were furnished by individual AmeriFlux scientists who encourage their use. Please kindly inform in writing (or e-mail) the appropriate AmeriFlux scientist(s) of how you intend to use the data and of any publication plans. It is also important to contact the AmeriFlux investigator to assure you are downloading the latest revision of the data and to prevent potential misuse or misinterpretation of the data. Please acknowledge the data source as a citation or in the acknowledgments if no citation is available. If the AmeriFlux Principal Investigators (PIs) feel that they should be acknowledged or offered participation as authors, they will let you know and we assume that an agreement on such matters will be reached before publishing and/or use of the data for publication. If your work directly competes with the PI's analysis they may ask that they have the opportunity to submit a manuscript before you submit one that uses unpublished data. In addition, when publishing please acknowledge the agency that supported the research. Lastly, we kindly request that those publishing papers using AmeriFlux data provide reprints to the PIs providing the data and to the AmeriFlux archive via ameriflux.lbl.gov."

6.166.4 Programs

The `level4_to_obs.f90` file is the source for the main converter program. Look at the source code where it reads the example data file. You will almost certainly need to change the "read" statement to match your data format. The example code reads each text line into a character buffer and then reads from that buffer to parse up the data items.

To compile and test, go into the work subdirectory and run the `quickbuild.csh` script to build the converter and a couple of general purpose utilities. `advance_time` helps with calendar and time computations, and the `obs_sequence_tool` manipulates DART observation files once they have been created.

To change the observation types, look in the `DART/obs_def` directory. If you can find an obs_def_XXX_mod.f90 file with an appropriate set of observation types, change the 'use' lines in the converter source to include those types. Then add that filename in the `input.nml` namelist file to the &preprocess_nml namelist, the 'input_files' variable. Multiple files can be listed. Then run quickbuild.csh again. It remakes the table of supported observation types before trying to recompile the source code.

An example script for converting batches of files is in the `shell_scripts` directory. A tiny example data file is in the `data` directory. These are *NOT* intended to be turnkey scripts; they will certainly need to be customized for your use. There are comments at the top of the script saying what options they include, and should be commented enough to indicate where changes will be likely to need to be made.

6.166.5 Decisions you might need to make

See the discussion in the obs_converters/README.md page about what options are available for the things you need to specify. These include setting a time, specifying an expected error, setting a location, and an observation type.

6.167 PROGRAM `cice_to_obs`

6.167.1 Overview

Sea ice percentage observations to DART converter

This converter reads the binary sea ice observations from the snow and ice data center files and outputs DART obs_seq format files. It will loop over multiple days inside a single run of the converter program.

6.167.2 Data sources

The National Snow and Ice Data Center supplies the data files read by this converter. (I believe it is this format?)

6.167.3 Programs

The `cice_to_obs.f90` file is the source for the main converter program. More documentation is in the source code file especially around where the namelist variables are declared.

6.168 PROGRAM `dwl_to_obs`

6.168.1 Overview

DWL to DART converter

These are Doppler Wind Lidar measurements which have previously been extracted from the incoming format and output in ascii format, one pair of wind component observations per line. This converter reads in the ascii file and outputs the data in DART observation sequence (obs_seq) format.

This is OSSE data from a satellite which is expected to be launched in 2015. Information on the satellite mission is here at http://en.wikipedia.org/wiki/ADM-Aeolus.

The workflow is:

- read in the needed information about each observation - location, time, observation values, obs errors - from an ascii file
- call a series of DART library routines to construct a derived type that contains all the information about a single observation
- call another set of DART library routines to put it into a time-sorted series
- repeat the last 2 steps until all observations are processed
- finally, call a write subroutine that writes out the entire series to a file in a format that DART can read in

6.168.2 Data sources

Matic Savli at University of Ljubljana has programs which read the expected instrument formats, do the proper conversions, and write out ascii lines, one per wind observation.

6.168.3 Programs

The `dwl_to_obs.f90` file is the source for the main converter program. There is a sample data file in the "data" directory. The converter reads each text line into a character buffer and then reads from that buffer to parse up the data items.

To compile and test, go into the work subdirectory and run the `quickbuild.csh` script to build the converter and a couple of general purpose utilities. `advance_time` helps with calendar and time computations, and the `obs_sequence_tool` manipulates DART observation files once they have been created.

The observation types are defined in `DART/obs_def/obs_def_dwl_mod.f90`. That filename must be added to the `input.nml` namelist file, to the &preprocess_nml namelist, the 'input_files' variable before compiling any program that uses these observation types. Multiple files can be listed. Then run quickbuild.csh again. It remakes the table of supported observation types before trying to recompile the source code.

An example script for converting batches of files is in the `shell_scripts` directory. It will need customization before being used.

6.169 PROGRAM `MIDAS_to_obs`

6.169.1 Overview

MIDAS netCDF file to DART observation converter

Alex Chartier (University of Bath, UK) is the point-of-contact for this effort.

> "MIDAS runs in Matlab. The raw observations come from GPS receivers as RINEX files, but we can't use them directly just yet . . . Currently, the 'slant' (satellite-to-receiver path) observations are inverted by MIDAS to make vertical, column-integrated 'observations' of plasma density."

6.169.2 Data sources

Contact Alex for MIDAS observations.

Alex writes out netCDF files that may be converted to DART observation sequence files. The netCDF files have a pretty simple format.

```
netcdf Test {
dimensions:
        latitude = 5 ;
        longitude = 6 ;
        height = 30 ;
        time = UNLIMITED ; // (1 currently)
variables:
        double latitude(latitude) ;
                latitude:units = "degrees_north" ;
                latitude:long_name = "latitude" ;
```

(continues on next page)

(continued from previous page)

```
                latitude:standard_name = "latitude" ;
        double longitude(longitude) ;
                longitude:units = "degrees_east" ;
                longitude:long_name = "longitude" ;
                longitude:standard_name = "longitude" ;
        double height(height) ;
                height:units = "metres" ;
                height:long_name = "height" ;
                height:standard_name = "height" ;
        double time(time) ;
                time:units = "Days since 1601-01-01" ;
                time:long_name = "Time (UT)" ;
                time:standard_name = "Time" ;
        double Ne(height, latitude, longitude) ;
                Ne:grid_mapping = "standard" ;
                Ne:units = "1E11 e/m^3" ;
                Ne:long_name = "electron density" ;
                Ne:coordinates = "latitude longitude" ;
        double TEC(time, latitude, longitude) ;
                TEC:grid_mapping = "standard" ;
                TEC:units = "1E16 e/m^2" ;
                TEC:long_name = "total electron content" ;
                TEC:coordinates = "latitude longitude" ;
        double Variance(time, latitude, longitude) ;
                Variance:grid_mapping = "standard" ;
                Variance:units = "1E16 e/m^2" ;
                Variance:long_name = "Variance of total electron content" ;
                Variance:coordinates = "latitude longitude" ;
                Variance:standard_name = "TEC variance" ;
// global attributes:
                :Conventions = "CF-1.5" ;
}
```

6.169.3 Programs

The MIDAS_to_obs.f90 file is the source code for the main converter program.

To compile and test, go into the MIDAS/work subdirectory and run the quickbuild.csh script to build the converter and a couple of general purpose utilities. The *program obs_sequence_tool* manipulates (i.e. combines, subsets) DART observation files once they have been created. The default observations supported are those defined in observations/forward_operators/obs_def_upper_atm_mod.f90. If you need additional observation types, you will have to add the appropriate obs_def_XXX_mod.f90 file to the input.nml &preprocess_nml:input_files variable and run quickbuild.csh again. It rebuilds the table of supported observation types before compiling the source code.

6.169.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&MIDAS_to_obs_nml
   input_file     = 'infile.nc'
   obs_out_file   = 'obs_seq.out',
   verbose        = .false.
   /
```

Item	Type	Description
input_file	character(len=256)	Name of the input netCDF MIDAS file to read.
obs_out_file	character(len=256)	Name of the output observation sequence file that is created.
verbose	logical	Controls how much informational output is printed during a conversion. `.true.` the most amount of output. `.false.` the least amount of output.

Example

```
&MIDAS_to_obs_nml
   input_file     = '../data/Test.nc',
   obs_out_file   = 'obs_seq.out',
   verbose        = .TRUE.,
```

6.169.5 References

6.170 PROGRAM sst_to_obs, oi_sst_to_obs

6.170.1 Overview

There are two gridded SST observation converters in this directory, one for data from PODAAC, and one from NOAA/NCDC. `sst_to_obs` converts data from PODAAC and has been used by Romain Escudier for regional studies with ROMS. `oi_sst_to_obs` converts data from NOAA/NCDC and has been used by Fred Castruccio for global studies with POP.

sst_to_obs – GHRSST to DART observation sequence converter

These routines are designed to convert the GHRSST Level 4 AVHRR_OI Global Blended Sea Surface Temperature Analysis (GDS version 2) from NCEI data distributed by the Physical Oceanography Distributed Active Archive Center. Please remember to cite the data in your publications, specific instructions from PODAAC are available here. This is an example:

> National Centers for Environmental Information. 2016. GHRSST Level 4 AVHRR_OI Global Blended Sea Surface Temperature Analysis (GDS version 2) from NCEI. Ver. 2.0. PO.DAAC, CA, USA. Dataset accessed [YYYY-MM-DD] at http://dx.doi.org/10.5067/GHAAO-4BC02.

Many thanks to Romain Escudier (then at Rutgers) who did the bulk of the work and graciously contributed his efforts to the DART project. Romain gave us scripts and source code to download the data from the PODAAC site, subset the global files to a region of interest, and convert that subsetted file to a DART observation sequence file. Those scripts and programs have been only lightly modified to work with the Manhattan version of DART and contain a bit more documentation.

The workflow is usually:

1. compile the converters by running `work/quickbuild.csh` in the usual way.
2. customize the `shell_scripts/parameters_SST` resource file to specify variables used by the rest of the scripting.
3. run `shell_scripts/get_sst_ftp.sh` to download the data from PODAAC.
4. provide a mask for the desired study area.
5. run `shell_scripts/Prepare_SST.sh` to subset the PODAAC data and create the DART observation sequence files. Be aware that the `Prepare_SST.sh` modifies the `shell_scripts/input.nml.template` file and generates its own `input.nml`. `work/input.nml` is not used.
6. combine all output files for the region and timeframe of interest into one file using the *program obs_sequence_tool*

Example

It is worth describing a small example. If you configure `get_sst_ftp.sh` to download the last two days of 2010 and then specify the mask to subset for the NorthWestAtlantic (NWA) and run `Prepare_SST.sh` your directory structure should look like the following:

```
0[1234] cheyenne6:/<6>obs_converters/SST
.
|-- ObsData
|   `-- SST
|       |-- ncfile
|       |   `-- 2010
|       |       |-- 20101230120000-NCEI-L4_GHRSST-SSTblend-AVHRR_OI-GLOB-v02.0-fv02.0.
↪nc
|       |       `-- 20101231120000-NCEI-L4_GHRSST-SSTblend-AVHRR_OI-GLOB-v02.0-fv02.0.
↪nc
|       `-- nwaSST
|           `-- 2010
|               |-- 20101230120000-NCEI-L4_GHRSST-SSTblend-AVHRR_OI-GLOB-v02.0-fv02.0_
↪NWA.nc
|               `-- 20101231120000-NCEI-L4_GHRSST-SSTblend-AVHRR_OI-GLOB-v02.0-fv02.0_
↪NWA.nc
|-- oi_sst_to_obs.f90
|-- oi_sst_to_obs.nml
|-- sst_to_obs.f90
|-- sst_to_obs.nml
|-- shell_scripts
|   |-- Prepare_SST.sh
|   |-- functions.sh
|   |-- get_sst_ftp.sh
|   |-- input.nml
|   |-- input.nml.template
|   |-- my_log.txt
|   |-- parameters_SST
```

(continues on next page)

(continued from previous page)

```
|   `-- prepare_SST_file_NWA.sh
|-- masks
|   |-- Mask_NWA-NCDC-L4LRblend-GLOB-v01-fv02_0-AVHRR_OI.nc
|   `-- Mask_NWA120000-NCEI-L4_GHRSST-SSTblend-AVHRR_OI-GLOB-v02.0-fv02.0.nc
`-- work
    |-- Makefile
    |-- advance_time
    |-- input.nml
    |-- mkmf_advance_time
    |-- mkmf_obs_sequence_tool
    |-- mkmf_oi_sst_to_obs
    |-- mkmf_preprocess
    |-- mkmf_sst_to_obs
    |-- obs_sequence_tool
    |-- oi_sst_to_obs
    |-- path_names_advance_time
    |-- path_names_obs_sequence_tool
    |-- path_names_oi_sst_to_obs
    |-- path_names_preprocess
    |-- path_names_sst_to_obs
    |-- preprocess
    |-- quickbuild.csh
    `-- sst_to_obs
```

The location of the DART observation sequence files is specified by `parameter_SST:DIR_OUT_DART`. That directory should contain the following two files:

```
0[1236] cheyenne6:/<6>v2/Err30 > ls -l
'total 7104
-rw-r--r-- 1 thoar p86850054 3626065 Jan 10 11:08 obs_seq.sst.20101230
-rw-r--r-- 1 thoar p86850054 3626065 Jan 10 11:08 obs_seq.sst.20101231
```

6.170.2 oi_sst_to_obs – noaa/ncdc to DART observation sequence converter

`oi_sst_to_obs` is designed to convert the NOAA High-resolution Blended Analysis: Daily Values using AVHRR only data. The global metadata of a typical file is shown here:

```
:Conventions = "CF-1.5" ;
:title = "NOAA High-resolution Blended Analysis: Daily Values using AVHRR only" ;
:institution = "NOAA/NCDC" ;
:source = "NOAA/NCDC  ftp://eclipse.ncdc.noaa.gov/pub/OI-daily-v2/" ;
:comment = "Reynolds, et al., 2007:
     Daily High-Resolution-Blended Analyses for Sea Surface Temperature.
     J. Climate, 20, 5473-5496.
     Climatology is based on 1971-2000 OI.v2 SST,
     Satellite data: Navy NOAA17 NOAA18 AVHRR, Ice data: NCEP ice." ;
:history = "Thu Aug 24 13:46:51 2017: ncatted -O -a References,global,d,, sst.day.
↪mean.2004.v2.nc\n",
        "Version 1.0" ;
:references = "https://www.esrl.noaa.gov/psd/data/gridded/data.noaa.oisst.v2.highres.
↪html" ;
:dataset_title = "NOAA Daily Optimum Interpolation Sea Surface Temperature" ;
```

The workflow is usually:

1. compile the converters by running `work/quickbuild.csh` in the usual way.

2. download the desired data.
3. customize the `work/input.nml` file.
4. run `work/oi_sst_to_obs` to create a single DART observation sequence file.
5. combine all output files for the region and timeframe of interest into one file using the *program obs_sequence_tool*

6.170.3 sst_to_obs namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&sst_to_obs_nml
   sst_netcdf_file     = '1234567.nc'
   sst_netcdf_filelist = 'sst_to_obs_filelist'
   sst_out_file        = 'obs_seq.sst'
   subsample_intv      = 1
   sst_rep_error       = 0.3
   debug               = .false.
   /
```

Contents	Type	Description
sst_netcdf_file	character(len=256)	Name of the (usually subsetted) netcdf data file. This may be a relative or absolute filename. If you run the scripts 'as is', this will be something like: `../ObsData/SST/nwaSST/2010/20101231120000-NCEI-L4_GHRSST-SSTblen d-AVHRR_OI-GLOB-v02.0-fv02.0_NWA.nc`
sst_netcdf_filelist	character(len=256)	Name of the file that contains a list of (usually subsetted) data files, one per line. **You may not specify both sst_netcdf_file AND sst_netcdf_filelist.** One of them must be empty.
sst_out_file	character(len=256)	Name of the output observation sequence file.
subsample_intv	integer	It is possible to 'thin' the observations. `subsample_intv` allows one to take every Nth observation.
sst_rep_error	real	In DART the observation error variance can be thought of as having two components, an instrument error and a representativeness error. In `sst_to_obs` the instrument error is specified in the netCDF file by the variable `analysis_error`. The representativeness error is specified by `sst_rep_error`, which is specified as a standard deviation. These two values are added together and squared and used as the observation error variance. **Note:** This algorithm maintains backwards compatibility, but is technically not the right way to combine these two quantities. If they both specified variance, adding them together and then taking the square root would correctly specify a standard deviation. Variances add, standard deviations do not. Since the true observation error variance (in general) is not known, we are content to live with an algorithm that produces useful observation error variances. If your research comes to a more definitive conclusion, please let us know.
debug	logical	Print extra information during the `sst_to_obs` execution.

6.170.4 oi_sst_to_obs namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&oi_sst_to_obs_nml
   input_file       = '1234567.nc'
   output_file_base = 'obs_seq.sst'
   subsample_intv   = 1
   sst_error_std    = 0.3
   debug            = .false.
   /
```

Contents	Type	Description
input_file	character(len=256)	Name of the input netcdf data file. This may be a relative or absolute filename. If you run the scripts 'as is', this will be something like: `../ObsData/SST/nwaSST/2010/ 20101231120000-NCEI-L4_GHRSST-SSTblen d-AVHRR_OI-GLOB-v02.0-fv02.0_NWA.nc`
output_file_base	character(len=256)	Partial filename for the output file. The date and time are appended to `output_file_base` to construct a unique filename reflecting the time of the observations in the file.
subsample_intv	integer	It is possible to 'thin' the observations. `subsample_intv` allows one to take every Nth observation.
sst_error_std	real	This is the total observation error standard deviation.
debug	logical	Print extra information during the `oi_sst_to_obs` execution.

6.170.5 Decisions you might need to make

See the general discussion in the obs_converters/README.md page about what options are available for the things you need to specify. These include setting a time, specifying an expected error, setting a location, and an observation type.

6.171 PROGRAM `MOD15A2_to_obs`

6.171.1 MODIS land product subsets (collection 5) to DART observation sequence converter

Overview

This routine is designed to convert the MODIS Land Product Subsets data of Leaf Area Index (**LAI**) and Fraction of Photosynthetically Active Radiation (**FPAR**) 8 day composite [MOD15A2] to a DART observation sequence file. According to the MODIS LAI/FPAR Product User's Guide:

> Leaf area index (LAI; dimensionless) is defined as the one-sided green leaf area per unit ground area in broadleaf canopies and as one-half the total needle surface area per unit ground area in coniferous canopies. Fraction of Photosynthetically Active Radiation absorbed by vegetation (FPAR; dimensionless)

> is defined as the fraction of incident photosynthetically active radiation (400-700 nm) absorbed by the green elements of a vegetation canopy.

Specifically, the composites are comma-separated-values (.csv format) ASCII files where each line is a record. The input `.csv` files are directly from the Oak Ridge National Laboratory DAAC. There are two streams to download the data formats we support, they differ only in the very first line of the file. One of the formats has a header record, the other does not. Other than that, the file formats are identical. The format with the header record is fully described in https://lpdaac.usgs.gov/dataset_discovery/modis. Please remember to cite the data in your publications, specific instructions from LP DAAC are available here. This is an example:

> Data Citation: Oak Ridge National Laboratory Distributed Active Archive Center (ORNL DAAC). 2012. MODIS subsetted land products, Collection 5. Available on-line [http://daac.ornl.gov/MODIS/modis.html] from ORNL DAAC, Oak Ridge, Tennessee, U.S.A. Accessed *Month dd, yyyy*.

For more information on *downloading* the data, see DATA SOURCES below. The MODIS Land Product Subsets page indicates that the Collection 5 MODIS Subsets are available three ways:

1. Field Site and Flux tower. Since the files are preprocessed, the download is immediate. The current state of the converter supports this format.
2. Global Tool. This requires exact knowledge of the location(s) of interest. Because some processing to fulfill the request is needed, a job is scheduled on the DAAC server and an email notification is sent with instuctions on how to retrieve the file(s) of interest. The converter **does not** currently support this format, but will soon. Worst case scenario is that you make your own header file and add your 'site' to the metadata file described below.
3. Web Service. I have not used the Web Service.

The DART workflow is usually:

1. download the MOD15A2 data for the sites and years in question (see DATA SOURCES below)
2. build the DART executables with support for `MODIS_LEAF_AREA_INDEX` and `MODIS_FPAR` observations. This is done by running `preprocess` with `obs_def_land_mod.f90` in the list of `input_files` for `preprocess_nml` and then building `MOD15A2_to_obs` in the usual DART way.
3. provide basic information via the `input.nml`:`MOD15A2_to_obs_nml` namelist
4. convert each MODIS data file individually using `MOD15A2_to_obs`
5. combine all output files for the region and timeframe of interest into one file using *program obs_sequence_tool*

For some models (CLM, for example), it is required to reorganize the observation sequence files into a series of files that contains ONLY the observations for each assimilation. This can be achieved with the makedaily.sh script.

6.171.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&MOD15A2_to_obs_nml
   text_input_file = 'MOD15A2.fn_usbouldr.txt',
   metadata_file   = 'MOD15A2_site_metadata.txt',
   obs_out_file    = 'obs_seq.out',
   maxgoodqc       = 10,
   verbose         = .false.
   /
```

Contents	Type	Description
text_input_file	character(len=256)	Name of the MODIS file of comma-separated values. This may be a relative or absolute filename.
metadata_file	character(len=256)	Name of the file that contains the location information for the specific sites. This may be a relative or absolute filename. If this file does not exist, it is **presumed** that the location information is part of the 'site' column. If this is not true, the program will fail. For more information see the section Presumed Format
obs_out_file	character(len=128)	Name of the output observation sequence file.
maxgoodqc	real	maximum value of any observation quality control flag to pass through to the output observation sequence. Keep in mind that `filter` has the ability to discriminate on the value, so there is really little to be gained by rejecting them during the conversion. The QC value is passed through in its native value, i.e. it is not converted to play nicely with observations that have values 0,1,2,3,4,5 etc.
verbose	logical	Print extra information during the `MOD15A2_to_obs` execution.

6.171.3 Data sources

Field site and flux tower

The download site for the 'Field Site and Flux tower' data is
http://daac.ornl.gov/cgi-bin/MODIS/GR_col5_1/mod_viz.html. Since the files are preprocessed, the download is immediate. This method results in files **with** the header record, and requires a small amount of additional work:

- Download the metadata file containing the locations for the Field Sites ftp://daac.ornl.gov/data/modis_ascii_subsets/5_MODIS_SUBSETS_C5_&_FLUXNET.csv
- I usually convert this to UNIX format with the UNIX utility `dos2unix` and rename it to `MOD15A2_site_metadata.txt`

The data files have names like `MOD15A2.fn_uswiirpi.txt` or `MOD15A2.fn_dehambur.txt` and have very long lines. The first line (i.e. record) of the file is a comma-separated list explaining the file format for all the remaining lines/records.
These files contain records with 49 pixel values where each pixel represents the values for a 1km by 1km voxel. The center pixel is the only value converted to a DART observation value.

```
MODIS_LAI % head -1 MOD15A2.fn_dehambur.txt
HDFname,Product,Date,Site,ProcessDate,Band,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,
↪18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,
↪46,47,48,49
```

The format of the `Site` in these files is the predominant difference between the files from the download methods. The `Site` fields in these files have specified site names that must have a case-sensitive match to a site in the metadata file specified by `input.nml`:`metadata_file` .

Global tool

This format is not supported yet.
The download site for the 'Global Tool' data is http://daac.ornl.gov/cgi-bin/MODIS/GLBVIZ_1_Glb/modis_subset_order_global_col5.pl. Because some processing to fulfill the request is needed, a job is scheduled on the DAAC server and an email notification is sent with instuctions on how to retrieve the file(s) of interest. **This method requires exact knowledge of the location(s) of interest.** `MOD15A2_to_obs` presumes prior knowledge of the file format and that the latitude and longitude are coded in the site name (which is the default behavior). **Do not change the format of the file.** Please follow the download instructions below - **exactly.** These instructions were accurate as of 11 April 2014.

1. go to the DAAC download site for MODIS global data.
2. Select either
    1. "Country" (it helps to FIRST clear out the values from the "lat/lon" boxes)
    2. or a specific latitude and longitude. Be precise. This will specify the center pixel location.
3. click "Continue"
4. Select the "[MOD15A2] Leaf Area Index (LAI) and Fraction of Photsyntetically Active Radiation (FPAR) 8 Day Composite" from the pull-down menu.
5. **Important:** Specify 3 **and only 3** kilometers to encompass the center location. This results in the 7 km by 7 km resolution required by `MOD15A2_to_obs`.
6. click "Continue"
7. select the Starting Date and Ending Date from the list. You can convert the entire dataset into one long DART observation sequence file and then subset it later if need be.
8. **Important:** Make sure you check the button "Generate GeoTIFF and Reproject to Geographic Lat/long"
9. Supply your REAL email address
10. click "Continue"
11. Review the confirmation page. Make sure the requested resolution and area is correct. You should see something like "The Requested Data Area is Approximately 7 Kilometers Wide and 7 Kilometers High"
12. click "Continue"
13. At some point later (perhaps even days), you will get an email with the subject "ORNL DAAC MODIS MOD15A2 order", follow the instructions to complete the download.

The resulting ASCII files will have the same format as described below. The 'site name' column for these files is of the form: `Lat47.61666667Lon12.58333333Samp7Line7` which provides the location information otherwise provided by the `MOD15A2_site_metadata.txt` file for the predefined sites.

Web service

I have not used the Web Service.

6.171.4 Format

The data product "Leaf Area Index - Fraction of Photosynthetically Active Radiation 8-Day L4 Global 1km" (**MOD15A2**) is described in https://lpdaac.usgs.gov/products/modis_products_table/mod15a2 (**expand the 'Layers' tab**). The units and the QC values are described there. What I have not been able to determine is how to interpret the 'Date' … if it is 2000049 … It is day 49 of year 2000. Is that the start of the 8 day composite, the middle, the end? If you know the answer, please let me know.

Taken (almost) directly from https://lpdaac.usgs.gov/tools/lp_daac_web_services and modified only slightly with examples more appropriate for the LAI/FPAR product.

The MODIS MOD15A2 products in question are ASCII files of comma-separated values. If the file contains a header record/line, all columns are interpreted based on this header column. If the file does not contain a header, the following format is REQUIRED.

- ASCII values are comma delimited
- Row 1 is the header row (which may not exist for products generated by the Global Tool)
- Data values start in row 2 if the header row is present.
- Rows of QC data are interleaved with measurement data as indicated in Column 6.
- Note that values may contain embedded periods, dashes, and underscores (".,-, _").

Column Number	Column Description	Example Values
1	Unique row identifier	MOD15A2.A2000049.fn_ruyakuts.005.2006268205917.Fpar_1km MOD15A2.A2000049.fn_ruyakuts.005.2006268205917.Lai_1km
2	MODIS Land Product Code	MOD15A2
3	MODIS Acquisition Date A(YYYYDDD)	A2000049 (?this is an 8 day average) What does 49 indicate? start? middle? end?
4	SiteID Each site is assigned a unique ID. Click Here to get Site name information from SiteID	fn_ustnwalk, Lat47.61666667Lon12.58333333Samp7Line7
5	MODIS Processing Date (YYYYDDDHHMMSS)	2006269073558
6	Product Scientific Data Set (Band): Indicates type of values to follow. Specific values vary by Product. Data quality information are interleaved.	MOD15A2: FparExtra_QC, FparLai_QC, FparStdDev_1km, Fpar_1km, LaiStdDev_1km, Lai_1km
7 to N	Data values of type as specified. Number of data columns as given in Column 4. Definition of QC component values vary by Scientific Data Set.	QC: 00100001,01100001,01100001, … Measurement: 2,2,1,1,1,1,1,0,0,0,1,1,0,0, to N

QC flags are binary-coded ascii strings e.g., 10011101 bits 5,6,7 (the last three) are decoded as follows:

- 000 … Main(RT) method used, best result possible (no saturation)
- 001 … Main(RT) method used with saturation, Good, very usable
- 010 … Main(RT) method failed due to bad geometry, empirical algorithm used

- 011 … Main(RT) method failed due to other problems
- 100 … pixel not produced at all

Consequently, the last three digits are used by DART's data processing logic.

6.171.5 Programs

The `MOD15A2_to_obs.f90` file is the source for the main converter program. Look at the source code where it reads the example data file. You will almost certainly need to change the "read" statement to match your data format. The example code reads each text line into a character buffer and then reads from that buffer to parse up the data items.
FIXME Explain the 10% for the obs error for FPAR and question the LAIStddev …

To compile and test, go into the work subdirectory and run the `quickbuild.csh` script to build the converter and a couple of general purpose utilities. `advance_time` helps with calendar and time computations, and the `obs_sequence_tool` manipulates DART observation files once they have been created.

To change the observation types, look in the `DART/obs_def` directory. If you can find an obs_def_XXX_mod.f90 file with an appropriate set of observation types, change the 'use' lines in the converter source to include those types. Then add that filename in the `input.nml` namelist file to the &preprocess_nml namelist, the 'input_files' variable. Multiple files can be listed. Then run quickbuild.csh again. It remakes the table of supported observation types before trying to recompile the source code.

An example script for converting batches of files is in the `shell_scripts` directory. A tiny example data file is in the `data` directory. These are *NOT* intended to be turnkey scripts; they will certainly need to be customized for your use. There are comments at the top of the script saying what options they include, and should be commented enough to indicate where changes will be likely to need to be made.

6.171.6 Decisions you might need to make

See the general discussion in the obs_converters/README.md page about what options are available for the things you need to specify. These include setting a time, specifying an expected error, setting a location, and an observation type.

6.172 DART observations and MODIS products.

There are many MODIS products, in many formats. This document will list all of the data products and formats that have DART programs to convert them to observation sequence files.

6.172.1 Programs

PROGRAM MOD15A2_to_obs	Converts MODIS Land Product Subsets Leaf Area Index (**LAI**) and Fraction of Photosynthetically Active Radiation (**FPAR**) 8 day composite [MOD15A2]

6.172.2 Plans

1. Support MOD15A2 'Global Tool' records.
2. The work that remains is to get the IGBP landcover code for the site and incorporate that into the observation metadata. I *almost* have everything I need. Once that happens, the forward observation operator can be made to be much more accurate by only using model landunits that have the right landcover class.
3. Support more products. Put in a request to help me prioritize.

6.173 PROGRAM `prepbufr`

6.173.1 Overview

Translating NCEP PREPBUFR files into DART obs_seq.out files (input file to filter) is a 2 stage process. The first stage uses NCEP software to translate the PREPBUFR file into an intermediate text file. This is described in this document. The second step is to translate the intermediate files into obs_seq.out files, which is done by create_real_obs, as described in *PROGRAM create_real_obs* .

6.173.2 Instructions

The prep_bufr package is free-standing and has not been completely assimilated into the DART architecture. It also requires adaptation of the sources codes and scripts to the computing environment where it will be run. It is not so robust that it can be controlled just with input parameters. It may not have the same levels of error detection and warning that the rest of DART has, so the user should very careful about checking the end product for correctness.

Overview of what needs to be built and run

More detailed instructions follow, but this section describes a quick overview of what programs you will be building and running.

Building

Running the install.sh script will build the library and main executable. You will probably have to edit this script to set which fortran compiler is available on your system.

If you have raw unblocked PREPBUFR files you will need to convert them to blocked format (what prepbufr expects as input). The blk/ublk section of the build script compiles the `cword.x` converter program.

If you are running on an Intel (little-endian) based machine you will need the `grabbufr` byte swapping program that is also built by this script.

One-shot execution

If you are converting a single obs file, or are walking through the process by hand for the first time, you can follow the more detailed build instructions below, and then run the prep_bufr.x program by hand. This involves the following steps:

- building the executables.
- running the blocker if needed (generally not if you have downloaded the blocked format PREPBUFR files).
- running the binary format converter if you are on an Intel (little-endian) machine.
- linking the input file to a fixed input filename
- running prepbufr.x to convert the file
- copying the fixed output filename to the desired output filename

Production mode

If you have multiple days (or months) of observations that you are intending to convert, there is a script in the work subdirectory which is set up to run the converter on a sequence of raw data files, and concatenate the output files together into one output file per day. Edit the work/prepbufr.csh script and set the necessary values in the 'USER SET PARAMETERS' section near the top. This script can either be run from the command line, or it can be submitted to a batch queue for a long series of conversion runs.

Installation of the ncep prepbufr decoding program

This package is currently organized into files under the DART/observations/NCEP/prep_bufr directory:

```
src            Source code of the NCEP PREPBUFR decoder
lib            NCEP BUFR library source
install.sh     A script to install the NCEP PREPBUFR decoder and the NCEP BUFR library.
exe            Executables of the decoder and converter.
data           Where the NCEP PREPBUFR files (prepqm****) could be loaded into
               from the NCAR Mass Store (the script assumes this is the default␣
↪location).
work           Where we run the script to do the decoding.
convert_bufr   Source code (grabbufr) to convert the binary big-endian PREPBUFR files␣
↪to
               little-endian files, and a script to compile the program.
blk_ublk       Source code (cwordsh) to convert between blocked and unblocked format.
docs           Some background information about NCEP PREPBUFR observations.
```

The decoding program: src/prepbufr.f

The program prepbufr.f is used to decode the NCEP reanalysis PREPBUFR data into intermediate text files. This program was originally developed by NCEP. It has been modified to output surface pressure, dry temperature, specific humidity, and wind components (U/V) of conventional radiosonde, aircraft reports, and satellite cloud motion derived wind. There are additional observation types on the PREPBUFR files, but using them they would require significant modifications of prepbufr and require detailed knowledge of the NCEP PREPBUFR files. The NCEP quality control indexes for these observations based on NCEP forecasts are also output and used in DART observation sequence files. The NCEP PREPBUFR decoding program is written in Fortran 77 and has been successfully compiled on Linux computers using pgi90, SGI® computers with f77, IBM® SP® systems with xlf, and Intel® based Mac® with gfortran.

If your operating system uses modules you may need to remove the default compiler and add the one desired for this package. For example

- which pgf90 (to see if pgf90 is available.)
- module rm intel64 netcdf64 mpich64
- module add pgi32

To compile the BUFR libraries and the decoding program, set the CPLAT variable in the install.sh script to match the compilers available on your system. CPLAT = linux is the default. Execute the install.sh script to complete the compilations for the main decoding program, the NCEP BUFR library, and the conversion utilities.

The executables (i.e., prepbufr.x, prepbufr_03Z.x) are placed in the ../exe directory.

Platforms tested:

- Linux clusters with Intel, PGI, Pathscale, GNU Fortran,
- Mac OS X with Intel, GNU Fortran,
- SGI Altix with Intel
- Cray with Intel, Cray Fortran.

The byte-swapping program convert_bufr/grabbufr.f

For platforms with little-endian binary file format (e.g. Intel, AMD®, and non-MIPS SGI processors) the program grabbufr.f is used to convert the big-endian format NCEP PREPBUFR data into little-endian format. The grabbufr.f code is written in Fortran 90, and has been compiled can be compiled with the pgf90 compiler on a Linux system, with gfortran on an Intel based Mac, and the ifort compiler on other Linux machines. More detailed instructions for building it can be found in convert_bufr/README, but the base install script should build this by default. In case of problems, cd into the convert_bufr subdirectory, edit convert_bufr.csh to set your compiler, and run it to compile the converter code (grabbufr).

This program reads the whole PREPBUFR file into memory, and needs to know the size of the file (in bytes). Unfortunately, the system call STAT() returns this size as one number in an array, and the index into that array differs depending on the system and sometimes the word size (32 vs 64) of the compiler. To test that the program is using the right offset into this array, you can compile and run the stat_test.f program. It takes a single filename argument and prints out information about that file. One of the numbers will be the file size in bytes. Compare this to the size you see with the 'ls -l' command for that same file. If the numbers do not agree, find the right index and edit the grabbufr.f source file. Look for the INDEXVAL line near the first section of executable code.

If grabbufr.f does not compile because the getarg() or iargc() subroutines are not found or not available, then either use the arg_test.f program to debug how to get command line arguments into a fortran program on your system, or simply go into the grabbufr.f source and comment out the section which tries to parse command line arguments and comment in the hardcoded input and output filenames. Now to run this program you must either rename the data files to these predetermined filenames, or you can use links to temporarily give the files the names needed.

The blocking program blk_ublk/cword.x

The prepbufr.x program expects to read a blocked input file, which is generally what is available for download. However, if you have an unblocked file that you need to convert, there is a conversion program. The install.sh script will try to build this by default, but in case of problems you can build it separately. Change directories into the blk_ublk subdirectory and read the README_cwordsh file for more help. The cwordsh shell-script wrapper shows how to run the executable cwordsh.x executable.

Note that if you can get the blocked file formats to begin with, this program is not needed.

Getting the ncep reanalysis prepbufr format data from ncar hpss

The NCEP PREPBUFR files (prepqmYYMMDDHH) can be found within the NCEP reanalysis dataset, ds090.0, on NCAR Mass Store System (HPSS).

To find the files:

- go to the NCAR/NCEP reanalysis archive.
- Click on the “Inventories” tab.
- Select the year you are interested in.
- Search for files with the string “prepqm” in the name.
- Depending on the year the format of the filenames change, but they should contain the year, usually as 2 digits, the month, and then either the start/stop day for weekly files, or the letters A and B for semi-monthly files.

Depending on the year you select, the prepqm files can be weekly, monthly, or semi-monthly. Each tar file has a unique dataset number of the form “A#####”. For example, for January of 2003, the 4 HPSS TAR files are: A21899, A21900, A21901, A21902. After September 2003, these files include AIRCRAFT data (airplane readings taken at cruising elevation) but not ACARS data (airplane readings taken during takeoff and landing). There are different datasets which include ACARS data but their use is restricted and you must contact the RDA group to get access.

If you are running on a machine with direct access to the NCAR HPSS, then change directories into the
prep_bufr/data subdirectory and run:
> *hsi get /DSS/A##### rawfile*
where ##### is the data set number you want.

These files may be readable tar files, or they may require running the `cosconvert` program first. See if the `tar` command can read them:
> *tar -tvf rawfile*
If you get a good table of contents then simply rename the file and untar it:
> *mv rawfile data.tar*
> *tar -xvf data.tar*
However, if you get an error from the tar command you will need to run the `cosconvert` program to convert the file into a readable tar file. On the NCAR machine *yellowstone*, run:
> */glade/u/home/rdadata/bin/cosconvert -b rawfile data.tar*
On other platforms, download the appropriate version from: http://rda.ucar.edu/libraries/io/cos_blocking/utils/ .
Build and run the converter and then you should have a tar file you can unpack.

The output of tar should yield individual 6-hourly NCEP PREPBUFR data files for the observations in the +/- 3-hour time windows of 00Z, 06Z, 12Z, and 18Z of each day. Note that DART obs_seq files are organized such that a 24 hour file with 4 observation times would contain observations from 3:01Z to 3:00Z of the next day, centered on 6Z, 12Z, 18Z and "24Z". In addition, there are some observations at 3:00Z on the PREPBUFR file labelled with 06Z. Then, in order to make a full day intermediate file incorporating all the required obs from the "next" day, you'll need the PREPBUFR files through 6Z of the day after the last day of interest. For example, to generate the observation sequence for Jan 1, 2003, the decoded NCEP PREPBUFR text files for Jan 1 and 2, 2003 are needed, and hence the PREPBUFR files

- prepqm03010106
- prepqm03010112
- prepqm03010118
- prepqm03010200
- prepqm03010206

are needed.

Running the ncep prepbufr decoding program

In prep_bufr/work/prepbufr.csh set the appropriate values of the year, month, first day, and last day of the period you desire, and the variable "convert" to control conversion from big- to little-endian. Confirm that the raw PREPBUFR files are in ../data, or that prepbufr.csh has been changed to find them. Execute prepbufr.csh in the work directory. It has code for running in the LSF batch environment, but not PBS.

Currently, this script generates decoded PREPBUFR text data each 24 hours which contains the observations within the time window of -3:01 hours to +3:00Z within each six-hour synoptic time. These daily output text files are named as temp_obs.yyyymmdd. These text PREPBUFR data files can then be read by DART/observations/NCEP/ascii_to_obs/work/*PROGRAM create_real_obs* to generate the DART daily observation sequence files.

There is an alternate section in the script which creates a decoded PREPBUFR text data file each 6 hours (so they are 1-for-1 with the original PREPBUFR files). Edit the script prepbufr.csh and look for the commented out code which outputs 4 individual files per day. Note that if you chose this option, you will have to make corresponding changes in the create_obs_seq.csh script in step 2.

6.173.3 Other modules used

This is a piece of code that is intended to be 'close' to the original, as such, we have not modified it to use the DART build mechanism. This code does not use any DART modules.

6.173.4 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&prep_bufr_nml
   obs_window        = 1.5,
   obs_window_upa    = 1.5,
   obs_window_air    = 1.5,
   obs_window_sfc    = 0.8,
   obs_window_cw     = 1.5,
   land_temp_error   = 2.5,
   land_wind_error   = 3.5,
```

(continues on next page)

(continued from previous page)

```
   land_moist_error = 0.2,
   otype_use        = missing,
   qctype_use       = missing,
/
```

Item	Type	Description
obs_window	real	Window of time to include observations. If > 0, overrides all the other more specific window sizes. Set to -1.0 to use different time windows for different obs types. The window is +/- this number of hours, so the total window size is twice this value.
obs_window_upa	real	Window of time to include sonde observations (+/- hours) if obs_window is < 0, otherwise ignored.
obs_window_air	real	Window of time to include aircraft observations (+/- hours) if obs_window is < 0, otherwise ignored.
obs_window_sfc	real	Window of time to include surface observations (+/- hours) if obs_window is < 0, otherwise ignored.
obs_window_cw	real	Window of time to include cloud wind observations (+/- hours) if obs_window is < 0, otherwise ignored.
otype_use	real(300)	Report Types to extract from bufr file. If unspecified, all types will be converted.
qctype_use	integer(300)	QC types to include from the bufr file. If unspecified, all QC values will be accepted.
land_temp_error	real	observation error for land surface temperature observations when none is in the input file.
land_wind_error	real	observation error for land surface wind observations when none is in the input file.
land_moist_error	real	observation error for land surface moisture observations when none is in the input file.

6.173.5 Files

- input file(s); NCEP PREPBUFR observation files named using ObsBase with the “yymmddhh” date tag on the end. Input to grabbufr if big- to little-endian is to be done. Input to prepbufr if not.
- intermediate (binary) prepqm.little; output from grabbufr, input to prepbufr.
- intermediate (text) file(s) “temp_obs.yyyymmddhh”; output from prepbufr, input to create_real_obs

6.173.6 References

DART/observations/NCEP/prep_bufr/docs/* (NCEP text files describing the PREPBUFR files)

6.174 PROGRAM create_real_obs

6.174.1 Overview

Translating NCEP BUFR files into DART obs_seq.out files (input file to filter) is a 2 stage process. The first stage uses NCEP software to translate the BUFR file into an "intermediate" text file. This is described in *PROGRAM prepbufr*. The second step is to translate the intermediate files into an `obs_seq.out` files, which is done by `create_real_obs`, as described in this document.

This program provides a number of options to select several observation types (radiosonde, aircraft, and satellite data, etc.) and the DART observation variables (U, V, T, Q, Ps) which are specified in its optional namelist interface `&ncepobs_nml` which may be read from file `input.nml`.

6.174.2 Instructions

- Go to DART/observations/NCEP/ascii_to_obs/work
- Use `quickbuild.csh` to compile all executable programs in the directory. To rebuild just one program:
  - Use `mkmf_create_real_obs` to generate the makefile to compile `create_real_obs.f90`.
  - Type `make` to get the executable.
- Make appropriate changes to the `&ncep_obs_nml` namelist in `input.nml`, as follows.
- run `create_real_obs`.

The selection of any combinations of the specific observation fields (T, Q, U/V, and surface pressure) and types (radiosonde, aircraft reports, or satellite wind, etc.) is made in the namelist `&ncepobs_nml`. All the available combinations of fields X types (i.e. ADPUPA and obs_U) will be written to the obs_seq file. (You will be able to select which of those to use during an assimilation in another namelist (`assimilate_these_obs`, in `&obs_kind_nml`), so be sure to include all the fields and types you might want.) You should change `Obsbase` to the pathname of the decoded PREPBUFR text data files. Be sure that `daily_file` is set to .TRUE. to create a single 24 hour file; .FALSE. converts input files one-for-one with output files. The default action is to tag each observation with the exact time it was taken and is the recommended setting. However, if you want to bin the observations in time, for example to do additional post-processing, the time on all observations in the window can be overwritten and set to the nearest synoptic time (e.g. 0Z, 6Z, 12Z, or 18Z), by setting `obs_time` to false.

Generally you will want to customize the namelist for your own use. For example, here is a sample namelist:

```
&ncepobs_nml
  year = 2007,
  month = 3,
  day = 1,
  tot_days = 31,
  max_num = 700000,
  ObsBase = '../prep_bufr/work/temp_obs.'
  select_obs   = 1,
  ADPUPA = .true.,
  AIRCAR = .false.,
  AIRCFT = .true.,
  SATEMP = .false.,
  SFCSHP = .false.,
  ADPSFC = .false.,
  SATWND = .true.,
  obs_U  = .true.,
  obs_V  = .true.,
```

(continues on next page)

(continued from previous page)

```
  obs_T  = .true.,
  obs_PS = .false.,
  obs_QV = .false.,
  daily_file = .true.
  obs_time = .true.,
/

&obs_sequence_nml
  write_binary_obs_sequence = .false.
/
```

This will produce daily observation sequence files for the period of March 2007, which have the selected observation types and fields; T, U, and V from radiosondes (ADPUPA) and aircraft (AIRCFT). No surface pressure or specific humidity would appear in the obs_seq files, nor observations from ACARS, satellites, and surface stations. The output files look like "obs_seq200703dd", with dd = 1,. . . ,31.

6.174.3 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&ncepobs_nml
   year        = 2003,
   month       = 1,
   day         = 1,
   tot_days    = 31,
   max_num     = 800000,
   select_obs  = 0,
   ObsBase     = 'temp_obs.',
   ADPUPA      = .false.,
   AIRCAR      = .false.,
   AIRCFT      = .false.,
   SATEMP      = .false.,
   SFCSHP      = .false.,
   ADPSFC      = .false.,
   SATWND      = .false.,
   obs_U       = .false.,
   obs_V       = .false.,
   obs_T       = .false.,
   obs_PS      = .false.,
   obs_QV      = .false.,
   daily_file  = .true.,
   obs_time    = .true.,
   lon1        =   0.0,
   lon2        = 360.0,
   lat1        = -90.0,
   lat2        =  90.0
/
```

Item	Type	Description
year, month, day	integer	Beginning year, month, day of the observation period.
tot_days	integer	Total days in the observation period. The converter cannot cross month boundaries.
max_num	integer	Maximum observation number for the current one day files.
select_obs	integer	Controls whether to select a subset of observations from the NCEP BUFR decoded daily ascii files. • 0 = All observations are selected. • 1 = Select observations using the logical parameters below.
daily_file	logical	Controls timespan of observations in each obs_seq file: • true = 24 hour spans (3:01Z to 3:00Z of the next day). Filenames have the form obs_seqYYYYMMDD. • false = 6 hour spans (3:01Z to 9:00Z, 9:01Z to 15:00Z, 15:01Z to 21:00Z, and 21:01Z to 3:00Z of the next day. Filenames have the form obs_seqYYYYMMDDHH, where HH is 06, 12, 18, and 24.
ObsBase	character(len=129)	Path that contains the decoded NCEP BUFR daily observation files. To work with the example scripts this should be 'temp_obs.', or if it includes a pathname then it should end with a '/temp_obs.'
include_specific_humidity, include_relative_humidity, include_dewpoint	logical	Controls which moisture observations are created. The default is to create only specific humidity obs, but any, all, or none can be requested. Set to .TRUE. to output that obs type, .FALSE. skips it.
ADPUPA	logical	Select the NCEP type ADPUPA observations which includes land and ship launched radiosondes and pibals as well as a few profile dropsonde. This involves, at 00Z and 12Z, about 650 - 1000 stations, and at 06Z and 18Z (which are mostly pibals), about 150 - 400 stations.
AIRCFT	logical	Select the NCEP type AIRCFT observations, which includes commercial, some military and reconnaissance reports. They are flight level reports.
AIRCAR	logical	Select the NCEP type AIRCAR observations, which includes data

6.174.4 Modules used

```
types_mod
utilities_mod
obs_utilities_mod
obs_sequence_mod
obs_kind_mod
obs_def_mod
assim_model_mod
model_mod
cov_cutoff_mod
location_mod
random_seq_mod
time_manager_mod
null_mpi_utilities_mod
real_obs_mod
```

6.174.5 Files

- path_names_create_real_obs; the list of modules used in the compilation of create_real_obs.
- temp_obs.yyyymmdd; (input) NCEP BUFR (decoded/intermediate) observation file(s) Each one has 00Z of the next day on it.
- input.nml; the namelist file used by create_real_obs.
- obs_seqYYYYMMDD[HH]; (output) the obs_seq files used by DART.

6.174.6 References

- …/DART/observations/NCEP/prep_bufr/docs/* (NCEP text files describing the BUFR files)

6.175 SSUSI F16 EDR-DSK format to observation sequence converters

6.175.1 Overview

The Special Sensor Ultraviolet Spectrographic Imager SSUSI is designed to remotely sense the ionosphere and thermosphere. The following is repeated from the SSUSI home page:

> *Overview Beginning in 2003, the Defense Meteorological Satellite Program (DMSP) satellites began carrying the SSUSI instrument - a combination of spectrographic imaging and photometric systems designed to remotely sense the ionosphere and thermosphere. The long term focus of the SSUSI program is to provide data concerning the upper atmospheric response to the sun over the changing conditions of the solar cycle. Data collected by SSUSI instrument can help identify structure in the equatorial and polar regions. Mission SSUSI was designed for the DMSP Block 5D-3 satellites. These satellites are placed into nearly polar, sun-synchronous orbits at an altitude of about 850 km. SSUSI is a remote-sensing instrument which measures ultraviolet (UV) emissions in five different wavelength bands from the Earth's upper atmosphere. SSUSI is mounted on a nadir-looking panel of the satellite. The multicolor images*

> *from SSUSI cover the visible Earth disk from horizon to horizon and the anti-sunward limb up to an altitude of approximately 520 km. The UV images and the derived environmental data provide the Air Force Weather Agency (Offutt Air Force Base, Bellevue, NE) with near real-time information that can be utilized in a number of applications, such as maintenance of high frequency (HF) communication links and related systems and assessment of the environmental hazard to astronauts on the Space Station.*

`convert_f16_edr_dsk.f90` will extract the ON2 observations from the F16 "edr-dsk" format files and create DART observation sequence files. There is one additional preprocessing step before the edr-dsk files may be converted.

The ON2_UNCERTAINTY variable in the netcdf files have IEEE NaN values, but none of the required metadata to interpret them correctly. These 2 lines will add the required attributes so that NaNs are replaced with a fill value that can be queried and manipulated. Since the ON2_UNCERTAINTY is a standard deviation, it is sufficient to make the fill value negative. See the section on Known Bugs

```
ncatted -a _FillValue,ON2_UNCERTAINTY,o,f,NaN        input_file.nc
ncatted -a _FillValue,ON2_UNCERTAINTY,m,f,-1.0       input_file.nc
```

6.175.2 Data sources

http://ssusi.jhuapl.edu/data_products

Please read their data usage policy.

6.175.3 Programs

`DART/observations/SSUSI/convert_f16_edr_dsk.f90` will extract ON2 data from the distribution files and create DART observation sequence (obs_seq) files. Build it in the `SSUSI/work` directory by running the `./quickbuild.csh` script located there. In addition to the converters, the `advance_time` and `obs_sequence_tool` utilities will be built.

An example data file is in the `data` directory. An example scripts for adding the required metadata to the ON2_UNCERTAINTY variable in the `shell_scripts` directory. These are *NOT* intended to be turnkey scripts; they will certainly need to be customized for your use. There are comments at the top of the scripts saying what options they include, and should be commented enough to indicate where changes will be likely to need to be made.

6.175.4 Errors

The code for setting observation error variances is using fixed values, and we are not certain if they are correct. Incoming QC values larger than 0 are suspect, but it is not clear if they really signal unusable values or whether there are some codes we should accept.

6.176 Oklahoma Mesonet MDF Data

6.176.1 Overview

Program to convert Oklahoma Mesonet MDF files into DART observation sequence files.

6.176.2 Data sources

The observation files can be obtained from the Oklahoma Mesonet archive using urls of the format: http://www.mesonet.org/index.php/dataMdfMts/dataController/getFile/YYYYMMDDHHMM/mdf/TEXT where YYYYMMDDHHMM is the date and time of the desired set of observations. Files are available every 5 minutes.

If you are located outside of Oklahoma or are going to use this for a non-research purpose see this web page for information about access: http://www.mesonet.org/index.php/site/about/data_access_and_pricing

Static fields are drawn from the station description file provided by the OK Mesonet. Update the local file from: http://www.mesonet.org/index.php/api/siteinfo/from_all_active_with_geo_fields/format/csv

6.176.3 Programs

The programs in the `DART/observations/ok_mesonet/` directory extract data from the distribution files and create DART observation sequence (obs_seq) files. Build them in the `work` directory by running the `./quickbuild.csh` script. In addition to the converters, the `advance_time` and `obs_sequence_tool` utilities will be built.

The converter is a preliminary version which has no namelist inputs. It has hard-coded input and output filenames. It always reads a data file named `okmeso_mdf.in` and creates an output file named `obs_seq.okmeso`. The converter also requires a text file with the location of all the observating stations, called `geoinfo.csv`.

The converter creates observations of the following types:

- LAND_SFC_ALTIMETER
- LAND_SFC_U_WIND_COMPONENT
- LAND_SFC_V_WIND_COMPONENT
- LAND_SFC_TEMPERATURE
- LAND_SFC_SPECIFIC_HUMIDITY
- LAND_SFC_DEWPOINT
- LAND_SFC_RELATIVE_HUMIDITY

Example data files are in the `data` directory. Example scripts for converting batches of these files are in the `shell_scripts` directory. These are *NOT* intended to be turnkey scripts; they will certainly need to be customized for your use. There are comments at the top of the scripts saying what options they include, and should be commented enough to indicate where changes will be likely to need to be made.

The expected usage pattern is that a script will copy, rename, or make a symbolic link from the actual input file (which often contains a timestamp in the name) to the fixed input name before conversion, and move the output file to an appropriate filename before the next invocation of the converter. If an existing observation sequence file of the same output name is found when the converter is run again, it will open that file and append the next set of observations to it.

6.177 PROGRAM snow_to_obs

6.177.1 MODIS snowcover fraction observation converter

Overview

There are several satellite sources for snow observations. Generally the data is distributed in HDF-EOS format. The converter code in this directory DOES NOT READ HDF FILES as input. It expects the files to have been preprocessed to contain text, one line per observation, with northern hemisphere data only.

6.177.2 Data sources

not sure.

6.177.3 Programs

The snow_to_obs.f90 file is the source for the main converter program.

To compile and test, go into the work subdirectory and run the quickbuild.csh script to build the converter and a couple of general purpose utilities. advance_time helps with calendar and time computations, and the obs_sequence_tool manipulates DART observation files once they have been created.

This converter creates observations of the "MODIS_SNOWCOVER_FRAC" type.

There is another program in this directory called snow_to_obs_netcdf.f90 which is a prototype for reading netcdf files that contain some metadata and presumably have been converted from the original HDF. THIS HAS NOT BEEN TESTED but if you have such data, please contact dart@ucar.edu for more assistance. If you write something that reads the HDF-EOS MODIS files directly, please, please contact us! Thanks.

6.177.4 Namelist

This namelist is read from the file input.nml. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&snow_to_obs_nml
  longrid         = 360,
  latgrid         = 90,
  year            = 2000,
  doy             = 1,
  snow_input_file = 'snowdata.input',
  missing_value   = -20.0,
  debug           = .false.
/
```

Item	Type	Description
longrid	integer	The number of divisions in the longitude dimension.
latgrid	integer	The number of divisions in the latitude dimension. This converter assumes the data is for the northern hemisphere only. A namelist item could be added to select northern verses southern hemisphere if needed.
year	integer	The year number of the data.
doy	integer	The day number in the year. Valid range 1 to 365 in a non-leap year, 1 to 366 in a leap year.
snow_input_file	character(len=128)	The name of the input file.
missing_value	real(r8)	The value used to mark missing data.
debug	logical	If set to .true. the converter will print out more information as it does the conversion.

6.178 PROGRAM `text_to_obs`

6.178.1 Text file to DART converter

Overview

If you have observations in spreadsheet or column format, in text, with a single line per observation, then the files this directory are a template for how to convert these observations into a format suitable for DART use.

The workflow is usually:

- read in the needed information about each observation - location, time, data value, observation type - from a data source (usually a file)
- call a series of DART library routines to construct a derived type that contains all the information about a single observation
- call another set of DART library routines to put it into a time-sorted series
- repeat the last 2 steps until all observations are processed
- finally, call a write subroutine that writes out the entire series to a file in a format that DART can read in

It is not recommended that you try to mimic the ascii file format by other means; the format is subject to change and the library routines will continue to be supported even if the physical format changes.

If your input data is in some kind of format like netCDF or HDF, then one of the other converters (e.g. the MADIS ones for netCDF) might be a better starting place for adapting code.

6.178.2 Data sources

This part is up to you. For each observation you will need a location, a data value, a type, a time, and some kind of error estimate. The error estimate can be hardcoded in the converter if they are not available in the input data. See below for more details on selecting an appropriate error value.

6.178.3 Programs

The text_to_obs.f90 file is the source for the main converter program. Look at the source code where it reads the example data file. You will almost certainly need to change the "read" statement to match your data format. The example code reads each text line into a character buffer and then reads from that buffer to parse up the data items.

To compile and test, go into the work subdirectory and run the quickbuild.csh script to build the converter and a couple of general purpose utilities. advance_time helps with calendar and time computations, and the obs_sequence_tool manipulates DART observation files once they have been created.

To change the observation types, look in the DART/obs_def directory. If you can find an obs_def_XXX_mod.f90 file with an appropriate set of observation types, change the 'use' lines in the converter source to include those types. Then add that filename in the input.nml namelist file to the &preprocess_nml namelist, the 'input_files' variable. Multiple files can be listed. Then run quickbuild.csh again. It remakes the table of supported observation types before trying to recompile the source code.

An example script for converting batches of files is in the shell_scripts directory. A tiny example data file is in the data directory. These are *NOT* intended to be turnkey scripts; they will certainly need to be customized for your use. There are comments at the top of the script saying what options they include, and should be commented enough to indicate where changes will be likely to need to be made.

6.178.4 Decisions you might need to make

See the discussion in the obs_converters/README.md page about what options are available for the things you need to specify. These include setting a time, specifying an expected error, setting a location, and an observation type.

6.179 Radar Observations

6.179.1 Overview

Several programs for converting radar observations into DART obs_seq format exist, and will be placed in this directory when they are ready for distribution. Observations generated by these programs have been successfully assimilated with weather models in the DART framework.

This directory currently contains a program for generating synthetic radar observations for a WSR-88D (NEXRAD). It can generate reflectivity and/or doppler radial velocity observations with clear-air or storm sweep patterns, for testing or for OSSEs (Observing System Simulation Experiments).

There are challenges to working with radar data; for more information contact us.

6.179.2 Data sources

6.179.3 Programs

create_obs_radar_sequence generates one or more sets of synthetic radar observations. Change into the work subdirectory and run quickbuild.csh to build this program.

Many DART users working with radar observations are using the WRF Weather and Research Forecast model. See the WRF tests directory for pointers to data to run a radar test case.

In addition to the programs available in the DART distribution, the following external program produces DART observation sequence files:

- Observation Processing And Wind Synthesis (OPAWS): OPAWS can process NCAR Dorade (sweep) and NCAR EOL Foray (netcdf) radar data. It analyzes (grids) data in either two-dimensions (on the conical surface of each sweep) or three-dimensions (Cartesian). Analyses are output in netcdf, Vis5d, and/or DART (Data Assimilation Research Testbed) formats.

6.180 MADIS Data Ingest System

6.180.1 Overview

The MADIS (Meteorological Assimilation Data Ingest System) service provides access to real-time and archived data of a variety of types, with added Quality Control (QC) and integration of data from a variety of sources.

To convert a series of MADIS data files (where different types of observations are distributed in separate files), one high level view of the workflow is:

1. convert each madis file, by platform type, into an obs_seq file. one file in, one file out. no time changes. use the `shell_scripts/madis_conv.csh` script. there are script options for hourly output files, or a single daily output file.
2. if you aren't using the wrf preprocessing program, you're ready to go.
3. if you do want to do subsequent wrf preprocessing, you need to:
    1. decide on the windowing. each platform has a different convention and if you're going to put them into the wrf preprocessing you'll need to have the windowing match. use the `shell_scripts/windowing.csh` script.
    2. the wrf preprocessing takes a list of files and assumes they will all be assimilated at the same time, for superob'ing purposes, so it should match the expected assimilation window when running filter.

6.180.2 Data sources

http://madis.noaa.gov

There are two satellite wind converter programs; the one in this directory and one in the *SSEC Data Center* directory. The observations distributed here come from NESDIS. The SSEC observations are processed by SSEC itself and will differ from the observations converted here.

6.180.3 Programs

The programs in the `DART/observations/MADIS/` directory extract data from the distribution files and create DART observation sequence (obs_seq) files. Build them in the `work` directory by running the `./quickbuild.csh` script. In addition to the converters, the `advance_time` and `obs_sequence_tool` utilities will be built.

There are currently converters for these data types:

ACARS aircraft T,U,V,Q data	convert_madis_acars
Marine surface data	convert_madis_marine
Mesonet surface data	convert_madis_mesonet
Metar data	convert_madis_metar
Wind Profiler data	convert_madis_profiler
Rawinsonde/Radiosonde data	convert_madis_rawin
Satellite Wind data	convert_madis_satwnd

Example data files are in the `data` directory. Example scripts for converting batches of these files are in the `shell_scripts` directory. These are *NOT* intended to be turnkey scripts; they will certainly need to be customized for your use. There are comments at the top of the scripts saying what options they include, and should be commented enough to indicate where changes will be likely to need to be made.

Several converters have compile-time choices for outputting various types of moist variables. Check the source code for more details. Some converters also read multiple T/F strings from the console (standard input) to control at run-time what types of observations to convert. Again, check the source code for more details.

Each converter has hard-coded input and output filenames:

convert_madis_acars:	acars_input.nc	obs_seq.acars
convert_madis_marine:	marine_input.nc	obs_seq.marine
convert_madis_mesonet:	mesonet_input.nc	obs_seq.mesonet
convert_madis_metar:	metar_input.nc	obs_seq.metar
convert_madis_profiler:	profiler_input.nc	obs_seq.profiler
convert_madis_rawin:	rawin_input.nc	obs_seq.rawin
convert_madis_satwnd:	satwnd_input.nc	obs_seq.satwnd

The expected usage pattern is that a script will copy, rename, or make a symbolic link from the actual input file (which often contains a timestamp in the name) to the fixed input name before conversion, and move the output file to an appropriate filename before the next invocation of the converter. If an existing observation sequence file of the same output name is found when the converter is run again, it will open that file and append the next set of observations to it.

6.181 QuikSCAT SeaWinds Data

6.181.1 Overview

NASA's QuikSCAT mission is described in http://winds.jpl.nasa.gov/missions/quikscat/. "QuikSCAT" refers to the satellite, "SeaWinds" refers to the instrument that provides near-surface wind speeds and directions over large bodies of water. QuikSCAT has an orbit of about 100 minutes, and the SeaWinds microwave radar covers a swath under the satellite. The swath is comprised of successive scans (or rows) and each scan has many wind-vector-cells (WVCs). For the purpose of this document, we will focus only the **Level 2B** product at 25km resolution. If you go to the official JPL data distribution site http://podaac.jpl.nasa.gov/DATA_CATALOG/quikscatinfo.html , we are using the product labelled **L2B OWV 25km Swath**. Each orbit consists of (potentially) 76 WVCs in each of 1624 rows or scans. The azimuthal diversity of the radar returns affects the error characteristics of the retrieved wind speeds and directions, as does rain, interference of land in the radar footprint, and very low wind speeds. Hence, not all wind retrievals are created equal.

The algorithm that converts the 'sigma naughts' (the measure of radar backscatter) into wind speeds and directions has multiple solutions. Each candidate solution is called an 'ambiguity', and there are several ways of choosing 'the best' ambiguity. Beauty is in the eye of the beholder. At present, the routine to convert the original L2B data files (one per orbit) in HDF format into the DART observation sequence file makes several assumptions:

1. All retrievals are labelled with a 10m height, in accordance with the retrieval algorithm.
2. Only the highest-ranked (by the MLE method) solution is desired.
3. Only the WVCs with a wvc_quality_flag of **zero** are desired.
4. The mission specification of a wind speed rms error of 2 ms (for winds less than 20 m/s) and 10% for windspeeds between 20 and 30 m/s can be extended to all winds with a qc flag of zero.
5. The mission specification of an error in direction of 20 degrees rms is applicable to all retrieved directions.

6. All retrievals with wind speeds less than 1.0 are not used.
7. The above error characterstics can be simplified when deriving the horizontal wind components (i.e. U,V). **Note :** this may or may not be a good assumption, and efforts to assimilate the speed and direction directly are under way.

6.181.2 Data sources

The NASA Jet Propulsion Laboratory (JPL) data repository has a collection of animations and data sets from this instrument. In keeping with NASA tradition, these data are in HDF format (specifically, HDF4), so if you want to read these files directly, you will need to install the HDF4 libraries (which can be downloaded from http://www.hdfgroup.org/products/hdf4/)

If you go to the official JPL data distribution site http://podaac.jpl.nasa.gov/DATA_CATALOG/quikscatinfo.html, we are using the product labelled **L2B OWV 25km Swath**. They are organized in folders by day ... with each orbit (each revolution) in one compressed file. There are 14 revolutions per day. The conversion to DART observation sequence format is done on each revolution, multiple revolutions may be combined 'after the fact' by any `obs_sequence_tool` in the `work` directory of any model.

6.181.3 Programs

There are several programs that are distributed from the JPL www-site, ftp://podaac.jpl.nasa.gov/pub/ocean_wind/quikscat/L2B/sw/; we specifically started from the Fortran file read_qscat2b.f and modified it to be called as a subroutine to make it more similar to the rest of the DART framework. The original `Makefile` and `read_qscat2b.f` are included in the DART distribution in the `DART/observations/quikscat` directory. You will have to modify the `Makefile` to build the executable.

convert_L2b.f90

`convert_L2b` is the executable that reads the HDF files distributed by JPL. `DART/observations/quikscat/work` has the expected `mkmf_convert_L2b` and `path_names_convert_L2b` files and compiles the executable in the typical DART fashion - with one exception. The location of the HDF (and possible dependencies) installation must be conveyed to the `mkmf` build mechanism. Since this information is not required by the rest of DART, it made sense (to me) to isolate it in the `mkmf_convert_L2b` script. **It will be necessary to modify the ``mkmf_convert_L2b`` script to be able to build ``convert_L2b``**. In particular, you will have to change the two lines specifying the location of the HDF (and probably the JPG) libraries. The rest of the script should require little, if any, modification.

set JPGDIR = */contrib/jpeg-6b_gnu-4.1.2-64* set HDFDIR = */contrib/hdf-4.2r4_gnu-4.1.2-64*

There are a lot of observations in every QuikSCAT orbit. Consequently, the observation sequence files are pretty large - particularly if you use the ASCII format. Using the binary format (i.e. *obs_sequence_nml:write_binary_obs_sequence = .true.*) will result in observation sequence files that are about *half* the size of the ASCII format.

Since there are about 14 QuikSCAT orbits per day, it may be useful to convert individual orbits to an observation sequence file and then concatenate multiple observation sequence files into one file per day. This may be trivially accomplished with the `obs_sequence_tool` program in any `model/xxxx/work` directory. Be sure to include the `'../../../obs_def/obs_def_QuikSCAT_mod.f90'` string in `input.nml&preprocess_nml:input_files` when you run `preprocess`.

Obs_to_table.f90, plot_wind_vectors.m

`DART/diagnostics/threed_sphere/obs_to_table.f90` is a potentially useful tool. You can run the observation sequence files through this filter to come up with a 'XYZ'-like file that can be readily plotted with `DART/diagnostics/matlab/plot_wind_vectors.m`.

6.181.4 Namelist

This namelist is read from the file `input.nml`. We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist. The following values are the defaults for these namelist items.

```
&convert_L2b_nml
   l2b_file = '',
   datadir = '.',
   outputdir = '.',
   lon1 = 0.0,
   lon2 = 360.0,
   lat1 = -90.0,
   lat2 = 90.0,
   along_track_thin = 0,
   cross_track_thin = 0
 /
```

It is possible to restrict the output observation sequence to contain data from a region of interest throught the use of the namelist parameters. If you need a region that spans the Prime Meridian lon1 can be a larger number than lon2, for example, a region from 300 E to 40 E and 60 S to 30 S (some of the South Atlantic), would be *lon1 = 300, lon2 = 40, lat1 = -60, lat2 = -30.*

Contents	Type	Description
l2b_file	character(len=128)	name of the HDF file to read - NOT including the directory, e.g. QS_S2B44444.20080021548
datadir	character(len=128)	the directory containing the HDF files
outputdir	character(len=128)	the directory for the output observation sequence files.
lon1	real(r4)	the West-most longitude of interest in degrees. [0.0, 360]
lon2	real(r4)	the East-most longitude of interest in degrees. [0.0, 360]
lat1	real(r4)	the South-most latitude of interest in degrees. [-90.0, 90.0]
lat2	real(r8)	the North-most latitude of interest in degrees. [-90.0, 90.0]
along_track_thin	integer	provides ability to thin the data by keeping only every Nth row. e.g. 3 == keep every 3rd row.
cross_track_thin	integer	provides ability to thin the data by keeping only every Nth wind vector cell in a particular row. e.g. 5 == keep every 5th cell.

6.182 AIRS Observations

6.182.1 Overview

The AIRS instrument is an Atmospheric Infrared Sounder flying on the Aqua spacecraft. Aqua is one of a group of satellites flying close together in a polar orbit, collectively known as the “A-train”. The programs in this directory help to extract the data from the distribution files and put them into DART observation sequence (obs_seq) file format.

AIRS data includes atmospheric temperature in the troposphere, derived moisture profiles, land and ocean surface temperatures, surface emmissivity, cloud fraction, cloud top height, and ozone burden in the atmosphere.

6.182.2 Data sources

Access to the web pages where the AIRS data are stored is available by registering as a data user.

There are two products this converter can be used on: AIRX2RET, which is the L2 standard retrieval product using AIRS IR and AMSU (without-HSB); and AIRS2RET, which is the L2 standard retrieval product using AIRS IR-only. More detailed information on the AIRS2RET data product and the AIRX2RET data product is available from the nasa web pages.

The data is distributed in HDF-4 format, using some additional conventions for metadata called HDF-EOS. There is a basic library for accessing data in hdf files, and a variety of generic tools that work with hdf files. The specific libraries we use are the HDF-EOS2 library built on HDF4. The web page has a link to specific build instructions. Also, see below on this web page for very specific instructions for getting the required software and building it. If you find more recent instructions online, use those. But in the absence of anything else, it’s someplace to start.

Besides the programs in this directory, a variety of specific tools targeted at AIRS data are available to help read and browse the data. General information on using hdf in the earth sciences is available here.

Several types of AIRS data, with varying levels of processing, are available. The following descriptions are taken from the V5_Data_Release_UG document:

> The L1B data product includes geolocated, calibrated observed microwave, infrared and visible/near infrared radiances, as well as Quality Assessment (QA) data. The radiances are well calibrated; however, not all QA data have been validated. Each product granule contains 6 minutes of data. Thus there are 240 granules of each L1B product produced every day.
>
> The L2 data product includes geolocated, calibrated cloud-cleared radiances and 2-dimensional and 3-dimensional retrieved physical quantities (e.g., surface properties and temperature, moisture, ozone, carbon monoxide and methane profiles throughout the atmosphere). Each product granule contains 6 minutes of data. Thus there are 240 granules of each L2 product produced every day.
>
> The L3 data are created from the L2 data product by binning them in 1°x1° grids. There are three products: daily, 8-day and monthly. Each product provides separate ascending (daytime) and descending (nighttime) binned data sets.

The converter in this directory processes level 2 (L2) data files, using data set `AIRS_DP` and data product `AIRX2RET` or `AIRS2RET` without `HSB` (the instrument measuring humidity which failed).

The Atmospheric Infrared Sounder (AIRS) is a facility instrument aboard the second Earth Observing System (EOS) polar-orbiting platform, EOS Aqua. In combination with the Advanced Microwave Sounding Unit (AMSU) and the Humidity Sounder for Brazil (HSB), AIRS constitutes an innovative atmospheric sounding group of visible, infrared, and microwave sensors. AIRS data will be generated continuously. Global coverage will be obtained twice daily (day and night) on a 1:30pm sun synchronous orbit from a 705-km altitude.

The AIRS Standard Retrieval Product consists of retrieved estimates of cloud and surface properties, plus profiles of retrieved temperature, water vapor, ozone, carbon monoxide and methane. Estimates of the errors associated with these quantities will also be part of the Standard Product. The temperature profile vertical resolution is 28 levels total between 1100 mb and 0.1 mb, while moisture profile is reported at 14 atmospheric layers between 1100 mb and 50 mb. The horizontal resolution is 50 km. An AIRS granule has been set as 6 minutes of data, 30 footprints cross track by 45 lines along track. The Shortname for this product is AIRX2RET. (AIRS2RET is the same product but without the AMSU data.)

The converter outputs temperature observations at the corresponding vertical pressure levels. However, the moisture obs are the mean for the layer, so the location in the vertical is the midpoint, in log space, of the current layer and the layer above it. There is an alternative computation for the moisture across the layer which may be more accurate, but requires a forward operator subroutine to be written and for the observation to contain metadata. The observation could be defined with a layer top, in pressure, and a number of points to use for the integration across the layer. Then the forward operator would query the model at each of the N points in the vertical for a given horizontal location, and compute the mean moisture value. This code has not been implemented yet, and would require a different QTY_xxx to distinguish it from the simple location/value moisture obs. See the GPS non-local operator code for an example of how this would need to be implemented.

Getting the data currently means putting in a start/stop time at this web page. The keyword is `AIRX2RET` and put in the time range of interest and optionally a geographic region. Each file contains 6 minutes of data, is about 2.3 Megabytes, and globally there are 240 files/day (about 550 Megabytes/day). There are additional options for getting only particular variables of interest, but the current reader expects whole files to be present. Depending on your connection to the internet, there are various options for downloading. We have chosen to download a `wget` script which is created by the web page after adding the selected files to a ‘cart’ and ‘checking out’. The script has a series of `wget` commands which downloads each file, one at a time, which is run on the machine where you want the data to end up.

6.182.3 Programs

The temperature observations are located on standard levels; there is a single array of heights in each file and all temperature data is located on one of these levels. The moisture observations, however, are an integrated quantity for the space between the levels; in their terminology the fixed heights are ‘levels’ and the space between them are ‘layers’. The current converter locates the moisture obs at the midpoint, in log space, between the levels.

The hdf files need to be downloaded from the data server, in any manner you choose. The converter program reads each hdf granule and outputs a DART obs_seq file containing up to 56700 observations. Only those with a quality control of 0 (Best) are kept. The resulting obs_seq files can be merged with the *program obs_sequence_tool* into larger time periods.

It is possible to restrict the output observation sequence to contain data from a region of interest throught the use of the namelist parameters. If you need a region that spans the Prime Meridian lon1 can be a larger number than lon2, for example, a region from 300 E to 40 E and 60 S to 30 S (some of the South Atlantic), would be *lon1 = 300, lon2 = 40, lat1 = -60, lat2 = -30*.

The scripts directory here includes some shell scripts that make use of the fact that the AIRS data is also archived on the NCAR HPSS (tape library) in daily tar files. The script has options to download a day of granule files, convert them, merge them into daily files, and remove the original data files and repeat the process for any specified time period. (See `oneday_down.sh`)

Here is a very specific script I used to build the required libraries on a Linux cluster. If you find more up-to-date instructions, use those. But in the absence of anything else, here’s a place to start:

```
wget https://observer.gsfc.nasa.gov/ftp/edhs/hdfeos/latest_release/*

# NOTE: direct ftp does not work for me anymore

##ftp edhs1.gsfc.nasa.gov
### (log in as 'anonymous' and your email as the password)
##cd /edhs/hdfeos/latest_release
```

(continues on next page)

(continued from previous page)

```
##mget *
##quit

# mar 2013, the dir contents:
#
# hdf-4.2.6.tar.gz
# HDF-EOS2.18v1.00.tar.Z
# HDF-EOS2.18v1.00_TestDriver.tar.Z
# HDF_EOS_REF.pdf
# HDF_EOS_UG.pdf
# jpegsrc.v6b.tar.gz
# zlib-1.2.5.tar.gz
#
# (i skipped a 'windows' dir).
#
# mar 2019 contents:
#      HDF-EOS2.20v1.00.tar.Z  08-Jan-2018 15:21  7.3M
#      HDF-EOS2.20v1.00_Tes..> 08-Jan-2018 15:21  9.5M
#      HDF-EOS_REF.pdf         07-Nov-2018 13:45  695K
#      HDF-EOS_UG.pdf          08-Jan-2018 15:28  429K
#      hdf-4.2.13.tar.gz       08-Jan-2018 15:14  4.3M
#      jpegsrc.v9b.tar.gz      09-Jan-2018 13:44  1.0M
#      zlib-1.2.11.tar.gz      08-Jan-2018 15:22  593K
#
for i in *.tar.gz
do
  tar -zxvf $i
done

#
# start with smaller libs, work up to HDF-EOS.
#
#

echo zlib:

cd zlib-1.2.11
./configure --prefix=/glade/p/work/nancy
make
make test
make install

echo jpeg:

cd jpeg-9b
./configure --prefix=/glade/p/work/nancy
make
make test
mkdir /glade/p/work/nancy/{bin,man,man/man1}
make install

# (make install wouldn't create the dirs if they didn't exist.
# lib was there from the zlib install, the others weren't.)

echo hdf:

cd hdf-4.2.13
```

(continues on next page)

(continued from previous page)

```
./configure --prefix=/glade/p/work/nancy
# (it found zlib and jpeg, from the install prefix i guess)
make
# (there is apparently no 'make test')
make install

echo hdf-eos:

cd hdfeos
./configure CC='/glade/p/work/nancy/bin/h4cc -Df2cFortran' --prefix=/glade/p/
↪work/nancy
# (the CC= is crucial)
make
# (i didn't build the test drivers so i didn't do make test)
make install


echo AIRS converter:

cd $DART/observations/AIRS/work

echo edit mkmf_convert_airs_L2 to have all the base paths
echo be /glade/p/work/nancy instead of whatever.  make it look like:
echo ' '
echo 'set JPGDIR = /glade/work/nancy'
echo 'set HDFDIR = /glade/work/nancy'
echo 'set EOSDIR = /glade/work/nancy'
echo ' '

./quickbuild.csh

exit 0
```

6.182.4 Namelist

This namelist is read in a file called `input.nml`. We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&convert_airs_L2_nml
   l2_files            = 'input.hdf',
   l2_file_list        = '',
   datadir             = '.',
   outputdir           = '.',
   lon1                =   0.0,
   lon2                = 360.0,
   lat1                = -90.0,
   lat2                =  90.0,
   min_MMR_threshold   = 1.0e-30,
   top_pressure_level  = 0.0001,
   cross_track_thin    = 0,
   along_track_thin    = 0,
/
```

Contents	Type	Description	Default
l2_files	character(len=128) (:)	A list of one or more names of the HDF file(s) to read, NOT including the directory. If multiple files are listed, each will be read and the results will be placed in a separate file with an output filename constructed based on the input filename.	
l2_file_list	character(len=128)	The name of an ascii text file which contains one filename per line, NOT including the directory. Each file will be read and the observations converted into an output file where the output filename is based on the input filename. Only one of 'l2_files' and 'l2_file_list' can be specified. The other must be ' ' (empty).	
datadir	character(len=128)	The directory containing the HDF files	
outputdir	character(len=128)	The directory for the output observation sequence files.	
lon1	real(r4)	the West-most longitude of interest in degrees. [0.0, 360]	
lon2	real(r4)	the East-most longitude of interest in degrees. [0.0, 360]	
lat1	real(r4)	the South-most latitude of interest in degrees. [-90.0, 90.0]	
lat2	real(r8)	the North-most latitude of interest in degrees. [-90.0, 90.0]	
min_MMR_threshold	real(r8)	The data files contains 'Retrieved Water Vapor Mass Mixing Ratio'. This is the minimum threshold, in gm/kg, that will be converted into a specific humidity observation.	
cross_track_thin	integer	provides ability to thin the data by keeping only every Nth data value in a particular row. e.g. 3 == keep every third value.	
along_track_thin	integer	provides ability to thin the data by keeping only every Nth row. e.g. 4 == keep only every 4th row.	

6.183 Aviso+/CMEMS Observations

6.183.1 Overview

This short description of the SEALEVEL_GLO_SLA_L3_REP_OBSERVATIONS_008_018 product is repeated from the **INFORMATION** tab from the Copernicus Marine Environment Monitoring Service online catalogue (in April 2017).

> For the Global Ocean- Mono altimeter satellite along-track sea surface heights computed with respect to a twenty-year mean. Previously distributed by Aviso+, no change in the scientific content. All the missions are homogenized with respect to a reference mission which is currently Jason-2. This product is computed with an optimal and centered computation time window (6 weeks before and after the date). Two kinds of datasets are proposed: filtered (nominal dataset) and unfiltered.

The `convert_aviso.f90` program is designed to read a netCDF file containing the (Level 3) sea surface anomalies from any of the following platforms: "`Jason-1`", "`Envisat`", or "`Geosat Follow On`". One of those platforms must be listed in the netCDF file global attribute: `platform`

The data files have names like:
dt_global_j1_sla_vfec_20080101_20140106.nc,
dt_global_en_sla_vfec_20080101_20140106.nc, or
dt_global_g2_sla_vfec_20080101_20140106.nc; corresponding to the Jason-1, Envisat, and the Geosat Follow On platforms.
The DART observation TYPE corresponding to each of these platforms are J1_SEA_SURFACE_ANOMALY, EN_SEA_SURFACE_ANOMALY, and GFO_SEA_SURFACE_ANOMALY, respectively and are defined in obs_def_ocean_mod.f90.
Fred wrote a python script (shell_scripts/convert_aviso.py) to repeatedly call convert_aviso and decided it was easiest to simply provide the input file name as a command line argument and always have the output file have the name obs_seq.aviso. As such, there is no input namelist specifically for these parameters, but other DART modules still require run-time crontrol specified by input.nml.

After creating a large number of output observation sequence files, it is usually necessary to consolidate the files and subset them into files containing just the timeframe required for a single assimilation. **NOTE**: the obs_sequence_tool is constructed for just this purpose.
The shell_scripts/makedaily.sh script attempts to consolidate all the SLA observations and those that may have been (separately) converted from the World Ocean Database into 24-hour segments centered at midnight GMT. You will have to modify the makedaily.sh script to suit your filesystem and naming convention. It is provided as a starting point.

Reminder: (according to the data providers): In order to compute Absolute Dynamic Topography, the Mean Dynamic Topography (MDT) can be added. It is distributed by Aviso+ (http://www.aviso.altimetry.fr/en/data/products/auxiliary-products/mdt.html). Fred was using this product in assimilations with POP, so he chose a different source for MDT - consistent with POP's behavior.

6.183.2 Data sources

The Copernicus Marine and Environment Monitoring Service (CMEMS) has taken over the processing and distribution of the Ssalto/Duacs multimission altimeter products formerly administered by Aviso+. After a registration process, the along-track sea level anomalies (SLA) may be downloaded from http://marine.copernicus.eu/services-portfolio/access-to-products/ - search for the SEALEVEL_GLO_SLA_L3_REP_OBSERVATIONS_008_018 if it does not come up directly.

6.183.3 Programs

convert_aviso.f90	does the actual conversion from netCDF to a DART observation sequence file, which may be ASCII or binary.
shell_scripts/convert_aviso.py	python script to convert a series of input files and datestamp the output files.
shell_scripts/makedaily.sh	shell script to repeatedly call obs_sequence_tool to consolidate multiple observation sequence files into an observation sequence file that has ALL the observations from ALL platforms in a single file. makedaily.sh is capable of looping over time ranges and creating observation sequences for each time range.

6.183.4 Namelist

There is no namelist for `convert_aviso`, but other namelists control aspects of the execution, namely `&obs_sequence_nml:write_binary_obs_sequence`. see *MODULE obs_sequence_mod*.

6.183.5 Modules used

```
assimilation_code/location/threed_sphere/location_mod.f90
assimilation_code/modules/assimilation/assim_model_mod.f90
assimilation_code/modules/io/dart_time_io_mod.f90
assimilation_code/modules/observations/obs_kind_mod.f90
assimilation_code/modules/observations/obs_sequence_mod.f90
assimilation_code/modules/utilities/ensemble_manager_mod.f90
assimilation_code/modules/utilities/null_mpi_utilities_mod.f90
assimilation_code/modules/utilities/random_seq_mod.f90
assimilation_code/modules/utilities/sort_mod.f90
assimilation_code/modules/utilities/time_manager_mod.f90
assimilation_code/modules/utilities/types_mod.f90
assimilation_code/modules/utilities/utilities_mod.f90
models/template/model_mod.f90
observations/forward_operators/obs_def_mod.f90
observations/obs_converters/AVISO/convert_aviso.f90
observations/obs_converters/utilities/obs_utilities_mod.f90
```

6.184 null_model

6.184.1 Overview

DART interface module for the 'null_model'. This model provides very simple models for evaluating filtering algorithms. It can provide simple linear growth around a fixed point, a random draw from a Gaussian, or combinations of the two. Namelist controls can set the width of the Gaussian and change both the model advance method and the expected observation interpolation method.

The 18 public interfaces are standardized for all DART compliant models. These interfaces allow DART to advance the model, get the model state and metadata describing this state, find state variables that are close to a given location, and do spatial interpolation for model state variables.

6.184.2 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   model_size           = 2,
   delta_t              = 0.05,
   time_step_days       = 0,
   time_step_seconds    = 3600
   noise_amplitude      = 0.0_r8
   advance_method       = 'simple'
   interpolation_method = 'standard'
/
```

Description of each namelist entry

Item	Type	Description
model_size	integer	Model size.
delta_t	real(r8)	Internal model timestep parameter.
time_step_days	integer	Minimum model advance time in days.
time_step_seconds	integer	Minimum model advance time in seconds.
noise_amplitude	real(r8)	If greater than 0.0 sets the standard deviation of the added Gaussian noise during the model advance.
advance_method	character(64)	Controls the model advance method. The default is 'simple' timestepping. A 4-step Runga Kutta method can be selected with the string 'rk'.
interpolation_method	character(64)	Controls how the expected value of an observation is computed. The default is 'standard' which uses a linear interpolation between the two surrounding model points. Other options include 'square' which returns the square of the computed value, 'opposite_side' which adds on a value from the opposite side of the cyclical domain, and 'average' which averages 15 points to get the expected value. Model size should be > 15 to use the last option.

6.184.3 Files

filename	purpose
input.nml	to read the model_mod namelist
preassim.nc	the time-history of the model state before assimilation
analysis.nc	the time-history of the model state after assimilation
dart_log.out [default name]	the run-time diagnostic output
dart_log.nml [default name]	the record of all the namelists actually USED - contains the default values

6.185 PROGRAM `dart_to_ncommas`

`dart_to_ncommas` is the program that **updates** a ncommas netCDF-format restart file (usually `ncommas_restart.nc`) with the state information contained in a DART output/restart file (e.g. `perfect_ics, filter_ics, ...`). Only the CURRENT values in the ncommas restart file will be updated. The DART model time is compared to the time in the ncommas restart file. If the last time in the restart file does not match the DART model time, the program issues an error message and aborts.

From the user perspective, most of the time `dart_to_ncommas` will be used on DART files that have a header containing one time stamp followed by the model state.

The dart_to_ncommas_nml namelist allows `dart_to_ncommas` to read the `assim_model_state_ic` files that have *two* timestamps in the header. These files are temporarily generated when DART is used to advance the model. One timestamp is the 'advance_to' time, the other is the 'valid_time' of the model state. In this case, a namelist for ncommas (called `ncommas_in.DART`) is written that contains the `&time_manager_nml` settings appropriate to advance ncommas to the time requested by DART. The repository version of the `advance_model.csh` script has a section to ensure the proper DART namelist settings for this case.

Conditions required for successful execution of `dart_to_ncommas`:

- a valid `input.nml` namelist file for DART
- a valid `ncommas_vars.nml` namelist file for ncommas - the same one used to create the DART state vector, naturally,
- a DART file (typically `filter_restart.xxxx` or `filter_ics.xxxx`)
- a ncommas restart file (typically `ncommas_restart.nc`).

Since this program is called repeatedly for every ensemble member, we have found it convenient to link the DART input file to the default input filename (`dart_restart`). The same thing goes true for the ncommas output filename `ncommas_restart.nc`.

6.185.1 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   ncommas_restart_filename      = 'ncommas_restart.nc';
   assimilation_period_days      = 1,
   assimilation_period_seconds   = 0,
   model_perturbation_amplitude  = 0.2,
   output_state_vector           = .true.,
   calendar                      = 'Gregorian',
   debug                         = 0
/
```

```
&dart_to_ncommas_nml
   dart_to_ncommas_input_file = 'dart_restart',
   advance_time_present    = .false.
/
```

`dart_to_ncommas_nml` and `model_nml` are always read from a file called `input.nml`. The full description of the `model_nml` namelist is documented in the NCOMMAS model_mod.

Item	Type	Description
dart_to_ncommas_input_file	character(len=128)	The name of the DART file containing the model state to insert into the ncommas restart file.
advance_time_present	logical	If you are converting a DART initial conditions or restart file this should be `.false.`; these files have a single timestamp describing the valid time of the model state. If `.true.` TWO timestamps are expected to be the DART file header. In this case, a namelist for ncommas (called `ncommas_in.DART`) is created that contains the `&time_manager_nml` settings appropriate to advance ncommas to the time requested by DART.

ncommas_vars_nml is always read from a file called ncommas_vars.nml.

Item	Type	Description
ncommas_state_variables	character(len=NF90_MAX_NAME) :: dimension(160)	The list of variable names in the NCOMMAS restart file to use to create the DART state vector and their corresponding DART kind.

```
&ncommas_vars_nml
   ncommas_state_variables = 'U',   'QTY_U_WIND_COMPONENT',
                             'V',   'QTY_V_WIND_COMPONENT',
                             'W',   'QTY_VERTICAL_VELOCITY',
                             'TH',  'QTY_POTENTIAL_TEMPERATURE',
                             'DBZ', 'QTY_RADAR_REFLECTIVITY',
                             'WZ',  'QTY_VERTICAL_VORTICITY',
                             'PI',  'QTY_EXNER_FUNCTION',
                             'QV',  'QTY_VAPOR_MIXING_RATIO',
                             'QC',  'QTY_CLOUDWATER_MIXING_RATIO',
                             'QR',  'QTY_RAINWATER_MIXING_RATIO',
                             'QI',  'QTY_ICE_MIXING_RATIO',
                             'QS',  'QTY_SNOW_MIXING_RATIO',
                             'QH',  'QTY_GRAUPEL_MIXING_RATIO'
  /
```

6.185.2 Modules used

```
assim_model_mod
location_mod
model_mod
null_mpi_utilities_mod
obs_kind_mod
random_seq_mod
time_manager_mod
types_mod
utilities_mod
```

6.185.3 Files read

- DART initial conditions/restart file; e.g. `filter_ic`
- DART namelist file; `input.nml`
- ncommas namelist file; `ncommas_vars.nml`
- ncommas restart file `ncommas_restart.nc`

6.185.4 Files written

- ncommas restart file; `ncommas_restart.nc`
- ncommas namelist file; `ncommas_in.DART`

6.185.5 References

none

6.186 PROGRAM `ncommas_to_dart`

`ncommas_to_dart` is the program that reads a ncommas restart file (usually `ncommas_restart.nc`) and creates a DART state vector file (e.g. `perfect_ics, filter_ics, ...`).
The list of variables used to create the DART state vector are specified in the `ncommas_vars.nml` file.
Conditions required for successful execution of `ncommas_to_dart`:

- a valid `input.nml` namelist file for DART
- a valid `ncommas_vars.nml` namelist file for ncommas
- the ncommas restart file mentioned in the `input.nml&model_nml:ncommas_restart_filename` variable.

Since this program is called repeatedly for every ensemble member, we have found it convenient to link the ncommas restart files to the default input filename (`ncommas_restart.nc`). The default DART state vector filename is `dart_ics` - this may be moved or linked as necessary.

6.186.1 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   ncommas_restart_filename     = 'ncommas_restart.nc';
   assimilation_period_days     = 1,
   assimilation_period_seconds  = 0,
   model_perturbation_amplitude = 0.2,
   output_state_vector          = .true.,
   calendar                     = 'Gregorian',
   debug                        = 0
/
```

```
&ncommas_to_dart_nml
   ncommas_to_dart_output_file = 'dart_ics'
/
```

`ncommas_to_dart_nml` and `model_nml` are always read from a file called `input.nml`. The full description of the `model_nml` namelist is documented in the NCOMMAS model_mod.

Item	Type	Description
ncommas_to_dart_output_file	character(len=128)	The name of the DART file which contains the updated model state info that should be written into the NCOMMAS file.

`ncommas_vars_nml` is always read from a file called `ncommas_vars.nml`.

Item	Type	Description
ncommas_state_variables	character(len=NF90_MAX_NAME) :: dimension(160)	The list of variable names in the NCOMMAS restart file to use to create the DART state vector and their corresponding DART kind.

```
&ncommas_vars_nml
   ncommas_state_variables = 'U',   'QTY_U_WIND_COMPONENT',
                             'V',   'QTY_V_WIND_COMPONENT',
                             'W',   'QTY_VERTICAL_VELOCITY',
                             'TH',  'QTY_POTENTIAL_TEMPERATURE',
                             'DBZ', 'QTY_RADAR_REFLECTIVITY',
                             'WZ',  'QTY_VERTICAL_VORTICITY',
                             'PI',  'QTY_EXNER_FUNCTION',
                             'QV',  'QTY_VAPOR_MIXING_RATIO',
                             'QC',  'QTY_CLOUDWATER_MIXING_RATIO',
                             'QR',  'QTY_RAINWATER_MIXING_RATIO',
                             'QI',  'QTY_ICE_MIXING_RATIO',
                             'QS',  'QTY_SNOW_MIXING_RATIO',
                             'QH',  'QTY_GRAUPEL_MIXING_RATIO'
  /
```

6.186.2 Modules used

```
assim_model_mod
location_mod
model_mod
null_mpi_utilities_mod
obs_kind_mod
random_seq_mod
time_manager_mod
types_mod
utilities_mod
```

6.186.3 Files read

- ncommas restart file; `ncommas_restart.nc`
- DART namelist files; `input.nml` and `ncommas_vars.nml`

6.186.4 Files written

- DART state vector file; e.g. `dart_ics`

6.186.5 References

none

6.187 MODULE dart_pop_mod (POP)

6.187.1 Overview

`dart_pop_mod` provides a consistent collection of routines that are useful for multiple programs e.g. `dart_to_pop`, `pop_to_dart`, etc.

6.187.2 Namelist

There are no namelists unique to this module. It is necessary for this module to read some of the POP namelists, and so they are declared in this module. In one instance, DART will read the `time_manager_nml` namelist and **write** an updated version to control the length of the integration of POP. All other information is simply read from the namelists and is used in the same context as POP itself. The POP documentation should be consulted. **Only the variables of interest to DART are described in this document.**

All namelists are read from a file named `pop_in`.

```
namelist /time_manager_nml/  allow_leapyear, stop_count, stop_option
```

`dart_to_pop` controls the model advance of LANL/POP by creating a `&time_manager_nml` in `pop_in.DART` **IFF** the DART state being converted has the 'advance_to_time' record. The `pop_in.DART` must be concatenated with the other namelists needed by POP into a file called `pop_in` . We have chosen to store the other namelists (which contain static information) in a file called `pop_in.part2`. Initially, the `time_manager_nml` is stored in a companion file called `pop_in.part1` and the two files are concatenated into the expected `pop_in` - then, during the course of an assimilation experiment, DART keeps writing out a new `time_manager_nml` with new integration information - which gets appended with the static information in `pop_in.part2`

Contents	Type	Description
allow_leapyear	logical	DART ignores the setting of this parameter. All observations must use a Gregorian calendar. There are pathological cases, but if you are doing data assimilation, just use the Gregorian calendar. *[default: .true.]*
stop_count	integer	the number of model advance steps to take. *[default: 1]*
stop_option	character(len=64)	The units for the number of model advance steps (`stop_count`) to take. *[default: 'ndays']*

```
namelist /io_nml/  luse_pointer_files, pointer_filename
```

Contents	Type	Description
luse_pointer_files	logical	switch to indicate the use of pointer files or not. If `.true.`, a pointer file is used to contain the name of the restart file to be used. DART requires this to be `.true`. *[default: .true.]*
pointer_filename	character(len=100)	The name of the pointer file. All of the DART scripts presume and require the use of the default. Each ensmeble member gets its own pointer file. *[default: rpointer.ocn.[1-N].restart]*

```
namelist /restart_nml/  restart_freq_opt, restart_freq
```

Contents	Type	Description
luse_pointer_files	logical	switch to indicate the use of pointer files or not. If `.true.`, a pointer file is used to contain the name of the restart file to be used. DART requires this to be `.true`. *[default: .true.]*
pointer_filename	character(len=100)	The name of the pointer file. All of the DART scripts presume and require the use of the default. Each ensmeble member gets its own pointer file. *[default: rpointer.ocn.[1-N].restart]*

```
namelist /init_ts_nml/  init_ts_option, init_ts_file, init_ts_file_fmt
```

The `dart_pop_mod:initialize_module()` routine reads `pop_in` . There are several code stubs for future use that may allow for a more fully-supported POP namelist implementation. This namelist is one of them. Until further notice, the `init_ts_nml` is completely ignored by DART.

Contents	Type	Description
init_ts_option	character(len=64)	NOT USED by DART. All T,S information comes from a netCDF restart file named `pop.r.nc` *[default: 'restart']*
init_ts_file	character(len=100)	NOT USED by DART. All T,S information comes from `pop.r.nc` *[default: 'pop.r']*
init_ts_file_fmt	character(len=64)	NOT USED by DART. The file format is `'nc'` *[default: 'nc']*

```
namelist /domain_nml/  ew_boundary_type
```

DART needs to know if the East-West domain is cyclic for spatial interpolations. Presently, DART has only been tested for the dipole grid, which is cyclic E-W and closed N-S.

Contents	Type	Description
ew_boundary_type	character(len=64)	switch to indicate whether the East-West domain is cyclic or not. DART/POP has not been tested in a regional configuration, so DART requires this to be `'cyclic'`. *[default: 'cyclic']*

```
namelist /grid_nml/  horiz_grid_opt,  vert_grid_opt,  topography_opt, &
                               horiz_grid_file, vert_grid_file, topography_file
```

The POP grid information comes in several files: horizontal grid lat/lons in one, the vertical grid spacing in another, and the topography (lowest valid vertical level) in a third.
Here is what we can get from the (binary) horizontal grid file:

```
real(r8), dimension(:,:) :: ULAT,  &! latitude  (radians) of U points
real(r8), dimension(:,:) :: ULON,  &! longitude (radians) of U points
real(r8), dimension(:,:) :: HTN ,  &! length (cm) of north edge of T box
real(r8), dimension(:,:) :: HTE ,  &! length (cm) of east  edge of T box
real(r8), dimension(:,:) :: HUS ,  &! length (cm) of south edge of U box
real(r8), dimension(:,:) :: HUW ,  &! length (cm) of west  edge of U box
real(r8), dimension(:,:) :: ANGLE  &! angle
```

The vertical grid file is ascii, with 3 columns/line:

```
cell thickness(in cm)   cell center(in m)   cell bottom(in m)
```

Here is what we can get from the topography file:

```
integer, dimension(:,:), :: KMT    &! k index of deepest grid cell on T grid
```

These must be derived or come from someplace else . . .

```
KMU                k index of deepest grid cell on U grid
HT                 real(r8) value of deepest valid T depth (in cm)
HU                 real(r8) value of deepest valid U depth (in cm)
```

Contents	Type	Description
horiz_grid_opt, vert_grid_opt, topography_opt	character(len=64)	switch to indicate whether or not the grids will come from an external file or not. DART requires ALL of these to be `'file'`. *[default: 'file']*
horiz_grid_file	character(len=100)	The name of the binary file containing the values for the horizontal grid. The **dimensions** of the grid are read from `pop.r.nc`. It would have been nice to include the actual grid information in the netCDF files. *[default: 'horiz_grid.gx3v5.r8ieee.le']*
vert_grid_file	character(len=100)	The name of the ASCII file containing the values for the vertical grid. The file must contain three columns of data pertaining to the cell thickness (in cm), the cell center (in meters), and the cell bottom (in meters). Again, it would have been nice to include the vertical grid information in the netCDF files. *[default: 'vert_grid.gx3v5']*
topography_grid_file	character(len=100)	The name of the binary file containing the values for the topography information. The **dimensions** of the grid are read from `pop.r.nc`. *[default: 'topography.gx3v5.r8ieee.le']*

6.187.3 Other modules used

```
types_mod
time_manager_mod
utilities_mod
typesizes
netcdf
```

6.187.4 Public interfaces

Only a select number of interfaces used are discussed here. Each module has its own discussion of their routines.

Interface routines

use dart_pop_mod, only :	get_pop_calendar
	set_model_time_step
	get_horiz_grid_dims
	get_vert_grid_dim
	read_horiz_grid
	read_topography
	read_vert_grid
	write_pop_namelist
	get_pop_restart_filename

Required interface routines

call get_pop_calendar(calstring)

```
character(len=*), intent(out) :: calstring
```

Returns a string containing the type of calendar in use.

`calstring`	DART/POP uses a 'gregorian' calendar.

poptimestep = set_model_time_step()

```
type(time_type), intent(out) :: poptimestep
```

`set_model_time_step` returns the model time step that was set in the restart_nml`restart_freq`. This is the minimum amount of time DART thinks the POP model can advance. Indirectly, this specifies the minimum assimilation interval.

`poptimestep`	the minimum assimilation interval

call get_horiz_grid_dims(Nx, Ny)

```
integer, intent(out) :: Nx, Ny
```

`get_horiz_grid_dims` reads `pop.r.nc` to determine the number of longitudes and latitudes.

`Nx`	the length of the 'i' dimension in the POP restart file. The number of longitudes in use.
`Ny`	the length of the 'j' dimension in the POP restart file. The number of latitudes in use.

call get_vert_grid_dim(Nz)

```
integer, intent(out) :: Nz
```

`get_vert_grid_dim` reads `pop.r.nc` to determine the number of vertical levels in use.

`Nz`	the length of the 'k' dimension in the POP restart file. The number of vertical levels in use.

call read_horiz_grid(nx, ny, ULAT, ULON, TLAT, TLON)

```
integer,                    intent(in)  :: nx, ny
real(r8), dimension(nx,ny), intent(out) :: ULAT, ULON, TLAT, TLON
```

`read_horiz_grid` reads the direct access binary files containing the POP grid information. **The first record is REQUIRED to be 'ULAT', the second record is REQUIRED to be 'ULON'.**

`nx`	The number of longitudes in the grid.
`ny`	The number of latitudes in the grid.
`ULAT`	The matrix of latitudes for the UVEL and VVEL variables. Units are degrees [-90,90].
`ULON`	The matrix of longitudes for the UVEL and VVEL variables. Units are degrees. [0,360]
`TLAT`	The matrix of latitudes for the SALT and TEMP variables. Units are degrees [-90,90].
`TLON`	The matrix of longitudes for the SALT and TEMP variables. Units are degrees. [0,360]

call read_topography(nx, ny, KMT, KMU)

```
integer,                   intent(in)  :: nx, ny
integer, dimension(nx,ny), intent(out) :: KMT, KMU
```

`read_topography` reads the direct access binary files containing the POP topography information. **The first record is REQUIRED to be 'KMT'.** 'KMU' is calculated from 'KMT'.

nx	The number of longitudes in the grid.
ny	The number of latitudes in the grid.
KMT	The matrix containing the lowest valid depth index at grid centroids.
KMU	The matrix containing the lowest valid depth index at grid corners.

call read_vert_grid(nz, ZC, ZG)

```
integer,                   intent(in)  :: nz
real(r8), dimension(nz), intent(out) :: ZC, ZG
```

read_vert_grid reads the ASCII file containing the information about the vertical levels. The file must contain three columns of data pertaining to; 1) the cell thickness (in cm),

2) the cell center (in meters),

and 3) the cell bottom (in meters).

nz	The number of vertical levels.
ZC	The depth (in meters) at the grid centers.
ZG	The depth (in meters) at the grid edges.

call write_pop_namelist(model_time, adv_to_time)

```
type(time_type), intent(in)  :: model_time
type(time_type), intent(in)  :: adv_to_time
```

write_pop_namelist writes the POP namelist time_manager_nml with the information necessary to advance POP to the next assimilation time. The namelist is written to a file called pop_in.DART. Presently, DART is configured to minimally advance POP for 86400 seconds - i.e. 1 day. The forecast length (the difference between 'model_time' and 'adv_to_time') must be an integer number of days with the current setup. An error will result if it is not.

model_time	The 'valid' time of the current model state.
adv_to_time	The time of the next assimilation.

call get_pop_restart_filename(filename)

```
character(len=*), intent(out) :: filename
```

get_pop_restart_filename returns the filename containing the POP restart information. At this point the filename is **hardwired** to pop.r.nc, but may become more flexible in future versions. The filename may be derived from the restart_nml but is currently ignored.

filename	The name of the POP restart file.

6.187.5 Files

filename	purpose
pop_in	to read the POP namelists
pop.r.nc	provides grid dimensions and ‘valid_time’ of the model state
&grid_nml “horiz_grid_file”	contains the values of the horizontal grid
&grid_nml “vert_grid_file”	contains the number and values of the vertical levels
&grid_nml “topography_grid_file”	contains the indices of the wet/dry cells
pop_in.DART	to control the integration of the POP model advance

6.187.6 References

- none

6.187.7 Private components

N/A

6.188 PROGRAM model_to_dart for MPAS OCN

6.188.1 Overview

model_to_dart is the program that reads an MPAS OCN analysis file (nominally named mpas_restart.nc) and creates a DART state vector file (e.g. perfect_ics, filter_ics, ...). The MPAS analysis files have a **Time** UNLIMITED Dimension, which indicates there may (at some point) be more than one timestep in the file. The DART routines are currently designed to use the LAST timestep. If the Time dimension of length 3, we use the third timestep. A warning message is issued and indicates exactly the time being used.

input.nml&mpas_vars_nml defines the list of MPAS variables used to build the DART state vector. This namelist is more fully described in the *MPAS OCN* documentation. For example:

```
&mpas_vars_nml
   mpas_state_variables = 'temperature',  'QTY_TEMPERATURE',
                          'salinity',     'QTY_SALINITY',
```

(continues on next page)

(continued from previous page)

```
                          'rho',          'QTY_DENSITY',
                          'u',            'QTY_EDGE_NORMAL_SPEED',
                          'h',            'QTY_SEA_SURFACE_HEIGHT'
                          'tracer1',      'QTY_TRACER_CONCENTRATION'
   /
```

Conditions required for successful execution of `model_to_dart` are:

- a valid `input.nml` namelist file for DART which contains
- a MPAS OCN analysis file (nominally named `mpas_analysis.nc`).

Since this program is called repeatedly for every ensemble member, we have found it convenient to link the MPAS OCN analysis files to a static input filename (e.g. `mpas_analysis.nc`). The default DART filename is `dart_ics` - this may be moved or linked as necessary.

6.188.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_to_dart_nml
   model_to_dart_output_file = 'dart_ics'
   /
```

```
&model_nml
   model_analysis_filename  = 'mpas_analysis.nc'
   /

(partial namelist)
```

```
&mpas_vars_nml
   mpas_state_variables = '',
   mpas_state_bounds = '',
   /
```

The model_to_dart namelist includes:

Item	Type	Description
model_to_dart_output_file	character(len=128)	The name of the DART file containing the model state derived from the MPAS analysis file.

Two more namelists need to be mentioned. The model_nml namelist specifies the MPAS analysis file to be used as the source. The mpas_vars_nml namelist specifies the MPAS variables that will comprise the DART state vector.

For example:

```
&mpas_vars_nml
   mpas_state_variables = 'temperature',  'QTY_TEMPERATURE',
                          'salinity',     'QTY_SALINITY',
                          'rho',          'QTY_DENSITY',
                          'u',            'QTY_EDGE_NORMAL_SPEED',
                          'h',            'QTY_SEA_SURFACE_HEIGHT'
                          'tracer1',      'QTY_TRACER_CONCENTRATION'
   /
```

6.188.3 Modules used

```
assim_model_mod.f90
types_mod.f90
location_mod.f90
model_to_dart.f90
model_mod.f90
null_mpi_utilities_mod.f90
obs_kind_mod.f90
random_seq_mod.f90
time_manager_mod.f90
utilities_mod.f90
```

6.188.4 Files read

- MPAS analysis file; `mpas_analysis.nc`
- DART namelist file; input.nml

6.188.5 Files written

- DART initial conditions/restart file; e.g. `dart_ics`

6.188.6 References

none

6.189 PROGRAM `mpas_dart_obs_preprocess`

6.189.1 Overview

Program to preprocess observations, with specific knowledge of the MPAS grid.

This program can superob (average) aircraft and satellite wind obs if they are too dense, based on the given MPAS ATM grid. It will average all observations of the same type in each grid cell. The averaging grid can be different than the grid used for the assimilation run.

This program can read up to 10 additional obs_seq files and merge their data in with the basic obs_sequence file which is the main input.

This program can reject surface observations if the elevation encoded in the observation is too different from the mpas surface elevation.

This program can exclude observations above a specified height or pressure.

This program can exclude observations outside a given time window defined by the specified analysis time and a window width in hours.

This program can overwrite the incoming Data QC value with another.

6.189.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&mpas_obs_preproc_nml

  file_name_input          = 'obs_seq.old'
  file_name_output         = 'obs_seq.new'

  sonde_extra              = 'obs_seq.rawin'
  land_sfc_extra           = 'obs_seq.land_sfc'
  metar_extra              = 'obs_seq.metar'
  marine_sfc_extra         = 'obs_seq.marine'
  sat_wind_extra           = 'obs_seq.satwnd'
  profiler_extra           = 'obs_seq.profiler'
  gpsro_extra              = 'obs_seq.gpsro'
  acars_extra              = 'obs_seq.acars'
  gpspw_extra              = 'obs_seq.gpspw'
  trop_cyclone_extra       = 'obs_seq.tc'

  overwrite_obs_time       = .false.
```

(continues on next page)

(continued from previous page)

```
   windowing_obs_time        = .false.
   windowing_int_hour        = 1.5

   obs_boundary              = 0.0
   increase_bdy_error        = .false.
   maxobsfac                 = 2.5
   obsdistbdy                = 15.0

   sfc_elevation_check       = .false.
   sfc_elevation_tol         = 300.0
   obs_pressure_top          = 0.0
   obs_height_top            = 2.0e10

   include_sig_data          = .true.
   tc_sonde_radii            = -1.0
   superob_qc_threshold      = 4

   superob_aircraft          = .false.
   aircraft_horiz_int        = 36.0
   aircraft_pres_int         = 2500.0

   superob_sat_winds         = .false.
   sat_wind_horiz_int        = 100.0
   sat_wind_pres_int         = 2500.0

   overwrite_ncep_satwnd_qc = .false.
   overwrite_ncep_sfc_qc    = .false.

   max_num_obs               = 1000000
/
```

Item

Type

Description

Generic parameters:

file_name_input

character(len=129)

The input obs_seq file.

file_name_output

character(len=129)

The output obs_seq file.

sonde_extra, land_sfc_extra, metar_extra, marine_sfc_extra, marine_sfc_extra, sat_wind_extra, profiler_extra, gpsro_extra, acars_extra, gpspw_extra, trop_cyclone_extra

character(len=129)

The names of additional input obs_seq files, which if they exist, will be merged in with the obs from the `file_name_input` obs_seq file. If the files do not exist, they are silently ignored without error.

max_num_obs

integer

Must be larger than the total number of observations to be processed.

Parameters to reduce observation count:

sfc_elevation_check

logical

If true, check the height of surface observations against the surface height in the model. Observations further away than the specified tolerance will be excluded.

sfc_elevation_tol

real(r8)

If `sfc_elevation_check` is true, the maximum difference between the elevation of a surface observation and the model surface height, in meters. If the difference is larger than this value, the observation is excluded.

obs_pressure_top

real(r8)

Observations with a vertical coordinate in pressure which are located above this pressure level (i.e. the obs vertical value is smaller than the given pressure) will be excluded.

obs_height_top

real(r8)

Observations with a vertical coordinate in height which are located above this height value (i.e. the obs vertical value is larger than the given height) will be excluded.

Radio/Rawinsonde-specific parameters:

include_sig_data

logical

If true, include significant level data from radiosondes.

tc_sonde_radii

real(r8)

If greater than 0.0 remove any sonde observations closer than this distance in Kilometers to the center of a Tropical Cyclone.

Aircraft-specific parameters:

superob_aircraft

logical

If true, average all aircraft observations within the same MPAS grid cell, at the given vertical levels. The output obs will be only a single observation per cell, per vertical level.

aircraft_pres_int

real(r8)

If `superob_aircraft` is true, the vertical distance in pressure which defines a series of superob vertical bins.

superob_qc_threshold

integer

If `superob_aircraft` is true, the Quality Control threshold at which observations are ignored when doing superob averaging. The value specified here is the largest acceptable QC; values equal to or lower are kept, and values larger than this are rejected.

Satellite Wind-specific parameters:

superob_sat_winds

logical

If true, average all satellite wind observations within the same MPAS grid cell, at the given vertical levels. The output obs will be only a single observation per cell, per vertical level.

sat_wind_pres_int

real(r8)

If `superob_sat_winds` is true, the vertical distance in pressure which defines a series of superob vertical bins.

overwrite_ncep_satwnd_qc

logical

If true, replace the incoming Data QC value in satellite wind observations with 2.0.

Surface Observation-specific parameters:

overwrite_ncep_sfc_qc

logical

If true, replace the incoming Data QC value in surface observations with 2.0.

Parameters to select by time or alter observation time:

windowing_obs_time

logical

If true, exclude observations with a time outside the given window. The window is specified as a number of hours before and after the current analysis time.

windowing_int_hour

real(r8)

The window half-width, in hours. If 'windowing_obs_time' is .false. this value is ignored. If 'windowing_obs_time' is true, observations with a time further than this number of hours away from the analysis time will be excluded. To ensure disjoint subsets from a continueous sequence of observations, time values equal to the earliest time boundaries are discarded while time values equal to the latest time boundary are retained.

overwrite_obs_time

logical

If true, replace the incoming observation time with the analysis time. Not recommended.

6.189.3 Modules used

```
types_mod
obs_sequence_mod
utilities_mod
obs_kind_mod
time_manager_mod
model_mod
netcdf
```

6.189.4 Files

- Input namelist ; `input.nml`
- Input MPAS state netCDF file: `mpas_init.nc`
- Input obs_seq files (as specified in namelist)
- Output obs_seq file (as specified in namelist)

File formats

This utility can read one or more obs_seq files and combine them while doing the rest of the processing. It uses the standard DART observation sequence file format. It uses the grid information from an MPAS file to define the bins for combining nearby aircraft and satellite wind observations.

6.189.5 References

- Developed by Soyoung Ha, based on the WRF observation preprocessor contributed by Ryan Torn.

6.190 CESM+DART setup

6.190.1 Cesm+dart setup overview

If you found your way to this file without reading more basic DART help files, please read those first. $DART/README is a good place to find pointers to those files. This document gives specific help in setting up a CESM+DART assimilation for the first time. Unless you just came from there, also see the ../{your_model(s)}/model_mod.html documentation about the code-level interfaces and namelist values.

CESM context

Most other models are either called by DART (low order models), or are run by DART via a shell script command (e.g. WRF). In contrast, CESM runs its forecast, and then calls DART to do the assimilation. The result is that assimilation set-up scripts for CESM components focus on modifying the set-up and build of CESM to accommodate DART's needs, such as multi-instance forecasts, stopping at the assimilation times to run filter, and restarting with the updated model state. The amount of modification of CESM depends on which version of CESM is being used. Later versions require fewer changes because CESM has accommodated more of DART's needs with each CESM release. This release of DART focuses on selected CESM versions from 1_2_1 onward, through CESM2 (June, 2017) and versions to be added later. Using this DART with other CESM versions will quite possibly fail.

Since the ability to use DART has not been completely integrated into CESM testing, it is necessary to use some CESM fortran subroutines which have been modified for use with DART. These must be provided to CESM through the SourceMods mechanism. SourceMods for selected versions of CESM are available from the DART website. They can often be used as a template for making a SourceMods for a different CESM version. If you have other CESM modifications, they must be merged with the DART modifications.

CESM2

CESM2 (expected release May, 2017) has several helpful improvements, from DART's perspective.

- Reduced number of subroutines in DART's SourceMods.
- "Multi-instance" capability enables the ensemble forecasts DART needs.
- Cycling capability, enabling multiple assimilation cycles in a single job, which reduces the frequency of waiting in the queue.
- Removal of the short term archiver from the run script so that the MPI run doesn't need to idle while the single task archiver runs. This significantly reduces the core hours required.
- CESM's translation of the short term archiver to python, and control of it to an xml file ($caseroot/env_archive.xml), so that DART modifications to the short term archiver are more straight-forward.
- The creation of a new component class, "External Statistical Processing" ("esp"), of which DART is the first instance, integrates DART more fully into the CESM development, testing, and running environment. This is the same as the atm class, which has CAM as an instance. This will help make DART available in the most recent tagged CESM versions which have the most recent CESM component versions.

These have been exploited most fully in the CAM interfaces to DART, since the other components' interfaces still use older CESMs. The cam-fv/shell_scripts can be used as a template for updating other models' scripting. The multi-cycling capability, with the short- and long-term archivers running as separate jobs at the end, results in assimilation jobs which rapidly fill the scratch space. Cam-fv's and pop's assimilate.csh scripts have code to remove older and unneeded CESM restart file sets during the run. All of DART's output and user selected, restart file sets are preserved.

DART's manhattan release includes the change to hard-wired input and output filenames in filter. Cam-fv's
assimilate.csh renames these files into the CESM file format:

$case.$component{_$instance}.$filetype.$date.nc.

DART's hard-wired names are used as new filetypes, just like CESM's existing "r", "h0", etc. For example,
preassim_mean.nc from a CAM assimilation named Test0 will be renamed

Test0.cam.preassim_mean.2013-03-14-21600.nc

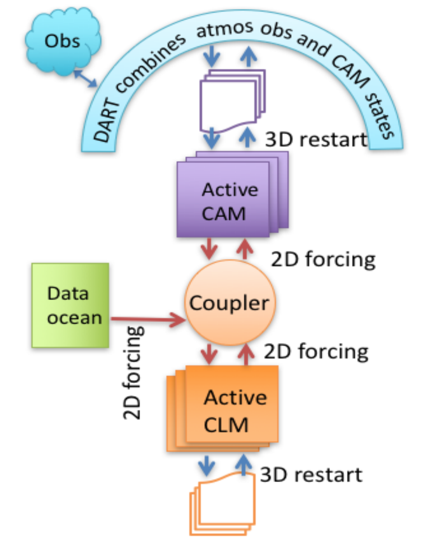
The obs_seq files remain an exception to this renaming, since they are not in NetCDF format (yet).

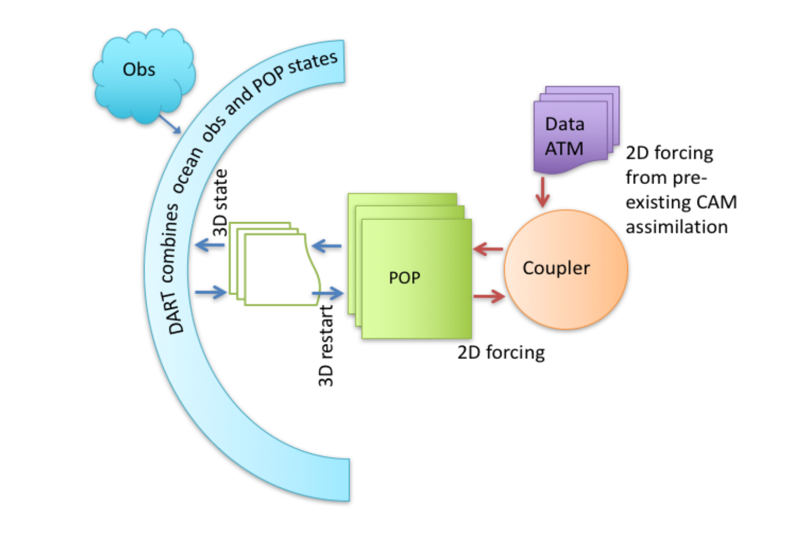
6.190.2 CESM component combinations

CESM can be configured with many combinations of its components (CAM, CLM, POP, CICE, . . .) some of which may be 'data' components, which merely read in data from some external source and pass it to the other, active, components to use. The components influence each other only through the coupler. There are several modes of assimilating observations in this context.

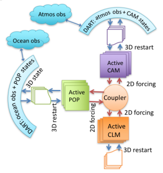
Single-component assimilation

The first, and simplest, consists of assimilating relevant observations into one active component. Most/all of the rest of the components are 'data'. For example, observations of the oceans can be assimilated into the POP model state, while the atmospheric forcing of the ocean comes from CAM re-analysis files, and is not changed by the observations. A variation of this is used by CAM assimilations. A CAM forecast usually uses an active land component (CLM) as well as an active atmospheric component. Atmospheric observations are assimilated only into the CAM state, while the land state is modified only through its interactions with CAM through the coupler. Each of these assimilations is handled by one of $DART/models/{cam-fv, pop, clm, . . . } If you want to use an unusual combination of active and data components, you may need to (work with us to) modify the setup scripts.



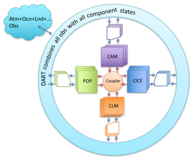

Obs
DART combines ocean obs and POP states
3D state
3D restart
POP
Data ATM
2D forcing from pre-existing CAM assimilation
Coupler
2D forcing

Multi-component assimilation (aka "weakly coupled")



It's also possible to assimilate observations into multiple active components, but restricting the impact of observations to only "their own" component. So in a "coupled" CESM with active CAM and POP, atmospheric observations change only the CAM model state while oceanic observations change only the POP model state. This mode uses multiple DART models; cam-fv and pop in this example to make a filter for each model.

Cross-component assimilation (aka "strongly coupled")

	Work is underway to enable the assimilation of all observations into multiple active CESM components. So observations of the atmosphere would directly change the POP state variables and vice versa. Some unresolved issues include defining the "distance" between an observation in the atmosphere and a grid point in the ocean (for localization), and how frequently to assimilate in CAM versus POP. This mode will use code in this models/CESM directory.

CAM-FV

6.190.3 $dart/models/{cesm components} organization

SCRIPT	NOTES
$DART/models/**cam-fv**/	A 'model' for each CAM dynamical core (see note below this outline)
model_mod.*	The fortran interface between CAM-FV and DART
shell_scripts/	
no_assimilate.csh,...	**In**dependent_of_cesm_version
cesm**1_5**/	
setup_hybrid,...	**De**pendent on CESM version
cesm**2_0**/	
setup_hybrid,...	**De**pendent on CESM version
$DART/models/**pop**/	A 'model' for each ocean model (MOM may be interfaced next)
model_mod.*	The fortran interface between CAM-FV and DART
shell_scripts/	
no_assimilate.csh,...	**In**dependent_of_cesm_version
cesm**1_5**/	
setup_hybrid,...	**De**pendent on CESM version
cesm**2_0**/	
setup_hybrid,...	**De**pendent on CESM version
...	

```
For each CAM dynamical core "model", e.g. "cam-fv",  the scripts  in cesm#_# will␣
↪handle:
   all CAM variants + vertical resolutions (*dy-core is NOT part of this.*):
       CAM5.5, CAM6, ...
       WACCM4, WACCM6, WACCM-X...
       CAM-Chem,
       ...
   all horizontal resolutions of its dy-core:
       1.9x2.5, 0.9x1.25, ..., for cam-fv
       ne30np4, ne0_CONUS,..., for cam-se
```

6.190.4 Assimilation set-up procedure

Here is a list of steps to set up an assimilation from scratch, except that it assumes you have downloaded DART and learned how to use it with low order models. Some of the steps can be skipped if you have a suitable replacement, as noted.

1. Decide which component(s) you want to use as the assimilating model(s). (The rest of this example assumes that you're building a cam-fv assimilation.) Look in models/cam-fv/shell_scripts to see which CESM versions are supported.
2. CESM: locate that version on your system, or check it out from http://www.cesm.ucar.edu/models/current.html
3. Choose a start date for your assimilation. Choosing/creating the initial ensemble is a complicated issue.
    - It's simpler for CAM assimilations. If you don't have an initial state and/or ensemble for this date, build a single instance of CESM (Fxxxx compset for cam-fv) and run it from the default Jan 1 start date until 2-4 weeks before your start date. Be sure to set the cam namelist variable inithist = 'ENDOFRUN' during the last stage, so that CAM will write an "initial" file, which DART needs.
    - For ocean and land assimilations,which use an ensemble of data atmospheres, creating usable initial ensemble is a different process.
4. Put the entire cam-fv restart file set (including the initial file) where it won't be scrubbed before you want to use it. Create a pseudo-ensemble by linking files with instance numbers in them to the restart file set (which has no instance number) using CESM/shell_scripts/link_ens_to_single.csh
5. Choose the options in $dart/mkmf/mkmf.template that are best for your assimilation. These will not affect the CESM build, only filter.
6. In models/cam-fv/work/input.nml, be sure to include all of your required obs_def_${platform}_mod.f90 file names in preprocess_nml:input_files. It's also useful to modify the rest of input.nml to make it do what you want for the first assimilation cycle. This input.nml will be copied to the $case_root directory and used by assimilate.csh.
7. Build the DART executables using quickbuild.csh.
8. Follow the directions in models/cam-fv/shell_scripts/cesm#_#/setup_hybrid to set up the assimilation and build of CESM. We recommend a tiny ensemble to start with, to more quickly test whether everything is in order.
9. After convincing yourself that the CESM+DART framework is working with no_assimilate.csh, activate the assimilation by changing CESM's env_run.xml:DATA_ASSIMILATION_SCRIPT to use assimilate.csh.
10. After the first forecast+assimilation cycle finishes correctly, change the input.nml, env_run.xml and env_batch.xml to do additional cycle(s) without the perturbation of the initial state, and with using the just created restart files. You may also want to turn on the st_archive program. Instructions are in setup_hybrid and cam-fv/work/input.nml.
11. Finally, build a new case with the full ensemble, activate the assimilate.csh script and repeat the steps in step 10.

6.190.5 Output directory

CESM's short term archiver (st_archive) is controlled by its `env_archive.xml`. DART's setup scripts modify that file to archive DART output along with CESM's. (See the *RMA notes* for a description of DART's output). DART's output is archived in `$arch_dir/dart/{hist,rest,logs,...}`, where arch_dir is defined in `setup_{hybrid,advanced}`, `hist` contains all of the state space and observation space output, and `rest` contains the inflation restart files.

`Central directory`	User	Location of scripts and pass-through point for files during execution. Typically named according defining characteristics of a *set* of experiments; resolution, model, obs being assimilated, unique model state variables, etc.

–>

The cam-XX assimilate.csh scripts also make a copy of the obs_seq.final files in a scratch space ($scratch/$case/Obs_seqs) which won't be removed by CESM's long term archiver, if that is run.

6.190.6 Helpful hints

6.190.7 Space requirements

Space requirements (Gb per ensemble member) for several CAM resolutions.

There are, no doubt, things missing from these lists, so don't struggle too long before contacting dart'at'ucar.edu.

Useful terms found in this web page.

6.191 MODULE model_mod

6.191.1 Overview

Every model that is DART compliant must provide an interface as documented here. The file `models/template/model_mod.f90` provides the fortran interfaces for a minimal implementation meeting these requirements. When adding a new model to DART you can either start by modifying a `model_mod.f90` file from a similar model already in DART or start with the template file. Either way, the supplied interface must match these descriptions exactly; no details of the underlying model can impact the interface.

Several of the routines listed below are allowed to be a NULL INTERFACE. This means the subroutine or function name must exist in this file, but it is ok if it contains no executable code.

A few of the routines listed below are allowed to be a PASS-THROUGH INTERFACE. This means the subroutine or function name can be listed on the 'use' line from the location_mod, and no subroutine or function with that name is supplied in this file. Alternatively, this file can provide an implementation which calls the underlying routines from the location_mod and then alters or augments the results based on model-specific requirements.

The system comes with several types of location modules for computing distances appropriately. Two of the ones most commonly used are for data in a 1D system and for data in a 3D spherical coordinate system. Make the selection by listing the appropriate choice from location/*/location_mod.f90 in the corresponding path_names_* file at compilation time.

6.191.2 Namelist

This namelist is read from the file input.nml. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
 /
```

Models are free to include a model namelist which can be read when static_init_model is called. A good example can be found in the lorenz_96 model_mod.f90.

6.191.3 Other modules used

```
types_mod
time_manager_mod
location_mod (multiple choices here)
utilities_mod
POSSIBLY MANY OTHERS DEPENDING ON MODEL DETAILS
```

6.191.4 Public interfaces

use model_mod, only :	get_model_size
	adv_1step
	get_state_meta_data
	model_interpolate
	shortest_time_between_assimilations
	static_init_model
	init_time
	init_conditions
	nc_write_model_atts
	nc_write_model_vars
	pert_model_copies
	get_close_obs
	get_close_state
	convert_vertical_obs
	convert_vertical_state
	read_model_time
	write_model_time
	end_model

A namelist interface `&model_nml` may be defined by the module, in which case it will be read from file `input.nml`. The details of the namelist are always model-specific (there are no generic namelist values).

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

model_size = get_model_size( )

```
integer(i8) :: get_model_size
```

Returns the length of the model state vector. Required.

model_size	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

Does a single timestep advance of the model. The input value of the vector x is the starting condition and x must be updated to reflect the changed state after a timestep. The time argument is intent in and is used for models that need to know the date/time to compute a timestep, for instance for radiation computations. This interface is only called if the namelist parameter async is set to 0 in perfect_model_obs or filter or if the program integrate_model is to be used to advance the model state as a separate executable. If one of these options is not going to be used (the model will *only* be advanced as a separate model-specific executable), this can be a NULL INTERFACE. (The subroutine name must still exist, but it can contain no code and it will not be called.)

x	State vector of length model_size.
time	Current time of the model state.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,             intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,   intent(out) ::  var_type
```

Given an integer index into the state vector, returns the associated location. An optional argument returns the generic quantity of this item, e.g. QTY_TEMPERATURE, QTY_DENSITY, QTY_SALINITY, QTY_U_WIND_COMPONENT. This interface is required to be functional for all applications.

index_in	Index of state vector element about which information is requested.
location	The location of state variable element.
var_type	The generic quantity of the state variable element.

call model_interpolate(state_handle, ens_size, location, obs_quantity, expected_obs, istatus)

```
type(ensemble_type),    intent(in)  :: state_handle
integer,                intent(in)  :: ens_size
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: obs_quantity
real(r8),               intent(out) :: expected_obs(ens_size)
integer,                intent(out) :: istatus(ens_size)
```

Given a handle containing information for a state vector, an ensemble size, a location, and a model state variable quantity interpolates the state variable field to that location and returns an ensemble-sized array of values in `expected_obs(:)`. The `istatus(:)` array should be 0 for successful ensemble members and a positive value for failures. The `obs_quantity` variable is one of the quantity (QTY) parameters defined in the *MODULE obs_kind_mod* file and defines the quantity to interpolate. In low-order models that have no notion of kinds of variables this argument may be ignored. For applications in which only perfect model experiments with identity observations (i.e. only the value of a particular state variable is observed), this can be a NULL INTERFACE. Otherwise it is required (which is the most common case).

`state_handle`	The handle to the state structure containing information about the state vector about which information is requested.
`ens_size`	The ensemble size.
`location`	Location to which to interpolate.
`obs_quantity`	Quantity of state field to be interpolated.
`expected_obs`	The interpolated values from the model.
`istatus`	Integer values return 0 for success. Other positive values can be defined for various failures.

var = shortest_time_between_assimilations()

```
type(time_type) :: shortest_time_between_assimilations
```

Returns the smallest increment in time that the model is capable of advancing the state in a given implementation. The actual value may be set by the model_mod namelist (depends on the model). This interface is required for all applications.

`var`	Smallest advance time of the model.

call static_init_model()

Called to do one time initialization of the model. As examples, might define information about the model size or model timestep, read in grid information, read a namelist, set options, etc. In models that require pre-computed static data, for instance spherical harmonic weights, these would also be computed here. Can be a NULL INTERFACE for the simplest models.

call init_time(time)

```
type(time_type), intent(out) :: time
```

Companion interface to init_conditions. Returns a time that is somehow appropriate for starting up a long integration of the model. At present, this is only used if the `perfect_model_obs` namelist parameter `read_input_state_from_file = .false.` If this option should not be used in `perfect_model_obs`, calling this routine should issue a fatal error.

time	Initial model time.

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

Returns a model state vector, x, that is some sort of appropriate initial condition for starting up a long integration of the model. At present, this is only used if the `perfect_model_obs` namelist parameter `read_input_state_from_file = .false.` If this option should not be used in `perfect_model_obs`, calling this routine should issue a fatal error.

x	Initial conditions for state vector.

call nc_write_model_atts(ncFileID, domain_id)

```
integer, intent(in) :: ncFileID
integer, intent(in) :: domain_id
```

This routine writes the model-specific attributes to netCDF files that DART creates. This includes coordinate variables and any metadata, but NOT the actual model state vector. `models/template/model_mod.f90` contains code that can be used for any model as-is.

The typical sequence for adding new dimensions, variables, attributes:

```
NF90_OPEN              ! open existing netCDF dataset
   NF90_redef          ! put into define mode
   NF90_def_dim        ! define additional dimensions (if any)
   NF90_def_var        ! define variables: from name, kind, and dims
   NF90_put_att        ! assign attribute values
NF90_ENDDEF            ! end definitions: leave define mode
   NF90_put_var        ! provide values for variable
NF90_CLOSE             ! close: save updated netCDF dataset
```

ncFileID	Integer file descriptor to previously-opened netCDF file.
domain_id	integer describing the domain (which can be a nesting level, a component model …) Models with nested grids are decomposed into 'domains' in DART. The concept is extended to refer to 'coupled' models where one model component may be the atmosphere, another component may be the ocean, or land, or ionosphere … these would be referenced as different domains.

call nc_write_model_vars(ncFileID, domain_id, state_ens_handle [, memberindex] [, timeindex])

```
integer,             intent(in) :: ncFileID
integer,             intent(in) :: domain_id
type(ensemble_type), intent(in) :: state_ens_handle
integer, optional,   intent(in) :: memberindex
integer, optional,   intent(in) :: timeindex
```

This routine may be used to write the model-specific state vector (data) to a netCDF file. Only used if `model_mod_writes_state_variables = .true.`
Typical sequence for adding new dimensions,variables,attributes:

```
NF90_OPEN              ! open existing netCDF dataset
   NF90_redef          ! put into define mode
   NF90_def_dim        ! define additional dimensions (if any)
   NF90_def_var        ! define variables: from name, kind, and dims
   NF90_put_att        ! assign attribute values
NF90_ENDDEF            ! end definitions: leave define mode
   NF90_put_var        ! provide values for variable
NF90_CLOSE             ! close: save updated netCDF dataset
```

`ncFileID`	file descriptor to previously-opened netCDF file.
`domain_id`	integer describing the domain (which can be a nesting level, a component model . . .)
`state_ens_handle`	The handle to the state structure containing information about the state vector about which information is requested.
`memberindex`	Integer index of ensemble member to be written.
`timeindex`	The timestep counter for the given state.

call pert_model_copies(state_ens_handle, ens_size, pert_amp, interf_provided)

```
type(ensemble_type), intent(inout) :: state_ens_handle
integer,             intent(in)    :: ens_size
real(r8),            intent(in)    :: pert_amp
logical,             intent(out)   :: interf_provided
```

Given an ensemble handle, the ensemble size, and a perturbation amplitude; perturb the ensemble. Used to generate initial conditions for spinning up ensembles. If the `model_mod` does not want to do this, instead allowing the default algorithms in `filter` to take effect, `interf_provided =&nbps;.false.` and the routine can be trivial. Otherwise, `interf_provided` must be returned as `.true.`

`state_ens_handle`	The handle containing an ensemble of state vectors to be perturbed.
`ens_size`	The number of ensemble members to perturb.
`pert_amp`	the amplitude of the perturbations. The interpretation is based on the model-specific implementation.
`interf_provided`	Returns false if model_mod cannot do this, else true.

call get_close_obs(gc, base_loc, base_type, locs, loc_qtys, loc_types, num_close, close_ind [, dist] [, state_handle)

```
type(get_close_type),          intent(in)  :: gc
type(location_type),           intent(in)  :: base_loc
integer,                       intent(in)  :: base_type
type(location_type),           intent(in)  :: locs(:)
integer,                       intent(in)  :: loc_qtys(:)
integer,                       intent(in)  :: loc_types(:)
integer,                       intent(out) :: num_close
integer,                       intent(out) :: close_ind(:)
real(r8),            optional, intent(out) :: dist(:)
type(ensemble_type), optional, intent(in)  :: state_handle
```

Given a location and quantity, compute the distances to all other locations in the `obs` list. The return values are the number of items which are within maxdist of the base, the index numbers in the original obs list, and optionally the distances. The `gc` contains precomputed information to speed the computations.

In general this is a PASS-THROUGH ROUTINE. It is listed on the use line for the locations_mod, and in the public list for this module, but has no subroutine declaration and no other code in this module:

```
use location_mod, only: get_close_obs

public :: get_close_obs
```

However, if the model needs to alter the values or wants to supply an alternative implementation it can intercept the call like so:

```
use location_mod, only: &
        lm_get_close_obs => get_close_obs

public :: get_close_obs
```

In this case a local `get_close_obs()` routine must be supplied. To call the original code in the location module use:

```
call lm_get_close_obs(gc, base_loc, ...)
```

This subroutine will be called after `get_close_maxdist_init` and `get_close_obs_init`.

In most cases the PASS-THROUGH ROUTINE will be used, but some models need to alter the actual distances depending on the observation or state vector kind, or based on the observation or state vector location. It is reasonable in this case to leave `get_close_maxdist_init()` and `get_close_obs_init()` as pass-through routines and intercept only `get_close_obs()`. The local `get_close_obs()` can first call the location mod routine and let it return a list of values, and then inspect the list and alter or remove any entries as needed. See the CAM and WRF model_mod files for examples of this use.

`gc`	The get_close_type which stores precomputed information about the locations to speed up searching
`base_loc`	Reference location. The distances will be computed between this location and every other location in the obs list
`base_type`	The DART quantity at the `base_loc`
`locs(:)`	Compute the distance between the `base_loc` and each of the locations in this list
`loc_qtys(:)`	The corresponding quantity of each item in the `locs` list
`loc_types(:)`	The corresponding type of each item in the `locs` list. This is not available in the default implementation but may be used in custom implementations.
`num_close`	The number of items from the `locs` list which are within maxdist of the base location
`close_ind(:)`	The list of index numbers from the `locs` list which are within maxdist of the base location
`dist(:)`	If present, return the distance between each entry in the close_ind list and the base location. If not present, all items in the obs list which are closer than maxdist will be added to the list but the overhead of computing the exact distances will be skipped.
`state_handle`	The handle to the state structure containing information about the state vector about which information is requested.

call get_close_state(gc, base_loc, base_type, state_loc, state_qtys, state_indx, num_close, close_ind [, dist, state_handle])

```
type(get_close_type),           intent(in)    :: gc
type(location_type),            intent(inout) :: base_loc
integer,                        intent(in)    :: base_type
type(location_type),            intent(inout) :: state_loc(:)
integer,                        intent(in)    :: state_qtys(:)
integer(i8),                    intent(in)    :: state_indx(:)
integer,                        intent(out)   :: num_close
integer,                        intent(out)   :: close_ind(:)
real(r8),             optional, intent(out)   :: dist(:)
type(ensemble_type), optional, intent(in)    :: state_handle
```

Given a location and quantity, compute the distances to all other locations in the `state_loc` list. The return values are the number of items which are within maxdist of the base, the index numbers in the original state_loc list, and optionally the distances. The `gc` contains precomputed information to speed the computations.

In general this is a PASS-THROUGH ROUTINE. It is listed on the use line for the locations_mod, and in the public list for this module, but has no subroutine declaration and no other code in this module:

```
use location_mod, only: get_close_state

public :: get_close_state
```

However, if the model needs to alter the values or wants to supply an alternative implementation it can intercept the call like so:

```
use location_mod, only: &
        lm_get_close_state => get_close_state

public :: get_close_state
```

In this case a local `get_close_state()` routine must be supplied. To call the original code in the location module use:

```
call loc_get_close_state(gc, base_loc, ...)
```

This subroutine will be called after `get_close_maxdist_init` and `get_close_state_init`.

In most cases the PASS-THROUGH ROUTINE will be used, but some models need to alter the actual distances depending on the observation or state vector kind, or based on the observation or state vector location. It is reasonable in this case to leave `get_close_maxdist_init()` and `get_close_state_init()` as pass-through routines and intercept only `get_close_state()`. The local `get_close_state()` can first call the location mod routine and let it return a list of values, and then inspect the list and alter or remove any entries as needed. See the CAM and WRF model_mod files for examples of this use.

`gc`	The get_close_type which stores precomputed information about the locations to speed up searching
`base_loc`	Reference location. The distances will be computed between this location and every other location in the list
`base_type`	The DART quantity at the `base_loc`
`state_loc(:)`	Compute the distance between the `base_loc` and each of the locations in this list
`state_qtys(:)`	The corresponding quantity of each item in the `state_loc` list
`state_indx(:)`	The corresponding DART index of each item in the `state_loc` list. This is not available in the default implementation but may be used in custom implementations.
`num_close`	The number of items from the `state_loc` list which are within maxdist of the base location
`close_ind(:)`	The list of index numbers from the `state_loc` list which are within maxdist of the base location
`dist(:)`	If present, return the distance between each entry in the `close_ind` list and the base location. If not present, all items in the `state_loc` list which are closer than maxdist will be added to the list but the overhead of computing the exact distances will be skipped.
`state_handle`	The handle to the state structure containing information about the state vector about which information is requested.

call convert_vertical_obs(state_handle, num, locs, loc_qtys, loc_types, which_vert, status)

```
type(ensemble_type), intent(in)  :: state_handle
integer,             intent(in)  :: num
type(location_type), intent(in)  :: locs(:)
integer,             intent(in)  :: loc_qtys(:)
integer,             intent(in)  :: loc_types(:)
integer,             intent(in)  :: which_vert
integer,             intent(out) :: status(:)
```

Converts the observations to the desired vertical localization coordinate system. Some models (toy models with no 'real' observations) will not need this. Most (real) models have observations in one or more coordinate systems (pressure, height) and the model is generally represented in only one coordinate system. To be able to interpolate the model state to the observation location, or to compute the true distance between the state and the observation, it is necessary to convert everything to a single coodinate system.

state_handle	The handle to the state.
num	the number of observation locations
locs	the array of observation locations
loc_qtys	the array of observation quantities.
loc_types	the array of observation types.
which_vert	the desired vertical coordinate system. There is a table in the location_mod.f90 that relates integers to vertical coordinate systems.
status	Success or failure of the vertical conversion. If istatus = 0, the conversion was a success. Any other value is a failure.

call convert_vertical_state(state_handle, num, locs, loc_qtys, loc_types, which_vert, status)

```
type(ensemble_type), intent(in)  :: state_handle
integer,             intent(in)  :: num
type(location_type), intent(in)  :: locs(:)
integer,             intent(in)  :: loc_qtys(:)
integer,             intent(in)  :: loc_types(:)
integer,             intent(in)  :: which_vert
integer,             intent(out) :: status(:)
```

Converts the state to the desired vertical localization coordinate system. Some models (toy models with no 'real' observations) will not need this. To compute the true distance between the state and the observation, it is necessary to convert everything to a single coodinate system.

state_handle	The handle to the state.
num	the number of state locations
locs	the array of state locations
loc_qtys	the array of state quantities.
loc_types	the array of state types.
which_vert	the desired vertical coordinate system. There is a table in the location_mod.f90 that relates integers to vertical coordinate systems.
status	Success or failure of the vertical conversion. If istatus = 0, the conversion was a success. Any other value is a failure.

model_time = read_model_time(filename)

```
character(len=*), intent(in) :: filename
type(time_type)              :: model_time
```

Reads the valid time of the model state in a netCDF file. There is a default routine in assimilation_code/modules/io/dart_time_io_mod.f90 that can be used as a pass-through. That routine will read the **last** timestep of a 'time' variable - which is the same strategy used for reading netCDF files that have multiple timesteps in them. If your model has some other representation of time (i.e. it does not use a netCDF variable named 'time') - you will have to write this routine.

ncid	handle to an open netCDF file
dart_time	The current time of the model state.

call write_model_time(ncid, dart_time)

```
integer,          intent(in) :: ncid
type(time_type),  intent(in) :: dart_time
```

Writes the assimilation time to a netCDF file. There is a default routine in `assimilation_code/modules/io/dart_time_io_mod.f90` that can be used as a pass-through. If your model has some other representation of time (i.e. it does not use a netCDF variable named 'time') - you will have to write this routine.

ncid	handle to an open netCDF file
dart_time	The current time of the model state.

call end_model()

Does any shutdown and clean-up needed for model. Can be a NULL INTERFACE if the model has no need to clean up storage, etc.

6.191.5 Files

- Models are free to read and write files as they see fit.

6.191.6 References

1. none

6.191.7 Private components

N/A

6.192 module atmos_radon_mod

6.192.1 Overview

This code allows the implementation of an extremely simplified radon tracer in the FMS framework.
It should be taken as the implementation of a very simple tracer which bears some characteristics of radon.

This module presents an implementation of a tracer. It should be taken as representing radon only in a rudimentary manner.

6.192.2 Other modules used

```
fms_mod
    time_manager_mod
    diag_manager_mod
  tracer_manager_mod
   field_manager_mod
tracer_utilities_mod
```

6.192.3 Public interface

```
use atmos_radon_mod [, only:  atmos_radon_sourcesink,
                              atmos_radon_init,
                              atmos_radon_end ]
```

atmos_radon_sourcesink: The routine that calculate the sources and sinks of radon.

atmos_radon_init: The constructor routine for the radon module.

atmos_radon_end: The destructor routine for the radon module.

6.192.4 Public data

None.

6.192.5 Public routines

a. **Atmos_radon_sourcesink**

```
call atmos_radon_sourcesink (lon, lat, land, pwt, radon, radon_dt, Time, kbot)
```

DESCRIPTION This is a very rudimentary implementation of radon. It is assumed that the Rn222 flux is 3.69e-21 kg/m*m/sec over land for latitudes < 60N Between 60N and 70N the source = source * .5 Rn222 has a half-life time of 3.83 days, which corresponds to an e-folding time of 5.52 days.

INPUT

lon	Longitude of the centre of the model gridcells [real, dimension(:,:)]
lat	Latitude of the centre of the model gridcells [real, dimension(:,:)]
land	Land/sea mask. [real, dimension(:,:)]
pwt	The pressure weighting array. = dP/grav [real, dimension(:,:,:)]
radon	The array of the radon mixing ratio. [real, dimension(:,:,:)]
Time	Model time. [type(time_type)]
kbot	Integer array describing which model layer intercepts the surface. [integer, optional, dimension(:,:)]

OUTPUT

radon_dt	The array of the tendency of the radon mixing ratio. [real, dimension(:,:,:)]

b. ### Atmos_radon_init

```
call atmos_radon_init
```

DESCRIPTION A routine to initialize the radon module.

INPUT

mask	optional mask (0. or 1.) that designates which grid points are above (=1.) or below (=0.) the ground dimensioned as (nlon,nlat,nlev). [real, optional, dimension(:,:,:)]
Time	Model time. [type(time_type)]
axes	The axes relating to the tracer array dimensioned as (nlon, nlat, nlev, ntime) [integer, dimension(4)]

INPUT/OUTPUT

r	Tracer fields dimensioned as (nlon,nlat,nlev,ntrace). [real, dimension(:,:,:,:)]

c. ### Atmos_radon_end

```
call atmos_radon_end
```

DESCRIPTION This subroutine writes the version name to logfile and exits.

6.192.6 Data sets

None.

6.192.7 Error messages

None.

top

6.193 module atmos_sulfur_hex_mod

6.193.1 Overview

This code allows the implementation of sulfur hexafluoride tracer in the FMS framework.

6.193.2 Other modules used

```
fms_mod
    time_manager_mod
    diag_manager_mod
  tracer_manager_mod
   field_manager_mod
tracer_utilities_mod
       constants_mod
```

6.193.3 Public interface

```
use atmos_sulfur_hex_mod [, only:  atmos_sf6_sourcesink,
                                   atmos_sulfur_hex_init,
                                   sulfur_hex_end ]
```

atmos_sf6_sourcesink: A routine to calculate the sources and sinks of sulfur hexafluoride.

atmos_sulfur_hex_init: The constructor routine for the sulfur hexafluoride module.

sulfur_hex_end: The destructor routine for the sulfur hexafluoride module.

6.193.4 Public data

None.

6.193.5 Public routines

a. **Atmos_sf6_sourcesink**

```
call atmos_sf6_sourcesink (lon, lat, land, pwt, sf6, sf6_dt, Time, is, ie, js, je,
↪ kbot)
```

DESCRIPTION A routine to calculate the sources and sinks of sulfur hexafluoride.

INPUT

`lon`	Longitude of the centre of the model gridcells. [real, dimension(:,:)]
`lat`	Latitude of the centre of the model gridcells. [real, dimension(:,:)]
`land`	Land/sea mask. [real, dimension(:,:)]
`pwt`	The pressure weighting array. = dP/grav [real, dimension(:,:,:)]
`sf6`	The array of the sulfur hexafluoride mixing ratio. [real, dimension(:,:,:)]
`Time`	Model time. [type(time_type)]
`is, ie, js, je`	Local domain boundaries. [integer]
`kbot`	Integer array describing which model layer intercepts the surface. [integer, optional, dimension(:,:)]

OUTPUT

`sf6_dt`	The array of the tendency of the sulfur hexafluoride mixing ratio. [real, dimension(:,:,:)]

b. **Atmos_sulfur_hex_init**

```
call atmos_sulfur_hex_init (lonb, latb, r, axes, Time, mask)
```

DESCRIPTION A routine to initialize the sulfur hexafluoride module.

INPUT

`lonb`	The longitudes for the local domain. [real, dimension(:)]
`latb`	The latitudes for the local domain. [real, dimension(:)]
`mask`	optional mask (0. or 1.) that designates which grid points are above (=1.) or below (=0.) the ground dimensioned as (nlon,nlat,nlev). [real, optional, dimension(:,:,:)]
`Time`	Model time. [type(time_type)]
`axes`	The axes relating to the tracer array dimensioned as (nlon, nlat, nlev, ntime) [integer, dimension(4)]

INPUT/OUTPUT

`r`	Tracer fields dimensioned as (nlon,nlat,nlev,ntrace). [real, dimension(:,:,:,:)]

c. **Sulfur_hex_end**

```
call sulfur_hex_end
```

DESCRIPTION This subroutine is the exit routine for the sulfur hexafluoride module.

6.193.6 Data sets

Sulfur hexaflouride emissions Monthly.emissions contains the estimated global emission rate of SF6 in Gg/yr for 62 months between December 1988 and January 1994, inclusive. These are based on the annual estimates of Levin and Hesshaimer (submitted), and have been linearly interpolated to monthly values. The last half of 1993 has been extrapolated using the trend for the previous 12 months. The dataset can be obtained from the contact person above.

6.193.7 Error messages

None.

6.193.8 References

1. Levin, I. and V. Hessahimer: Refining of atmospheric transport model entries by the globally observed passive tracer distributions of 85Krypton and Sulfur Hexafluoride (SF6). Submitted to the Journal of Geophysical Research.

6.193.9 Compiler specifics

None.

6.193.10 Precompiler options

None.

6.193.11 Loader options

None.

6.193.12 Test PROGRAM

None.

6.193.13 Notes

None.

6.194 module atmos_tracer_driver_mod

6.194.1 Overview

This code allows the user to easily add tracers to the FMS framework.

This code allows a user to easily implement tracer code in the FMS framework. The tracer and tracer tendency arrays are supplied along with longtitude, latitude, wind, temperature, and pressure information which allows a user to implement sources and sinks of the tracer which depend on these parameters. In the following example, radon being implemented in the atmosphere will be used as an example of how to implement a tracer in the FMS framework. Within the global scope of tracer_driver_mod a use statement should be inserted for each tracer to be added.

```
use radon_mod, only : radon_sourcesink, radon_init, radon_end
```

An integer parameter, which will be used as an identifier for the tracer, should be assigned.

```
integer :: nradon
```

Within tracer_driver_init a call to the tracer manager is needed in order to identify which tracer it has set the tracer as.

```
nradon = get_tracer_index(MODEL_ATMOS,'radon')
```

Here MODEL_ATMOS is a parameter defined in field_manager. 'radon' is the name of the tracer within the field_table. If the tracer exists then the integer returned will be positive and it can be used to call the initialization routines for the individual tracers.

```
if (nradon > 0) then
     call radon_init(Argument list)
endif
```

Within tracer_driver the user can also use the identifier to surround calls to the source-sink routines for the tracer of interest.

```
if (nradon > 0 .and. nradon <= nt) then
    call radon_sourcesink (Argument list)
    rdt(:,:,:,nradon)=rdt(:,:,:,nradon)+rtnd(:,:,:)
endif
```

It is the users responsibility to add the tendency generated by the sourcesink routine. Within tracer_driver_end the user can add calls to the terminators for the appropriate source sink routines.

```
call radon_end
```

This may simply be a deallocation statement or a routine to send output to the logfile stating that the termination routine has been called.

6.194.2 Other modules used

```
fms_mod
        time_manager_mod
      tracer_manager_mod
       field_manager_mod
    tracer_utilities_mod
           constants_mod
         atmos_radon_mod
atmos_carbon_aerosol_mod
    atmos_sulfur_hex_mod
```

6.194.3 Public interface

```
use atmos_tracer_driver_mod [, only:  atmos_tracer_driver,
                                      atmos_tracer_driver_init,
                                      atmos_tracer_driver_end ]
```

atmos_tracer_driver: A routine which allows tracer code to be called.

atmos_tracer_driver_init: Subroutine to initialize the tracer driver module.

atmos_tracer_driver_end: Subroutine to terminate the tracer driver module.

6.194.4 Public data

None.

6.194.5 Public routines

a. **Atmos_tracer_driver**

```
call atmos_tracer_driver (is, ie, js, je, Time, lon, lat, land, phalf, pfull, r, &
↪ u, v, t, q, u_star, rdt, rm, rdiag, kbot)
```

DESCRIPTION This subroutine calls the source sink routines for atmospheric tracers. This is the interface between the dynamical core of the model and the tracer code. It should supply all the necessary information to a user that they need in order to calculate the tendency of that tracer with respect to emissions or chemical losses.

INPUT

is, ie, js, je	Local domain boundaries. [integer]
Time	Model time. [type(time_type)]
lon	Longitude of the centre of the model gridcells [real, dimension(:,:)]
lat	Latitude of the centre of the model gridcells [real, dimension(:,:)]
land	Land/sea mask. [logical, dimension(:,:)]
phalf	Pressures on the model half levels. [real, dimension(:,:,:)]
pfull	Pressures on the model full levels. [real, dimension(:,:,:)]
r	The tracer array in the component model. [real, dimension(:,:,:,:)]
u	Zonal wind speed. [real, dimension(:,:,:)]
v	Meridonal wind speed. [real, dimension(:,:,:)]
t	Temperature. [real, dimension(:,:,:)]
q	Specific humidity. This may also be accessible as a portion of the tracer array. [real, dimension(:,:,:)]
u_star	Friction velocity :: The magnitude of the wind stress is density*(ustar**2) The drag coefficient for momentum is u_star**2/(u**2+v**2) [real, dimension(:,:)]
rm	The tracer array in the component model for the previous timestep. [real, dimension(:,:,:,:)]
kbot	Integer array describing which model layer intercepts the surface. [integer, optional, dimension(:,:)]

INPUT/OUTPUT

rdt	The tendency of the tracer array in the compenent model. The tendency due to sources and sinks computed in the individual tracer routines should be added to this array before exiting tracer_driver. [real, dimension(:,:,:,:)]
rdiag	The array of diagnostic tracers. As these may be changed within the tracer routines for diagnostic purposes, they need to be writable. [real, dimension(:,:,:,:)]

b. Atmos_tracer_driver_init

```
call atmos_tracer_driver_init (lonb,latb, r, mask, axes, Time)
```

DESCRIPTION The purpose of the arguments here are for passing on to the individual tracer code. The user may wish to provide initial values which can be implemented in the initialization part of the tracer code. Remember that the tracer manager will provide a simple fixed or exponential profile if the user provides data for this within the field table. However if a more complicated profile is required then it should be set up in the initialization section of the user tracer code.

INPUT

`lonb`	The longitudes for the local domain. [real, dimension(:)]
`latb`	The latitudes for the local domain. [real, dimension(:)]
`mask`	optional mask (0. or 1.) that designates which grid points are above (=1.) or below (=0.) the ground dimensioned as (nlon,nlat,nlev). [real, optional, dimension(:,:,:)]
`Time`	Model time. [type(time_type)]
`axes`	The axes relating to the tracer array dimensioned as (nlon, nlat, nlev, ntime) [integer, dimension(4)]

INPUT/OUTPUT

`r`	Tracer fields dimensioned as (nlon,nlat,nlev,ntrace). [real, dimension(:,:,:,:)]

c. Atmos_tracer_driver_end

```
call atmos_tracer_driver_end
```

DESCRIPTION Termination routine for tracer_driver. It should also call the destructors for the individual tracer routines.

6.194.6 Data sets

None.

6.194.7 Error messages

FATAL in atmos_tracer_driver tracer_driver_init must be called first. Tracer_driver_init needs to be called before tracer_driver.

top

6.195 module atmos_carbon_aerosol_mod

6.195.1 Overview

This code allows the implementation of black and organic carbon tracers in the FMS framework.

This module presents the method of Cooke et al. (1999, 2002) In its present implementation the black and organic carbon tracers are from the combustion of fossil fuel. While the code here should provide insights into the carbonaceous aerosol cycle it is provided here more as an example of how to implement a tracer module in the FMS infrastructure. The parameters of the model should be checked and set to values corresponding to previous works if a user wishes to try to reproduce those works.

6.195.2 Other modules used

```
fms_mod
    time_manager_mod
    diag_manager_mod
  tracer_manager_mod
   field_manager_mod
tracer_utilities_mod
       constants_mod
```

6.195.3 Public interface

```
use atmos_carbon_aerosol_mod [, only:  atmos_blackc_sourcesink,
                                       atmos_organic_sourcesink,
                                       atmos_carbon_aerosol_init,
                                       atmos_carbon_aerosol_end ]
```

atmos_blackc_sourcesink: A subroutine to calculate the source and sinks of black carbon aerosol.

atmos_organic_sourcesink: A subroutine to calculate the source and sinks of organic carbon aerosol.

atmos_carbon_aerosol_init: Subroutine to initialize the carbon aerosol module.

atmos_carbon_aerosol_end: The destructor routine for the carbon aerosol module.

6.195.4 Public data

None.

6.195.5 Public routines

a. **Atmos_blackc_sourcesink**

```
call atmos_blackc_sourcesink (lon, lat, land, pwt, & black_cphob, black_cphob_dt,␣
↪& black_cphil, black_cphil_dt, & Time, is, ie, js, je, kbot)
```

DESCRIPTION

This routine calculates the source and sink terms for black carbon. Simply put, the hydrophobic aerosol has sources from emissions and sinks from dry deposition and transformation into hydrophilic aerosol. The hydrophilic aerosol also has emission sources and has sinks of wet and dry deposition.

The following schematic shows how the black carbon scheme is implemented.

```
+------------+  Trans-   +------------+
|  Hydro-     | formation |  Hydro-     |
|  phobic     |           |  philic     |
|  black      |---------->|  black      |
|  carbon     |           |  carbon     |
|             |           |             |
+------------+           +------------+
   ^      |                ^    |   |
   |      |                |    |   |
   |      =                |    =   =
 Source  Dry            Source Dry Wet
         Dep.                  Dep Dep
```

The transformation time used here is 1 day, which corresponds to an e-folding time of 1.44 days. This can be varied as necessary.

INPUT

`lon`	Longitude of the centre of the model gridcells [real, dimension(:,:)]
`lat`	Latitude of the centre of the model gridcells [real, dimension(:,:)]
`land`	Land/sea mask. [real, dimension(:,:)]
`pwt`	The pressure weighting array. = dP/grav [real, dimension(:,:,:)]
`black_cphob`	The array of the hydrophobic black carbon aerosol mixing ratio [real, dimension(:,:,:)]
`black_cphil`	The array of the hydrophilic black carbon aerosol mixing ratio [real, dimension(:,:,:)]
`Time`	Model time. [type(time_type)]
`is, ie, js, je`	Local domain boundaries. [integer]
`kbot`	Integer array describing which model layer intercepts the surface. [integer, optional, dimension(:,:)]

OUTPUT

black_cphob_dt	The array of the tendency of the hydrophobic black carbon aerosol mixing ratio. [real, dimension(:,:,:)]
black_cphil_dt	The array of the tendency of the hydrophilic black carbon aerosol mixing ratio. [real, dimension(:,:,:)]

b. Atmos_organic_sourcesink

```
call atmos_organic_sourcesink (lon, lat, land, pwt, organic_carbon, organic_
↪carbon_dt, & Time, is, ie, js, je, kbot)
```

DESCRIPTION

This routine calculates the source and sink terms for organic carbon. Simply put, the hydrophobic aerosol has sources from emissions and sinks from dry deposition and transformation into hydrophilic aerosol. The hydrophilic aerosol also has emission sources and has sinks of wet and dry deposition.

The following schematic shows how the organic carbon scheme is implemented.

```
+------------+  Trans-   +------------+
|  Hydro-    | formation |  Hydro-    |
|  phobic    |           |  philic    |
|  organic   |---------->|  organic   |
|  carbon    |           |  carbon    |
|            |           |            |
+------------+           +------------+
   ^      |                 ^    |   |
   |      |                 |    |   |
   |      =                 |    =   =
 Source  Dry             Source Dry Wet
         Dep.                   Dep Dep
```

The transformation time used here is 2 days, which corresponds to an e-folding time of 2.88 days. This can be varied as necessary.

INPUT

lon	Longitude of the centre of the model gridcells [real, dimension(:,:)]
lat	Latitude of the centre of the model gridcells [real, dimension(:,:)]
land	Land/sea mask. [real, dimension(:,:)]
pwt	The pressure weighting array. = dP/grav [real, dimension(:,:,:)]
organic_carbon	The array of the organic carbon aerosol mixing ratio [real, dimension(:,:,:)]
Time	Model time. [type(time_type)]
is, ie, js, je	Local domain boundaries. [integer]
kbot	Integer array describing which model layer intercepts the surface. [integer, optional, dimension(:,:)]

OUTPUT

organic_carbon_dt	The array of the tendency of the organic carbon aerosol mixing ratio. [real, dimension(:,:,:)]

c. **Atmos_carbon_aerosol_init**

```
call atmos_carbon_aerosol_init (lonb, latb, r, axes, Time, mask)
```

DESCRIPTION This subroutine querys the tracer manager to find the indices for the various carbonaceous aerosol tracers. It also registers the emission fields for diagnostic purposes.

INPUT

lonb	The longitudes for the local domain. [real, dimension(:)]
latb	The latitudes for the local domain. [real, dimension(:)]
mask	optional mask (0. or 1.) that designates which grid points are above (=1.) or below (=0.) the ground dimensioned as (nlon,nlat,nlev). [real, optional, dimension(:,:,:)]
Time	Model time. [type(time_type)]
axes	The axes relating to the tracer array dimensioned as (nlon, nlat, nlev, ntime) [integer, dimension(4)]

INPUT/OUTPUT

r	Tracer fields dimensioned as (nlon,nlat,nlev,ntrace). [real, dimension(:,:,:,:)]

d. **Atmos_carbon_aerosol_end**

```
call atmos_carbon_aerosol_end
```

DESCRIPTION This subroutine writes the version name to logfile and exits.

6.195.6 Data sets

Black carbon emissions The black carbon emission dataset is that derived in Cooke et al. (1999) The dataset can be obtained from the contact person above.

Organic carbon emissions The organic carbon emission dataset is that derived in Cooke et al. (1999) The dataset can be obtained from the contact person above.

6.195.7 Error messages

None.

6.195.8 References

1. Cooke, W. F. and J. J. N. Wilson, A global black carbon aerosol model, J. Geophys. Res., 101, 19395-19409, 1996.

2. Cooke, W. F., C. Liousse, H. Cachier and J. Feichter, Construction of a 1 x 1 fossil fuel emission dataset for carbonaceous aerosol and implementation and radiative impact in the ECHAM-4 model, J. Geophys. Res., 104, 22137-22162, 1999

3. Cooke, W.F., V. Ramaswamy and P. Kasibathla, A GCM study of the global carbonaceous aerosol distribution. J. Geophys. Res., 107, accepted, 2002

6.195.9 Compiler specifics

None.

6.195.10 Precompiler options

None.

6.195.11 Loader options

None.

6.195.12 Test PROGRAM

None.

6.195.13 Notes

None.

6.196 module atmos_tracer_utilities_mod

6.196.1 Overview

This code provides some utility routines for atmospheric tracers in the FMS framework.

This module gives utility routines which can be used to provide consistent removal mechanisms for atmospheric tracers. In particular it provides schemes for wet and dry deposiiton that can be easily utilized.

6.196.2 Other modules used

```
fms_mod
  time_manager_mod
  diag_manager_mod
tracer_manager_mod
 field_manager_mod
     constants_mod
  horiz_interp_mod
```

6.196.3 Public interface

```
use atmos_tracer_utilities_mod [, only:  atmos_tracer_utilities_init,
                                         dry_deposition,
                                         wet_deposition,
                                         interp_emiss,
                                         tracer_utilities_end ]
```

atmos_tracer_utilities_init: This is a routine to create and register the dry and wet deposition fields of the tracers.

dry_deposition: Routine to calculate the fraction of tracer to be removed by dry deposition.

wet_deposition: Routine to calculate the fraction of tracer removed by wet deposition

interp_emiss: A routine to interpolate emission fields of arbitrary resolution onto the resolution of the model.

tracer_utilities_end: The destructor routine for the tracer utilities module.

6.196.4 Public data

None.

6.196.5 Public routines

a. **Atmos_tracer_utilities_init**

```
call atmos_tracer_utilities_init (lonb,latb, mass_axes, Time)
```

DESCRIPTION This routine creates diagnostic names for dry and wet deposition fields of the tracers. It takes the tracer name and appends "ddep" for the dry deposition field and "wdep" for the wet deposition field. This names can then be entered in the diag_table for diagnostic output of the tracer dry and wet deposition. The module name associated with these fields in "tracers". The units of the deposition fields are assumed to be kg/m2/s.

INPUT

`lonb`	The longitudes for the local domain. [real, dimension(:)]
`latb`	The latitudes for the local domain. [real, dimension(:)]
`mass_axes`	The axes relating to the tracer array. [integer, dimension(3)]
`Time`	Model time. [type(time_type)]

b. **Dry_deposition**

```
call dry_deposition ( n, is, js, u, v, T, pwt, pfull, u_star, landmask, dsinku,␣
↪tracer, Time)
```

DESCRIPTION

There are two types of dry deposition coded.

1) Wind driven derived dry deposition velocity.

2) Fixed dry deposition velocity.

The theory behind the wind driven dry deposition velocity calculation assumes that the deposition can be modeled as a parallel resistance type problem.

Total resistance to HNO3-type dry deposition,

```
     R = Ra + Rb
resisa = aerodynamic resistance
resisb = surface resistance (laminar layer + uptake)
       = 5/u*   [s/cm]        for neutral stability
    Vd = 1/R
```

For the fixed dry deposition velocity, there is no change in the deposition velocity but the variation of the depth of the surface layer implies that there is variation in the amount deposited.

To utilize this section of code add one of the following lines as a method for the tracer of interest in the field table.

```
"dry_deposition","wind_driven","surfr=XXX"
    where XXX is the total resistance defined above.

"dry_deposition","fixed","land=XXX, sea=YYY"
    where XXX is the dry deposition velocity (m/s) over land
      and YYY is the dry deposition velocity (m/s) over sea.
```

INPUT

n	The tracer number. [integer]
is, js	Start indices for array (computational indices). [integer]
u	U wind field. [real, dimension(:,:)]
v	V wind field. [real, dimension(:,:)]
T	Temperature. [real, dimension(:,:)]
pwt	Pressure differential of half levels. [real, dimension(:,:)]
pfull	Full pressure levels. [real, dimension(:,:)]
u_star	Friction velocity. [real, dimension(:,:)]
landmask	Land - sea mask. [logical]

OUTPUT

dsinku	The amount of tracer in the surface layer which is dry deposited per second. [real, dimension(:,:)]

c. Wet_deposition

```
call wet_deposition (n, T, pfull, phalf, rain, snow, qdt, tracer, tracer_dt, Time,
↪ cloud_param, is, js)
```

DESCRIPTION

Schemes allowed here are

1) Deposition removed in the same fractional amount as the modeled precipitation rate is to a standardized precipitation rate. Basically this scheme assumes that a fractional area of the gridbox is affected by precipitation and that this precipitation rate is due to a cloud of standardized cloud liquid water content. Removal is constant throughout the column where precipitation is occuring.

2) Removal according to Henry's Law. This law states that the ratio of the concentation in cloud water and the partial pressure in the interstitial air is a constant. In this instance, the units for Henry's constant are kg/L/Pa (normally it is M/L/Pa) Parameters for a large number of species can be found at http://www.mpch-mainz.mpg.de/~sander/res/henry.html To utilize this section of code add one of the following lines as a method for the tracer of interest in the field table.

```
"wet_deposition","henry","henry=XXX, dependence=YYY"
    where XXX is the Henry's constant for the tracer in question
      and YYY is the temperature dependence of the Henry's Law constant.

"wet_deposition","fraction","lslwc=XXX, convlwc=YYY"
    where XXX is the liquid water content of a standard large scale cloud
      and YYY is the liquid water content of a standard convective cloud.
```

INPUT

`n`	Tracer number [integer]
`is, js`	start indices for array (computational indices) [integer]
`T`	Temperature [real, dimension(:,:,:)]
`pfull`	Full level pressure field [real, dimension(:,:,:)]
`phalf`	Half level pressure field [real, dimension(:,:,:)]
`rain`	Precipitation in the form of rain [real, dimension(:,:)]
`snow`	Precipitation in the form of snow [real, dimension(:,:)]
`qdt`	The tendency of the specific humidity due to the cloud parametrization [real, dimension(:,:,:)]
`tracer`	The tracer field [real, dimension(:,:,:)]
`Time`	The time structure for submitting wet deposition as a diagnostic [type(time_type)]
`cloud_param`	Is this a convective (convect) or large scale (lscale) cloud parametrization? [character]

OUTPUT

`tracer_dt`	The tendency of the tracer field due to wet deposition. [real, dimension(:,:,:)]

d. Interp_emiss

```
call interp_emiss (global_source, start_lon, start_lat, & lon_resol, lat_resol,␣
↪data_out)
```

DESCRIPTION Routine to interpolate emission fields (or any 2D field) to the model resolution. The local section of the global field is returned to the local processor.

INPUT

`global_source`	Global emission field. [real, dimension(:,:)]
`start_lon`	Longitude of starting point of emission field (in radians). This is the westernmost boundary of the global field. [real]
`start_lat`	Latitude of starting point of emission field (in radians). This is the southern boundary of the global field. [real]
`lon_resol`	Longitudinal resolution of the emission data (in radians). [real]
`lat_resol`	Latitudinal resolution of the emission data (in radians). [real]

OUTPUT

`data_out`	Interpolated emission field on the local PE. [real, dimension(:,:)]

e. Tracer_utilities_end

DESCRIPTION This subroutine writes the version name to logfile and exits.

6.196.6 Data sets

None.

6.196.7 Error messages

None.

top

6.197 module vert_advection_mod

6.197.1 Overview

Computes the tendency due to vertical advection for an arbitrary quantity.

The advective tendency may be computed in *advective form* (for use in spectral models) or *flux form*. The spatial differencing may be *centered* (second or fourth order) or *finite volume* (van Leer) using a piecewise linear method.

6.197.2 Other modules used

```
fms_mod
```

6.197.3 Public interface

```
use vert_advection_mod [, only:  vert_advection,
                                 SECOND_CENTERED, FOURTH_CENTERED, VAN_LEER_LINEAR,
                                 FLUX_FORM, ADVECTIVE_FORM  ]
```

vert_advection: Computes the vertical advective tendency for an arbitrary quantity. There is no initialization routine necessary.

6.197.4 Public routines

a. **Vert_advection**

```
call vert_advection ( dt, w, dz, r, rdt [, mask, scheme, form] )

DESCRIPTION
   This routine computes the vertical advective tendency for
   an arbitrary quantity. The tendency can be computed using
   one of several different choices for spatial differencing
   and numerical form of the equations. These choices are
   controlled through optional arguments.
   There is no initialization routine necessary.

INPUT
   dt   time step in seconds [real]

   w    advecting velocity at the vertical boundaries of the grid boxes
        does not assume velocities at top and bottom are zero
        units = [units of dz / second]
        [real, dimension(:,:,:)]
        [real, dimension(:,:)]
        [real, dimension(:)]

   dz   depth of model layers in arbitrary units (usually pressure)
        [real, dimension(:,:,:)]
        [real, dimension(:,:)]
        [real, dimension(:)]

   r    advected quantity in arbitrary units
        [real, dimension(:,:,:)]
        [real, dimension(:,:)]
        [real, dimension(:)]

OUTPUT
   rdt  advective tendency for quantity "r" weighted by depth of layer
        units = [units of r * units of dz / second]
```

(continues on next page)

(continued from previous page)

```
        [real, dimension(:,:,:)]
        [real, dimension(:,:)]
        [real, dimension(:)]

OPTIONAL INPUT
   mask     mask for below ground layers,
            where mask > 0 for layers above ground
            [real, dimension(:,:,:)]
            [real, dimension(:,:)]
            [real, dimension(:)]

   scheme   spatial differencing scheme, use one of these values:
               SECOND_CENTERED = second-order centered
               FOURTH_CENTERED = fourth-order centered
               VAN_LEER_LINEAR = piecewise linear, finite volume (van Leer)
            [integer, default=VAN_LEER_LINEAR]

   form     form of equations, use one of these values:
               FLUX_FORM      = solves for -d(w*r)/dt
               ADVECTIVE_FORM = solves for -w*d(r)/dt
            [integer, default=FLUX_FORM]

NOTE
   The input/output arrays may be 1d, 2d, or 3d.
   The last dimension is always the vertical dimension.
   For the vertical dimension the following must be true:
   size(w,3) == size(dz,3)+1 == size(r,3)+1 == size(rdt,3)+1 == size(mask,3)+1
   All horizontal dimensions must have the same size (no check is done).
```

6.197.5 Error messages

Errors in vert_advection_mod vertical dimension of input arrays inconsistent The following was not true: size(w,3) = size(r,3)+1. invalid value for optional argument scheme The value of optional argument scheme must be one of the public parameters SECOND_CENTERED, FOURTH_CENTERED, or VAN_LEER_LINEAR. invalid value for optional argument form The value of optional argument form must be one of the public parameters FLUX_FORM or ADVECTIVE_FORM.

6.197.6 References

1. Lin, S.-J., W.C. Chao, Y.C. Sud, and G.K. Walker, 1994: A class of the van Leer-type transport schemes and its application to the moisture in a general circulation model. *Mon. Wea. Rev.*, **122**, 1575-1593.

6.197.7 Notes

None.

6.198 module time_manager_mod

6.198.1 Overview

A software package that provides a set of simple interfaces for modelers to perform computations related to time and dates.

The module defines a type that can be used to represent discrete times (accurate to one second) and to map these times into dates using a variety of calendars. A time is mapped to a date by representing the time with respect to an arbitrary base date (refer to <B>NOTES</B> section for the base date setting). The time_manager provides a single defined type, time_type, which is used to store time and date quantities. A time_type is a positive definite quantity that represents an interval of time. It can be most easily thought of as representing the number of seconds in some time interval. A time interval can be mapped to a date under a given calendar definition by using it to represent the time that has passed since some base date. A number of interfaces are provided to operate on time_type variables and their associated calendars. Time intervals can be as large as n days where n is the largest number represented by the default integer type on a compiler. This is typically considerably greater than 10 million years (assuming 32 bit integer representation) which is likely to be adequate for most applications. The description of the interfaces is separated into two sections. The first deals with operations on time intervals while the second deals with operations that convert time intervals to dates for a given calendar.

6.198.2 Other modules used

```
fms_mod
```

6.198.3 Public interface

```
use time_manager_mod [, only:  set_time,
                               get_time,
                               increment_time,
                               decrement_time,
                               time_gt,
                               time_ge,
                               time_lt,
                               time_le,
                               time_eq,
                               time_ne,
                               time_plus,
                               time_minus,
                               time_scalar_mult,
```

(continues on next page)

(continued from previous page)

```
                               scalar_time_mult,
                               time_divide,
                               time_real_divide,
                               time_scalar_divide,
                               interval_alarm,
                               repeat_alarm,
                               set_calendar_type,
                               get_calendar_type,
                               get_date,
                               set_date,
                               increment_date,
                               decrement_date,
                               days_in_month,
                               leap_year,
                               length_of_year,
                               days_in_year,
                               month_name,
                               time_manager_init,
                               print_time,
                               print_date ]
```

set_time: Given some number of seconds and days, returns the corresponding time_type.

get_time: Given a time interval, returns the corresponding seconds and days.

increment_time: Given a time and an increment of days and seconds, returns a time that adds this increment to an input time.

decrement_time: Given a time and a decrement of days and seconds, returns a time that subtracts this decrement from an input time.

time_gt: Returns true if time1 > time2.

time_ge: Returns true if time1 >= time2.

time_lt: Returns true if time1 < time2.

time_le: Returns true if time1 <= time2.

time_eq: Returns true if time1 == time2.

time_ne: Returns true if time1 /= time2.

time_plus: Returns sum of two time_types.

time_minus: Returns difference of two time_types.

time_scalar_mult: Returns time multiplied by integer factor n.

scalar_time_mult: Returns time multiplied by integer factor n.

time_divide: Returns the largest integer, n, for which time1 >= time2 * n.

time_real_divide: Returns the double precision quotient of two times.

time_scalar_divide: Returns the largest time, t, for which n * t <= time.

interval_alarm: Given a time, and a time interval, this function returns true if this is the closest time step to the alarm time.

repeat_alarm: Repeat_alarm supports an alarm that goes off with alarm_frequency and lasts for alarm_length.

set_calendar_type: Sets the default calendar type for mapping time intervals to dates.

get_calendar_type: Returns the value of the default calendar type for mapping from time to date.

get_date: Given a time_interval, returns the corresponding date under the selected calendar.

set_date: Given an input date in year, month, days, etc., creates a time_type that represents this time interval from the internally defined base date.

increment_date: Increments the date represented by a time interval and the default calendar type by a number of seconds, etc.

decrement_date: Decrements the date represented by a time interval and the default calendar type by a number of seconds, etc.

days_in_month: Given a time interval, gives the number of days in the month corresponding to the default calendar.

leap_year: Returns true if the year corresponding to the date for the default calendar is a leap year. Returns false for THIRTY_DAY_MONTHS and NO_LEAP.

length_of_year: Returns the mean length of the year in the default calendar setting.

days_in_year: Returns the number of days in the calendar year corresponding to the date represented by time for the default calendar.

month_name: Returns a character string containing the name of the month corresponding to month number n.

time_manager_init: Write the version information to the log file.

print_time: Prints the given time_type argument as a time (using days and seconds).

print_date: prints the time to standard output (or optional unit) as a date.

6.198.4 Public data

Name	Type	Value	Units	Description
time_type	derived type	—	—	Derived-type data variable used to store time and date quantities. It contains two PRIVATE variables: seconds and days.

6.198.5 Public routines

a. **Set_time**

```
<B> set_time </B>(seconds, days)
```

DESCRIPTION Given some number of seconds and days, returns the corresponding time_type.

INPUT

`seconds`	A number of seconds (can be greater than 86400), must be positive. [integer, dimension(scalar)]
`days`	A number of days, must be positive. [integer, dimension(scalar)]

OUTPUT

	A time interval corresponding to this number of days and seconds. [, dimension]

b. **Get_time**

```
call get_time </B>(time, seconds, days)
```

DESCRIPTION Given a time interval, returns the corresponding seconds and days.

INPUT

`time`	A time interval. [time_type]

OUTPUT

`seconds`	A number of seconds (< 86400). [integer, dimension(scalar)]
`days`	A number of days, must be positive. [integer, dimension(scalar)]

c. **Increment_time**

```
increment_time (time, seconds, days)
```

DESCRIPTION Given a time and an increment of days and seconds, returns a time that adds this increment to an input time. Increments a time by seconds and days; increments cannot be negative.

INPUT

`time`	A time interval. [time_type, dimension]
`seconds`	Increment of seconds (can be greater than 86400); must be positive. [integer, dimension(scalar)]
`days`	Increment of days; must be positive. [integer, dimension(scalar)]

OUTPUT

	A time that adds this increment to the input time. [, dimension]

d. **Decrement_time**

```
decrement_time (time, seconds, days)
```

DESCRIPTION Decrements a time by seconds and days; decrements cannot be negative.

INPUT

`time`	A time interval. [time_type, dimension]
`seconds`	Decrement of seconds (can be greater than 86400); must be positive. [integer, dimension(scalar)]
`days`	Decrement of days; must be positive. [integer, dimension(scalar)]

OUTPUT

	A time that subtracts this decrement from an input time. If the result is negative, it is considered a fatal error. [, dimension]

e. **Time_gt**

```
<B> time_gt </B>(time1, time2)
```

DESCRIPTION Returns true if time1 > time2.

INPUT

`time1`	A time interval. [time_type, dimension]
`time2`	A time interval. [time_type, dimension]

OUTPUT

	Returns true if time1 > time2 [logical, dimension]

f. **Time_ge**

```
<B> time_ge </B>(time1, time2)
```

DESCRIPTION Returns true if time1 >= time2.

INPUT

`time1`	A time interval. [time_type, dimension]
`time2`	A time interval. [time_type, dimension]

OUTPUT

	Returns true if time1 >= time2 [logical, dimension]

g. **Time_lt**

```
<B> time_lt </B>(time1, time2)
```

DESCRIPTION Returns true if time1 < time2.

INPUT

`time1`	A time interval. [time_type, dimension]
`time2`	A time interval. [time_type, dimension]

OUTPUT

	Returns true if time1 < time2 [logical, dimension]

h. Time_le

```
<B> time_le </B>(time1, time2)
```

DESCRIPTION Returns true if time1 <= time2.

INPUT

`time1`	A time interval. [time_type, dimension]
`time2`	A time interval. [time_type, dimension]

OUTPUT

	Returns true if time1 <= time2 [logical, dimension]

i. Time_eq

```
<B> time_eq </B>(time1, time2)
```

DESCRIPTION Returns true if time1 == time2.

INPUT

`time1`	A time interval. [time_type, dimension]
`time2`	A time interval. [time_type, dimension]

OUTPUT

	Returns true if time1 == time2 [logical, dimension]

j. Time_ne

```
<B> time_ne </B>(time1, time2)
```

DESCRIPTION Returns true if time1 /= time2.

INPUT

`time1`	A time interval. [time_type, dimension]
`time2`	A time interval. [time_type, dimension]

OUTPUT

	Returns true if time1 /= time2 [logical, dimension]

k. **Time_plus**

```
<B> time_plus </B>(time1, time2)
```

DESCRIPTION Returns sum of two time_types.

INPUT

time1	A time interval. [time_type, dimension]
time2	A time interval. [time_type, dimension]

OUTPUT

	Returns sum of two time_types. [time_type, dimension]

l. **Time_minus**

```
<B> time_minus </B>(time1, time2)
```

DESCRIPTION Returns difference of two time_types. WARNING: a time type is positive so by definition time1 - time2 is the same as time2 - time1.

INPUT

time1	A time interval. [time_type, dimension]
time2	A time interval. [time_type, dimension]

OUTPUT

	Returns difference of two time_types. [time_type, dimension]

m. **Time_scalar_mult**

```
<B> time_scalar_mult </B>(time, n)
```

DESCRIPTION Returns time multiplied by integer factor n.

INPUT

time	A time interval. [time_type, dimension]
n	A time interval. [integer, dimension]

OUTPUT

	Returns time multiplied by integer factor n. [time_type, dimension]

n. Scalar_time_mult

```
<B> scalar_time_mult </B>(n, time)
```

DESCRIPTION Returns time multiplied by integer factor n.

INPUT

`time`	A time interval. [time_type, dimension]
`n`	An integer. [integer, dimension]

OUTPUT

	Returns time multiplied by integer factor n. [time_type, dimension]

o. Time_divide

```
<B> time_divide </B>(time1, time2)
```

DESCRIPTION Returns the largest integer, n, for which time1 >= time2 * n.

INPUT

`time1`	A time interval. [time_type, dimension]
`time2`	A time interval. [time_type, dimension]

OUTPUT

	Returns the largest integer, n, for which time1 >= time2 * n. [integer, dimension]

p. Time_real_divide

```
<B> time_real_divide </B>(time1, time2)
```

DESCRIPTION Returns the double precision quotient of two times.

INPUT

`time1`	A time interval. [time_type, dimension]
`time2`	A time interval. [time_type, dimension]

OUTPUT

	Returns the double precision quotient of two times [integer, dimensiondouble precision]

q. Time_scalar_divide

```
<B> time_scalar_divide </B>(time, n)
```

DESCRIPTION Returns the largest time, t, for which n * t <= time.

INPUT

`time`	A time interval. [time_type, dimension]
`n`	An integer factor. [integer, dimension]

OUTPUT

	Returns the largest time, t, for which n * t <= time. [integer, dimensiondouble precision]

r. Interval_alarm

```
interval_alarm (time, time_interval, alarm, alarm_interval)
```

DESCRIPTION This is a specialized operation that is frequently performed in models. Given a time, and a time interval, this function is true if this is the closest time step to the alarm time. The actual computation is: if((alarm_time - time) <= (time_interval / 2)) If the function is true, the alarm time is incremented by the alarm_interval; WARNING, this is a featured side effect. Otherwise, the function is false and there are no other effects. CAUTION: if the alarm_interval is smaller than the time_interval, the alarm may fail to return true ever again. Watch for problems if the new alarm time is less than time + time_interval

INPUT

`time`	Current time. [time_type]
`time_interval`	A time interval. [time_type]
`alarm_interval`	A time interval. [time_type]

INPUT/OUTPUT

`alarm`	An alarm time, which is incremented by the alarm_interval if the function is true. [time_type]

OUTPUT

`interval_alarm`	Returns either True or false. [logical]

s. Repeat_alarm

```
repeat_alarm
```

DESCRIPTION Repeat_alarm supports an alarm that goes off with alarm_frequency and lasts for alarm_length. If the nearest occurence of an alarm time is less than half an alarm_length from the input time, repeat_alarm is true. For instance, if the alarm_frequency is 1 day, and the alarm_length is 2 hours, then repeat_alarm is true from time 2300 on day n to time 0100 on day n + 1 for all n.

INPUT

`time`	Current time. [time_type]
`alarm_frequency`	A time interval for alarm_frequency. [time_type]
`alarm_length`	A time interval for alarm_length. [time_type]

OUTPUT

`repeat_alarm`	Returns either True or false. [logical]

t. Set_calendar_type

```
call set_calendar_type (type)
```

DESCRIPTION A constant number for setting the calendar type.

INPUT

`type`	A constant number for setting the calendar type. [integer, dimension]

OUTPUT

`calendar_type`	A constant number for default calendar type. [integer]

NOTE At present, four integer constants are defined for setting the calendar type: THIRTY_DAY_MONTHS, JULIAN, NO_LEAP, and GREGORIAN. However, GREGORIAN CALENDAR is not completely implemented. Selection of this type will result in illegal type error.

u. Get_calendar_type

```
get_calendar_type ()
```

DESCRIPTION There are no arguments in this function. It returns the value of the default calendar type for mapping from time to date.

v. Get_date

```
call get_date (time, year, month, day, hour, minute, second)
```

DESCRIPTION Given a time_interval, returns the corresponding date under the selected calendar.

INPUT

`time`	A time interval. [time_type]

OUTPUT

`day`	[integer]
`month`	[integer]
`year`	[integer]
`second`	[integer]
`minute`	[integer]
`hour`	[integer]

NOTE For all but the thirty_day_months calendar, increments to months and years must be made separately from other units because of the non-associative nature of the addition. All the input increments must be positive.

w. Set_date

```
set_date (year, month, day, hours, minutes, seconds)
```

DESCRIPTION Given a date, computes the corresponding time given the selected date time mapping algorithm. Note that it is possible to specify any number of illegal dates; these should be checked for and generate errors as appropriate.

INPUT

`time`	A time interval. [time_type]
`day`	[integer]
`month`	[integer]
`year`	[integer]
`second`	[integer]
`minute`	[integer]
`hour`	[integer]

OUTPUT

`set_date`	A time interval. [time_type]

x. Increment_date

```
increment_date (time, years, months, days, hours, minutes, seconds)
```

DESCRIPTION Given a time and some date increment, computes a new time. Depending on the mapping algorithm from date to time, it may be possible to specify undefined increments (i.e. if one increments by 68 days and 3 months in a Julian calendar, it matters which order these operations are done and we don't want to deal with stuff like that, make it an error).

INPUT

`time`	A time interval. [time_type]
`day`	An increment of days. [integer]
`month`	An increment of months. [integer]
`year`	An increment of years. [integer]
`second`	An increment of seconds. [integer]
`minute`	An increment of minutes. [integer]
`hour`	An increment of hours. [integer]

OUTPUT

`increment_date`	A new time based on the input time interval and the default calendar type. [time_type]

y. Decrement_date

```
decrement_date (time, years, months, days, hours, minutes, seconds)
```

DESCRIPTION Given a time and some date decrement, computes a new time. Depending on the mapping algorithm from date to time, it may be possible to specify undefined decrements (i.e. if one decrements by 68 days and 3 months in a Julian calendar, it matters which order these operations are done and we don't want to deal with stuff like that, make it an error).

INPUT

`time`	A time interval. [time_type]
`day`	A decrement of days. [integer]
`month`	A deincrement of months. [integer]
`year`	A deincrement of years. [integer]
`second`	A deincrement of seconds. [integer]
`minute`	A deincrement of minutes. [integer]
`hour`	A deincrement of hours. [integer]

OUTPUT

`decrement_date`	A new time based on the input time interval and the default calendar type. [time_type]

NOTE For all but the thirty_day_months calendar, decrements to months and years must be made separately from other units because of the non-associative nature of addition. All the input decrements must be positive. If the result is a negative time (i.e. date before the base date) it is considered a fatal error.

z. Days_in_month

```
<B> days_in_month (time)
```

DESCRIPTION Given a time, computes the corresponding date given the selected date time mapping algorithm.

INPUT

`time`	A time interval. [time_type, dimension]

OUTPUT

`days_in_month`	The number of days in the month given the selected time mapping algorithm. [integer, dimension]

a. Leap_year

```
leap_year (time)
```

DESCRIPTION Is this date in a leap year for default calendar? Returns true if the year corresponding to the date for the default calendar is a leap year. Returns false for THIRTY_DAY_MONTHS and NO_LEAP.

INPUT

`time`	A time interval. [time_type, dimension]

OUTPUT

`leap_year`	True if the year corresponding to the date for the default calendar is a leap year. False for THIRTY_DAY_MONTHS and NO_LEAP and otherwise. [calendar_type, dimension]

b. Length_of_year

```
length_of_year ()
```

DESCRIPTION There are no arguments in this function. It returns the mean length of the year in the default calendar setting.

c. Days_in_year

```
days_in_year ()
```

DESCRIPTION Returns the number of days in the calendar year corresponding to the date represented by time for the default calendar.

INPUT

`time`	A time interval. [time_type]

OUTPUT

	The number of days in this year for the default calendar type.

d. Month_name

```
month_name (n)
```

DESCRIPTION Returns a character string containing the name of the month corresponding to month number n. Definition is the same for all calendar types.

INPUT

`n`	Month number. [integer]

OUTPUT

`month_name`	The character string associated with a month. For now all calendars have 12 months and will return standard names. [character]

e. Time_manager_init

```
time_manager_init ()
```

DESCRIPTION Initialization routine. This routine does not have to be called, all it does is write the version information to the log file.

f. Print_time

```
print_time (time,str,unit)
```

DESCRIPTION Prints the given time_type argument either as a time (using days and seconds). NOTE: there is no check for PE number.

INPUT

`time`	Time that will be printed. [time_type]
`str`	Character string that precedes the printed time or date. [character (len=*)]
`unit`	Unit number for printed output. The default unit is stdout. [integer]

g. Print_date

```
print_date (time,str,unit)
```

DESCRIPTION Prints the given time_type argument as a date (using year,month,day, hour,minutes and seconds). NOTE: there is no check for PE number.

INPUT

`time`	Time that will be printed. [time_type]
`str`	Character string that precedes the printed time or date. [character (len=*)]
`unit`	Unit number for printed output. The default unit is stdout. [integer]

6.198.6 Data sets

None.

6.198.7 Error messages

None.

6.198.8 References

None.

6.198.9 Compiler specifics

None.

6.198.10 Precompiler options

None.

6.198.11 Loader options

None.

6.198.12 Test PROGRAM

time_main2

```
        use time_manager_mod
        implicit none
        type(time_type) :: dt, init_date, astro_base_date, time, final_date
        type(time_type) :: next_rad_time, mid_date
        type(time_type) :: repeat_alarm_freq, repeat_alarm_length
        integer :: num_steps, i, days, months, years, seconds, minutes, hours
        integer :: months2, length
        real :: astro_days

Set calendar type
    call set_calendar_type(THIRTY_DAY_MONTHS)
        call set_calendar_type(JULIAN)
    call set_calendar_type(NO_LEAP)

 Set timestep
        dt = set_time(1100, 0)
```

(continues on next page)

(continued from previous page)

```
Set initial date
       init_date = set_date(1992, 1, 1)

Set date for astronomy delta calculation
       astro_base_date = set_date(1970, 1, 1, 12, 0, 0)

Copy initial time to model current time
       time = init_date

Determine how many steps to do to run one year
       final_date = increment_date(init_date, years = 1)
       num_steps = (final_date - init_date) / dt
       write(*, *) 'Number of steps is' , num_steps

Want to compute radiation at initial step, then every two hours
       next_rad_time = time + set_time(7200, 0)

Test repeat alarm
       repeat_alarm_freq = set_time(0, 1)
       repeat_alarm_length = set_time(7200, 0)

Loop through a year
       do i = 1, num_steps

Increment time
       time = time + dt

Test repeat alarm
       if(repeat_alarm(time, repeat_alarm_freq, repeat_alarm_length)) &
       write(*, *) 'REPEAT ALARM IS TRUE'

Should radiation be computed? Three possible tests.
First test assumes exact interval; just ask if times are equal
    if(time == next_rad_time) then
Second test computes rad on last time step that is <= radiation time
    if((next_rad_time - time) < dt .and. time < next_rad) then
Third test computes rad on time step closest to radiation time
        if(interval_alarm(time, dt, next_rad_time, set_time(7200, 0))) then
          call get_date(time, years, months, days, hours, minutes, seconds)
          write(*, *) days, month_name(months), years, hours, minutes, seconds

Need to compute real number of days between current time and astro_base
          call get_time(time - astro_base_date, seconds, days)
          astro_days = days + seconds / 86400.
      write(*, *) 'astro offset ', astro_days
       end if

Can compute daily, monthly, yearly, hourly, etc. diagnostics as for rad

Example: do diagnostics on last time step of this month
       call get_date(time + dt, years, months2, days, hours, minutes, seconds)
       call get_date(time, years, months, days, hours, minutes, seconds)
       if(months /= months2) then
          write(*, *) 'last timestep of month'
          write(*, *) days, months, years, hours, minutes, seconds
       endif
```

(continues on next page)

(continued from previous page)

```
Example: mid-month diagnostics; inefficient to make things clear
       length = days_in_month(time)
       call get_date(time, years, months, days, hours, minutes, seconds)
       mid_date = set_date(years, months, 1) + set_time(0, length) / 2

       if(time < mid_date .and. (mid_date - time) < dt) then
          write(*, *) 'mid-month time'
          write(*, *) days, months, years, hours, minutes, seconds
       endif

       end do
```

end program time_main2

6.198.13 Notes

The Gregorian calendar type is not completely implemented, and currently no effort is put on it since it doesn’t differ from Julian in use between 1901 and 2099. The <a name=”base date”>base date</a> is implicitly defined so users don’t need to be concerned with it. For the curious, the base date is defined as 0 seconds, 0 minutes, 0 hours, day 1, month 1, year 1 for the Julian and thirty_day_months calendars, and 1 January, 1900, 0 seconds, 0 minutes, 0 hour for the Gregorian calendar. Please note that a time is a positive definite quantity. See the Test Program for a simple program that shows some of the capabilities of the time manager.

6.199 module field_manager_mod

6.199.1 Overview

The field manager reads entries from a field table and stores this information along with the type of field it belongs to. This allows the component models to query the field manager to see if non-default methods of operation are desired. In essence the field table is a powerful type of namelist. Default values can be provided for all the fields through a namelist, individual fields can be modified through the field table however.

The field table consists of entries in the following format. The first line of an entry should consist of three quoted strings. The first quoted string will tell the field manager what type of field it is. At present the supported types of fields are “tracer” for tracers, “xland_mix” for cross-land mixing, and, “checkerboard” for checkerboard null mode. The second quoted string will tell the field manager which model the field is being applied to. The supported types at present are “atmos_mod” for the atmosphere model, “ocean_mod” for the ocean model, “land_mod” for the land model, and, “ice_mod” for the ice model. The third quoted string should be a unique name that can be used as a query. The second and following lines of each entry are called methods in this context. Methods can be developed within any module and these modules can query the field manager to find any methods that are supplied in the field table. These lines can consist of two or three quoted strings. The first string will be an identifier that the querying module will ask for. The second string will be a name that the querying module can use to set up values for the module. The third string, if present, can supply parameters to the calling module that can be parsed and used to further modify values.

An entry is ended with a backslash (/) as the final character in a row. Comments can be inserted in the field table by having a # as the first character in the line. An example of a field table entry could be

```
"tracer","atmos_mod","sphum"/
"tracer","atmos_mod","sf6"
"longname","sulf_hex"
"advection_scheme_horiz","2nd_order"
"Profile_type","Fixed","surface_value = 0.0E+00"/
```

In this example we have two field entries. The first is a simple declaration of a tracer called "sphum". The second is for a tracer called "sf6". Methods that are being applied to this tracer include initiating the long name of the tracer to be "sulf_hex", changing the horizontal advection scheme to be second order, and initiating the tracer with a profile with fixed values, in this example all zero.

6.199.2 Other modules used

```
   mpp_mod
mpp_io_mod
   fms_mod
```

6.199.3 Public interface

```
use field_manager_mod [, only:  field_manager_init,
                                field_manager_end,
                                find_field_index,
                                get_field_info,
                                get_field_method,
                                get_field_methods,
                                parse ]
```

field_manager_init: Routine to initialize the field manager.

field_manager_end: Destructor for field manager.

find_field_index: Function to return the index of the field.

get_field_info: This routine allows access to field information given an index.

get_field_method: A routine to get a specified method.

get_field_methods: A routine to obtain all the methods associated with a field.

parse: A function to parse an integer or an array of integers, a real or an array of reals, a string or an array of strings.

6.199.4 Public data

Name	Type	Value	Units	Description
NUM_MODELS	integer, parameter	5	—	Number of models.
module_is_initialized	logical	.false.	—	Field manager is initialized.
MODEL_ATMOS	integer, parameter	1	—	Atmospheric model.
MODEL_OCEAN	integer, parameter	2	—	Ocean model.
MODEL_LAND	integer, parameter	3	—	Land model.
MODEL_ICE	integer, parameter	4	—	Ice model.

6.199.5 Public routines

a. Field_manager_init

```
call field_manager_init (nfields, table_name)
```

DESCRIPTION This routine reads from a file containing formatted strings. These formatted strings contain information on which schemes are needed within various modules. The field manager does not initialize any of those schemes however. It simply holds the information and is queried by the appropriate module.

INPUT

`table_name`	The name of the field table. The default name is field_table. [character, optional, dimension(len=128)]

OUTPUT

`nfields`	The number of fields. [integer]

b. Field_manager_end

```
call field_manager_end
```

DESCRIPTION This subroutine writes to the logfile that the user is exiting field_manager and changes the initialized flag to false.

c. Find_field_index

```
value= find_field_index ( model, field_name )
```

DESCRIPTION This function when passed a model number and a field name will return the index of the field within the field manager. This index can be used to access other information from the field manager.

INPUT

`model`	The number indicating which model is used. [integer]

d. Get_field_info

```
call get_field_info ( n,fld_type,fld_name,model,num_methods )
```

DESCRIPTION When passed an index, this routine will return the type of field, the name of the field, the model which the field is associated and the number of methods associated with the field.

INPUT

n	The field index. [integer]

OUTPUT

fld_type	The field type. [character, dimension(*)]
fld_name	The name of the field. [character, dimension(*)]
model	The number indicating which model is used. [integer]
num_methods	The number of methods. [integer]

e. Get_field_method

```
call get_field_method ( n,m,method )
```

DESCRIPTION This routine, when passed a field index and a method index will return the method text associated with the field(n) method(m).

INPUT

n	The field index. [integer]
m	The method index. [integer]

f. Get_field_methods

```
call get_field_methods ( n,methods )
```

DESCRIPTION When passed a field index, this routine will return the text associated with all the methods attached to the field.

INPUT

n	The field index. [integer]

g. Parse

```
number = parse (text, label, value)
```

DESCRIPTION Parse is an integer function that decodes values from a text string. The text string has the form: “label=list” where “label” is an arbitrary user defined label describing the values being decoded, and “list” is a list of one or more values separated by commas. The values may be integer, real, or character. Parse returns the number of values decoded.

INPUT

`text`	The text string from which the values will be parsed. [character(len=*)]
`label`	A label which describes the values being decoded. [character(len=*)]

OUTPUT

`value`	The value or values that have been decoded. [integer, real, character(len=*)]
`parse`	The number of values that have been decoded. This allows a user to define a large array and fill it partially with values from a list. This should be the size of the value array. [integer]

6.199.6 Data sets

None.

6.199.7 Error messages

NOTE in field_manager_init No field table available, so no fields are being registered. The field table does not exist.

FATAL in field_manager_init max fields exceeded Maximum number of fields for this module has been exceeded.

FATAL in field_manager_init Too many fields in tracer entry. There are more that 3 fields in the tracer entry. This is probably due to separating the parameters entry into multiple strings. The entry should look like “Type”,”Name”,”Control1=XXX,Control2=YYY” and not like “Type”,”Name”,”Control1=XXX”,”Control2=YYY”

FATAL in field_manager_init Maximum number of methods for field exceeded Maximum number of methods allowed for entries in the field table has been exceeded.

NOTE in field_manager_init field with identical name and model name duplicate found, skipping The name of the field and the model name are identical. Skipping that field.

FATAL in field_manager_init error reading field table There is an error in reading the field table.

FATAL in get_field_info invalid field index The field index is invalid because it is less than 1 or greater than the number of fields.

FATAL in get_field_method invalid field index The field index is invalid because it is less than 1 or greater than the number of fields.

FATAL in get_field_method invalid method index The method index is invalid because it is less than 1 or greater than the number of methods.

FATAL in get_field_methods invalid field index The field index is invalid because it is less than 1 or greater than the number of fields.

FATAL in get_field_methods method array too small The method array is smaller than the number of methods.

top

6.200 module horiz_interp_mod

6.200.1 Overview

Performs spatial interpolation between rectangular latitude/longitude grids.

The interpolation algorithm uses a scheme that conserves the area-weighed integral of the input field. The domain of the output field must lie within the domain of the input field.

6.201 Module fms_mod

6.201.1 Overview

The fms module provides routines that are commonly used by most FMS modules.

Here is a summary of the functions performed by routines in the fms module. 1. Output module version numbers to a common (`log`) file using a common format. 2. Open specific types of files common to many FMS modules. These include namelist files, restart files, and 32-bit IEEE data files. There also is a matching interface to close the files. If other file types are needed the `mpp_open` and `mpp_close` interfaces in module ` <http://www.gfdl.noaa.gov/fms-cgi-bin/cvsweb.cgi/FMS/shared/mpp/models/bgrid_solo/fms_src/shared/mpp/mpp_io.html>`__ must be used. 3. Read and write distributed data to simple native unformatted files. This type of file (called a restart file) is used to checkpoint model integrations for a subsequent restart of the run. 4. Time sections of code (using a wrapper for mpp_mod). 5. For convenience there are several routines published from the ` <http://www.gfdl.noaa.gov/fms-cgi-bin/cvsweb.cgi/FMS/shared/mpp/models/bgrid_solo/fms_src/shared/mpp/mpp.html>`__ module. These are routines for getting processor numbers, commonly used I/O unit numbers, and error handling.

6.201.2 Other modules used

```
mpp_mod
mpp_domains_mod
      mpp_io_mod
      fms_io_mod
```

6.201.3 Public interface

fms_init: Initializes the FMS module and also calls the initialization routines for all modules in the MPP package. Will be called automatically if the user does not call it.

fms_end: Calls the termination routines for all modules in the MPP package.

file_exist: Checks the existence of a given file name.

error_mesg: Print notes, warnings and error messages; terminates program for warning and error messages. (use error levels NOTE,WARNING,FATAL, see example below)

check_nml_error: Checks the iostat argument that is returned after reading a namelist and determines if the error code is valid.

write_version_number: Prints to the log file (or a specified unit) the (cvs) version id string and (cvs) tag name.

mpp_clock_init: Returns an identifier for performance timing a section of code (similar to mpp_clock_id).

lowercase: Convert character strings to all lower case.

uppercase: Convert character strings to all upper case.

string_array_index: match the input character string to a string in an array/list of character strings

monotonic_array: Determines if a real input array has monotonically increasing or decreasing values.

6.201.4 Public data

None.

6.201.5 Public routines

a. **Fms_init**

```
call fms_init ( )
```

DESCRIPTION Initialization routine for the fms module. It also calls initialization routines for the mpp, mpp_domains, and mpp_io modules. Although this routine will be called automatically by other fms_mod routines, users should explicitly call fms_init. If this routine is called more than once it will return silently. There are no arguments.

b. **Fms_end**

```
call fms_end ( )
```

DESCRIPTION Termination routine for the fms module. It also calls destructor routines for the mpp, mpp_domains, and mpp_io modules. If this routine is called more than once it will return silently. There are no arguments.

c. **File_exist**

```
file_exist ( file_name )
```

DESCRIPTION Checks the existence of the given file name. If the file_name string has zero length or the first character is blank return a false result.

INPUT

`file_name`	A file name (or path name) that is checked for existence. [character]

OUTPUT

	This function returns a logical result. If file_name exists the result is true, otherwise false is returned. If the length of character string "file_name" is zero or the first character is blank, then the returned value will be false. When reading a file, this function is often used in conjunction with routine open_file. []

d. Error_mesg

```
call error_mesg ( routine, message, level )
```

DESCRIPTION Print notes, warnings and error messages; and terminates the program for error messages. This routine is a wrapper around mpp_error, and is provided for backward compatibility. This module also publishes mpp_error, **users should try to use the mpp_error interface**.

INPUT

`routine`	Routine name where the warning or error has occurred. [character]
`message`	Warning or error message to be printed. [character]
`level`	Level of severity; set to NOTE, WARNING, or FATAL Termination always occurs for FATAL, never for NOTE, and is settable for WARNING (see namelist). [integer]

NOTE Examples:

```
use fms_mod, only: error_mesg, FATAL, NOTE
call error_mesg ('fms_mod', 'initialization not called', FATAL)
call error_mesg ('fms_mod', 'fms_mod message', NOTE)
```

e. Check_nml_error

```
check_nml_error ( iostat, nml_name )
```

DESCRIPTION The FMS allows multiple namelist records to reside in the same file. Use this interface to check the iostat argument that is returned after reading a record from the namelist file. If an invalid iostat value is detected this routine will produce a fatal error. See the NOTE below.

INPUT

`iostat`	The iostat value returned when reading a namelist record. [integer]
`nml_name`	The name of the namelist. This name will be printed if an error is encountered, otherwise the name is not used. [character]

OUTPUT

	This function returns the input iostat value (integer) if it is an allowable error code. If the iostat error code is not allowable, an error message is printed and the program terminated. [integer]

NOTE

Some compilers will return non-zero iostat values when reading through files with multiple namelist. This routine will try skip these errors and only terminate for true namelist errors.

Examples

The following example checks if a file exists, reads a namelist input from that file, and checks for errors in that namelist. When the correct namelist is read and it has no errors the routine check_nml_error will return zero and the while loop will exit. This code segment should be used to read namelist files.

```
  integer :: unit, ierr, io

  if ( file_exist('input.nml') ) then
      unit = open_namelist_file ( )
      ierr=1
      do while (ierr /= 0)
        read  (unit, nml=moist_processes_nml, iostat=io, end=10)
        ierr = check_nml_error(io,'moist_processes_nml')
      enddo
10    call close_file (unit)
  endif
```

f. Write_version_number

```
call write_version_number ( version [, tag, unit] )
```

DESCRIPTION Prints to the log file (stdlog) or a specified unit the (cvs) version id string and (cvs) tag name.

INPUT

version	string that contains routine name and version number. [character(len=*)]
tag	The tag/name string, this is usually the Name string returned by CVS when checking out the code. [character(len=*)]
unit	The Fortran unit number of an open formatted file. If this unit number is not supplied the log file unit number is used (stdlog). [integer]

g. Mpp_clock_init

```
id = mpp_clock_init ( name, level [, flags] )
```

DESCRIPTION Returns an identifier for performance timing sections of code. Should be used in conjunction with mpp_clock_begin and mpp_clock_end. For more details see the documentation for the MPP module and look at the example below.

INPUT

name	A unique name string given to the code segment to be timed. The length should not exceed 32 characters. [character]
level	Level of timing. When level > timing_level, which is set by namelist &fms_nml, an identifier of zero is returned. This will turn off performance timing for the code section. [integer]
flags	Use the flags published via the mpp_mod to control whether synchronization or extra detail is desired. (flags = MPP_CLOCK_SYNC, MPP_CLOCK_DETAILED) [integer]

OUTPUT

id	The identification index returned by mpp_clocks_id. A zero value is returned (turning clocks off) when input argument level > namelist variable timing_level. [integer]

NOTE

1.The MPP_CLOCK_SYNC flag should be used whenever possible. This flag causes mpp_sync to be called at the begin of a code segment, resulting in more accurate performance timings. **Do not use the MPP_CLOCK_SYNC flag for code sections that may not be called on all processors.**

2.There is some amount of coordination required throughout an entire program for consistency of the "timing levels". As a guideline the following levels may be used, with higher levels added as desired to specific component models.

```
level
      example code section
 1
      main program
 2
      components models
 3
      atmosphere dynamics or physics
```

Examples:

The mpp_clock_init interface should be used in conjunction with the mpp_mod interfaces mpp_clock_begin and mpp_clock_end. For example:

```
use fms_mod, only: mpp_clock_init, mpp_clock_begin, &
                             mpp_clock_end. MPP_CLOCK_SYNC
          integer :: id_mycode
          integer :: timing_level = 5

          id_mycode = mpp_clock_init ('mycode loop', timing_level, &
                                   flags=MPP_CLOCK_SYNC)
          call mpp_clock_begin (id_mycode)
                        :
                        :
           ~~ this code will be timed ~~
                        :
                        :
          call mpp_clock_end (id_mycode)
```

h. Lowercase

```
string = lowercase ( cs )
```

DESCRIPTION Converts a character string to all lower case letters. The characters "A-Z" are converted to "a-z", all other characters are left unchanged.

INPUT

cs	Character string that may contain upper case letters. [character(len=*), scalar]

OUTPUT

string	Character string that contains all lower case letters. The length of this string must be the same as the input string. [character(len=len(cs)), scalar]

i. Uppercase

```
string = uppercase ( cs )
```

DESCRIPTION Converts a character string to all upper case letters. The characters “a-z” are converted to “A-Z”, all other characters are left unchanged.

INPUT

cs	Character string that may contain lower case letters. [character(len=*), scalar]

OUTPUT

string	Character string that contains all upper case letters. The length of this string must be the same as the input string. [character(len=len(cs)), scalar]

j. String_array_index

```
string_array_index ( string, string_array [, index] )
```

DESCRIPTION Tries to find a match for a character string in a list of character strings. The match is case sensitive and disregards blank characters to the right of the string.

INPUT

string	Character string of arbitrary length. [character(len=*), scalar]
string_array	Array/list of character strings. [character(len=*), dimension(:)]

OUTPUT

index	The index of string_array where the first match was found. If no match was found then index = 0. []
found	If an exact match was found then TRUE is returned, otherwise FALSE is returned. [logical]

NOTE Examples

```
string = "def"
string_array = (/ "abcd", "def ", "fghi" /)
string_array_index ( string, string_array, index )
Returns: TRUE, index = 2
```

k. Monotonic_array

```
monotonic_array ( array [, direction] )
```

DESCRIPTION Determines if the real input array has monotonically increasing or decreasing values.

INPUT

array	An array of real values. If the size(array) < 2 this function assumes the array is not monotonic, no fatal error will occur. [real, dimension(:)]

OUTPUT

`direction`	If the input array is: >> monotonic (small to large) then direction = +1. >> monotonic (large to small) then direction = -1. >> not monotonic then direction = 0. [integer]
	If the input array of real values either increases or decreases monotonically then TRUE is returned, otherwise FALSE is returned. [logical]

6.201.6 Namelist

&fms_nml

`timing_level` The level of performance timing. If calls to the performance timing routines have been inserted into the code then code sections with a level <= timing_level will be timed. The resulting output will be printed to STDOUT. See the MPP module or mpp_clock_init for more details. [integer, default: 0] `read_all_pe` Read global data on all processors extracting local part needed (TRUE) or read global data on PE0 and broadcast to all PEs (FALSE). [logical, default: true] `warning_level` Sets the termination condition for the WARNING flag to interfaces error_mesg/mpp_error. set warning_level = 'fatal' (program crashes for warning messages) or 'warning' (prints warning message and continues). [character, default: 'warning'] `iospec_ieee32` iospec flag used with the open_ieee32_file interface. [character, default: '-F f77,cachea:48:1'] `stack_size` The size in words of the MPP user stack. If stack_size > 0, the following MPP routine is called: call mpp_set_stack_size (stack_size). If stack_size = 0 (default) then the default size set by mpp_mod is used. [integer, default: 0] `domains_stack_size` The size in words of the MPP_DOMAINS user stack. If domains_stack_size > 0, the following MPP_DOMAINS routine is called: call mpp_domains_set_stack_size (domains_stack_size). If domains_stack_size = 0 (default) then the default size set by mpp_domains_mod is used. [integer, default: 0]

6.201.7 Data sets

None.

6.201.8 Error messages

FATAL in fms_init invalid entry for namelist variable warning_level The namelist variable warning_level must be either 'fatal' or 'warning' (case-insensitive).

FATAL in file_exist set_domain not called Before calling write_data you must first call set_domain with domain2d data type associated with the distributed data you are writing.

FATAL in check_nml_error while reading namelist …., iostat = #### There was an error message reading the namelist specified. Carefully examine all namelist variables for misspellings of type mismatches (e.g., integer vs. real).

6.201.9 References

None.

6.201.10 Compiler specifics

None.

6.201.11 Precompiler options

None.

6.201.12 Loader options

None.

6.201.13 Test PROGRAM

None.

6.201.14 Notes

1) If the **MPP** or **MPP_DOMAINS** stack size is exceeded the program will terminate after printing the required size.
2) When running on a very small number of processors or for high resolution models the default domains_stack_size will probably be insufficient.

6.202 module constants_mod

6.202.1 Overview

Defines useful constants for Earth in mks units.

Constants are defined as real parameters.They are accessed through the "use" statement. While the local name of constant may be changed, their values can not be redefined.

6.202.2 Other modules used

```
fms_mod
```

6.202.3 Public interface

```
use constants_mod [, only:  constants_init ]
```

constants_init: A optional initialization routine. The only purpose of this routine is to write the version and tag name information to the log file.

6.202.4 Public data

Name	Type	Value	Units	Description
RADIUS	real	6376.e3	meters	radius of the earth
OMEGA	real	7.292e-5	1/sec	rotation rate of the planet (earth)
GRAV	real	9.80	m/s2	acceleration due to gravity
RDGAS	real	287.04	J/kg/deg	gas constant for dry air
KAPPA	real	2./7.		RDGAS / CP
CP	real	RDGAS/KAPPA	J/kg/deg	specific heat capacity of dry air at constant pressure
RVGAS	real	461.50	J/Kg/deg	gas constant for water vapor
DENS_H2O	real	1000.	Kg/m3	density of liquid water
HLV	real	2.500e6	J/Kg	latent heat of evaporation
HLF	real	3.34e5	J/kg	latent heat of fusion
HLS	real	2.834e6	J/Kg	latent heat of sublimation
TFREEZE	real	273.16	deg K	temp where fresh water freezes
STEFAN	real	5.6734e-8	(W/m2/deg4	Stefan-Boltzmann constant
VONKARM	real	0.40	—	Von Karman constant
PI	real	3.14159265358979323846		is it enough?

6.202.5 Public routines

a. **Constants_init**

```
call constants_init
```

DESCRIPTION The only purpose of this routine is to write the version and tag name information to the log file. This routine does not have to be called. If it is called more than once or called from other than the root PE it will return silently. There are no arguments.

6.202.6 Data sets

None.

6.202.7 Error messages

None.

6.202.8 References

None.

6.202.9 Compiler specifics

None.

6.202.10 Precompiler options

None.

6.202.11 Loader options

None.

6.202.12 Test PROGRAM

None.

6.202.13 Notes

<B>NOTES ON USAGE:</B> All constants have been declared as type REAL, PARAMETER. The value a constant can not be changed in a users program. New constants can be defined in terms of values from the constants module using a parameter statement.<br><br> The name given to a particular constant may be changed.<br><br> Constants can be used on the right side on an assignment statement (their value can not be reassigned). <B>EXAMPLES:</B>

```
use constants_mod, only:  TFREEZE, grav_new => GRAV
real, parameter :: grav_inv = 1.0 / grav_new
tempc(:,:,:) = tempk(:,:,:) - TFREEZE
geopotential(:,:) = height(:,:) * grav_new
```

6.203 module platform_mod

6.203.1 Overview

`platform_mod` is a module that provides public entities whose value may depend on the operating system and compiler.

The combination of operating system and compiler is referred to here as the *platform*. `platform_mod` provides:

```
integer, parameter :: r8_kind, r4_kind, &
                      c8_kind, c4_kind, &
                      l8_kind, l4_kind, &
                      i8_kind, i4_kind, i2_kind
```

by use association. These are set to the appropriate KIND values that provide 4 and 8-byte versions of the fortran intrinsic types. The actual numerical value of the KIND may differ depending on platform.

These should be used when the actually bytelength of the variable is important. To specify numerical precision, Fortran recommends the use of the intrinsics `SELECTED_REAL_KIND` and `SELECTED_INT_KIND`.

6.203.2 Other other modules used

None.

6.203.3 Public interface

None.

6.203.4 Public data

None.

6.203.5 Public routines

None.

6.203.6 Namelist

None.

6.203.7 Compiling and linking source

Any module or program unit using `platform_mod` must contain the line

```
use platform_mod
```

6.203.8 Portability

None.

6.203.9 Acquiring source

The `platform_mod` source consists of the main source file `platform.F90` and also requires the following include files:

```
os.h
```

GFDL users can check it out of the main CVS repository as part of the `platform` CVS module. The current public tag is `fez`. Public access to the GFDL CVS repository will soon be made available.

6.203.10 Notes

None.

6.204 Module Utilities

6.205 module tracer_manager_mod

6.205.1 Overview

Code to manage the simple addition of tracers to the FMS code. This code keeps track of the numbers and names of tracers included in a tracer table.

This code is a grouping of calls which will allow the simple introduction of tracers into the FMS framework. It is designed to allow users of a variety of component models interact easily with the dynamical core of the model. In calling the tracer manager routines the user must provide a parameter identifying the model that the user is working with. This parameter is defined within field_manager as MODEL_X where X is one of [ATMOS, OCEAN, LAND, ICE]. In many of these calls the argument list includes model and tracer_index. These are the parameter corresponding to the component model and the tracer_index N is the Nth tracer within the component model. Therefore a call with MODEL_ATMOS and 5 is different from a call with MODEL_OCEAN and 5.

6.205.2 Other modules used

```
mpp_mod
       mpp_io_mod
          fms_mod
field_manager_mod
```

6.205.3 Public interface

```
use tracer_manager_mod [, only:  tracer_manager_init,
                                 register_tracers,
                                 get_number_tracers,
                                 get_tracer_indices,
                                 get_tracer_index,
                                 assign_tracer_field,
                                 tracer_manager_end,
                                 get_tracer_field,
                                 get_tracer_tlevels,
                                 get_tracer_tendency,
                                 get_tracer_names,
                                 get_family_name,
                                 check_if_prognostic,
                                 find_family_members,
                                 add_members_to_family,
                                 split_family_into_members,
                                 set_tracer_profile,
                                 query_method,
                                 query_combined,
                                 set_tracer_atts ]
```

tracer_manager_init: Routine to initialize the tracer manager

register_tracers: A routine to register the tracers included in a component model.

get_number_tracers: A routine to return the number of tracers included in a component model.

get_tracer_indices: Routine to return the component model tracer indices as defined within the tracer manager.

get_tracer_index: Function which returns the number assigned to the tracer name.

assign_tracer_field: Routine to point the appropriate field within the tracer_type to the appropriate field within the component model.

tracer_manager_end: Routine to write to the log file that the tracer manager is ending.

get_tracer_field: A function to retrieve the present timestep data.

get_tracer_tlevels: A function to retrieve the three time levels data.

get_tracer_tendency: A function to retrieve the tendency data.

get_tracer_names: Routine to find the names associated with a tracer number.

get_family_name: Routine to return the family name for tracer n.

check_if_prognostic: Function to see if a tracer is prognostic or diagnostic.

find_family_members: Subroutine to find which tracers are members of family family_name.

add_members_to_family: Routine to sum up the members of a family of tracers so that they may be advected and diffused as one tracer.

split_family_into_members: Subroutine that sets the present value of the member of a tracer family according to the fraction of the family that it was in the previous step.

set_tracer_profile: Subroutine to set the tracer field to the wanted profile.

query_method: A function to query the "methods" associated with each tracer.

query_combined: A function to query whether families of tracers have been combined already.

set_tracer_atts: A subroutine to allow the user set the tracer longname and units from the tracer initialization routine.

6.205.4 Public data

None.

6.205.5 Public routines

a. **Tracer_manager_init**

```
call tracer_manager_init
```

DESCRIPTION This routine writes the version and tagname to the logfile and sets the module initialization flag.

b. **Register_tracers**

```
call register_tracers (model, num_tracers,num_prog,num_diag,num_family)
```

DESCRIPTION This routine returns the total number of valid tracers, the number of prognostic and diagnostic tracers and the number of families of tracers.

INPUT

`model`	A parameter to identify which model is being used. [integer]

OUTPUT

`num_tracers`	The total number of valid tracers within the component model. [integer]
`num_prog`	The number of prognostic tracers within the component model. [integer]
`num_diag`	The number of diagnostic tracers within the component model. [integer]
`num_family`	The number of family tracers within the component model. [integer]

c. Get_number_tracers

```
call get_number_tracers (model, num_tracers,num_prog,num_diag,num_family)
```

DESCRIPTION This routine returns the total number of valid tracers, the number of prognostic and diagnostic tracers and the number of families of tracers.

INPUT

`model`	A parameter to identify which model is being used. [integer]

OUTPUT

`num_tracers`	The total number of valid tracers within the component model. [integer, optional]
`num_prog`	The number of prognostic tracers within the component model. [integer, optional]
`num_diag`	The number of diagnostic tracers within the component model. [integer, optional]
`num_family`	The number of family tracers within the component model. [integer, optional]

d. Get_tracer_indices

```
call get_tracer_indices (model, ind, prog_ind, diag_ind, fam_ind)
```

DESCRIPTION If several models are being used or redundant tracers have been written to the tracer_table, then the indices in the component model and the tracer manager may not have a one to one correspondence. Therefore the component model needs to know what index to pass to calls to tracer_manager routines in order that the correct tracer information be accessed.

INPUT

`model`	A parameter to identify which model is being used. [integer]

OUTPUT

`ind`	An array containing the tracer manager defined indices for all the tracers within the component model. [integer, optional, dimension(:)]
`prog_ind`	An array containing the tracer manager defined indices for the prognostic tracers within the component model. [integer, optional, dimension(:)]
`diag_ind`	An array containing the tracer manager defined indices for the diagnostic tracers within the component model. [integer, optional, dimension(:)]
`fam_ind`	An array containing the tracer manager defined indices for the family tracers within the component model. [integer, optional, dimension(:)]

e. Get_tracer_index

```
value= get_tracer_index (model, name, indices, verbose)
```

DESCRIPTION This is a function which returns the index, as implied within the component model.

INPUT

`model`	A parameter to identify which model is being used. [integer]
`name`	The name of the tracer (as assigned in the field table). [character]
`indices`	An array of the component model indices. This array can be found by calling get_tracer_indices. [integer, optional, dimension(:)]
`verbose`	A flag to allow the message saying that a tracer with this name has not been found. This should only be used for debugging purposes. [logical, optional]

OUTPUT

`get_tracer_index`	The index of the tracer named "name". If indices is passed then the result is the array index which corresponds to tracer named "name". [integer]

f. Assign_tracer_field

```
call assign_tracer_field (model,index, data, data_tlevels, tendency)
```

DESCRIPTION The generality provided here is that one can point the three dimensional tracer field at either a two time level scheme [data and tendency] or a three time level scheme [data_tlevels]. The tracer manager points the appropriate tracer_type field at the data supplied from the component model.

INPUT

`model`	A parameter representing the component model in use. [integer]
`index`	The tracer number that you wish to assign a tracer field for. [integer]
`data`	The 3D field that is associated with the present time step in the component model. [real, target, optional, dimension(:,:,:)]
`tendency`	The 3D field that is associated with the tendency time step in the component model. [real, target, optional, dimension(:,:,:)]
`data_tlevels`	The 4D field that is associated with the tracer field in the component model. [real, target, optional, dimension(:,:,:,:)]

g. Tracer_manager_end

```
call tracer_manager_end
```

DESCRIPTION Routine to write to the log file that the tracer manager is ending.

h. Get_tracer_field

```
array= get_tracer_field (model, tracer_index)
```

DESCRIPTION Function to point to the 3D field associated with a tracer.

INPUT

`model`	A parameter representing the component model in use. [integer]
`tracer_index`	The tracer number within the component model. [integer]

OUTPUT

`data`	The tracer field is returned in this array. [real, pointer, dimension(:,:,:)]

i. Get_tracer_tlevels

```
array= get_tracer_tlevels (model, tracer_index)
```

DESCRIPTION Function to point to the 4D field associated with a tracer.

INPUT

`model`	A parameter representing the component model in use. [integer]
`tracer_index`	The tracer number within the component model. [integer]

OUTPUT

`data`	The tracer field is returned in this array. [real, pointer, dimension(:,:,:,:)]

j. Get_tracer_tendency

```
array= get_tracer_tendency (model, tracer_index)
```

DESCRIPTION Function to point to the 3D field associated with a tracer.

INPUT

`model`	A parameter representing the component model in use. [integer]
`tracer_index`	The tracer number within the component model. [integer]

OUTPUT

`data`	The tracer tendency field is returned in this array. [real, pointer, dimension(:,:,:)]

k. Get_tracer_names

```
call get_tracer_names (model,n,name,longname, units)
```

DESCRIPTION This routine can return the name, long name and units associated with a tracer.

INPUT

model	A parameter representing the component model in use. [integer]
n	Tracer number. [integer]

OUTPUT

name	Field name associated with tracer number. [character]
longname	The long name associated with tracer number. [character, optional]
units	The units associated with tracer number. [character, optional]

l. Get_family_name

```
call get_family_name (model,n,name)
```

DESCRIPTION You may wish to use this routine to retrieve the name of the family that a tracer belongs to.

INPUT

model	A parameter representing the component model in use. [integer]
n	Tracer number that you want the family name for. [integer]

OUTPUT

name	The family name. [character]

m. Check_if_prognostic

```
logical = check_if_prognostic (model, n)
```

DESCRIPTION All tracers are assumed to be prognostic when read in from the field_table However a tracer can be changed to a diagnostic tracer by adding the line “tracer_type”,”diagnostic” to the tracer description in field_table.

INPUT

model	A parameter representing the component model in use. [integer]
n	Tracer number that you want the family name for. [integer]

OUTPUT

check_if_prognostic	A logical flag set TRUE if the tracer is prognostic. [logical]

n. Find_family_members

```
call find_family_members (model, family_name,is_family_member)
```

DESCRIPTION Subroutine to find which tracers are members of family family_name. This will return a logical array where the array positions corresponding to the tracer numbers for family members are set .TRUE.

INPUT

`model`	A parameter representing the component model in use. [integer]
`family_name`	The family name of the members one is seeking. [character]

OUTPUT

`is_family_member`	A logical array where the tracer number is used as the index to signify which tracer is part of the family. i.e. If tracers 1, 3, and 7 are part of the same family then is_family_member(1), is_family_member(3), and is_family_member(7) are set TRUE. [logical, dimension(:)]

o. Add_members_to_family

```
call add_members_to_family (model,family_name, cur, prev, next)
```

DESCRIPTION Routine to sum up the members of a family of tracers so that they may be advected and diffused as one tracer. This should only be used in conjunction with split_family_into_members and should be placed before the advection scheme is called.

INPUT

`model`	A parameter representing the component model in use. [integer]
`n`	Tracer number. [integer]
`cur`	Array index for the current time step. This is only of use with a three timestep model. [integer, optional]
`prev`	Array index for the previous time step. This is only of use with a three timestep model. [integer, optional]
`next`	Array index for the next time step. This is only of use with a three timestep model. [integer, optional]

NOTE This should be used with extreme caution. Unless the family member distributions are similar to each other spatially, advection as one tracer and subsequent splitting will result in a different result to advecting each tracer separately. The user should understand the possible repercussions of this before using it.

p. Split_family_into_members

```
call split_family_into_members (model,family_name,cur,prev,next)
```

DESCRIPTION Subroutine that sets the present value of the member of a tracer family according to the fraction of the family that it was in the previous step. This splits the transported family into the constituent members. This should only be used in conjunction with *add_members_to_family* and should be placed after the advection scheme is called.

INPUT

model	A parameter representing the component model in use. [integer]
family_name	The name of the family of tracers that you would like to split up. [character]
cur	Array index for the current time step. This is only of use with a three timestep model. [integer, optional]
prev	Array index for the previous time step. This is only of use with a three timestep model. [integer, optional]
next	Array index for the next time step. This is only of use with a three timestep model. [integer, optional]

NOTE This should be used with extreme caution. Unless the family member distributions are similar to each other spatially, advection as one tracer and subsequent splitting will result in a different result to advecting each tracer separately. The user should understand the possible repercussions of this before using it.

q. Set_tracer_profile

```
call set_tracer_profile (model, n, surf_value, multiplier)
```

DESCRIPTION If the profile type is ‘fixed’ then the tracer field values are set equal to the surface value. If the profile type is ‘profile’ then the top/bottom of model and surface values are read and an exponential profile is calculated, with the profile being dependent on the number of levels in the component model. This should be called from the part of the dynamical core where tracer restarts are called in the event that a tracer restart file does not exist. This can be activated by adding a method to the field_table e.g. “profile_type”,”fixed”,”surface_value = 1e-12” would return values of surf_value = 1e-12 and a multiplier of 1.0 One can use these to initialize the entire field with a value of 1e-12. “profile_type”,”profile”,”surface_value = 1e-12, top_value = 1e-15” In a 15 layer model this would return values of surf_value = 1e-12 and multiplier = 0.6309573 i.e 1e-15 = 1e-12*(0.6309573^15) In this case the model should be MODEL_ATMOS as you have a “top” value. If you wish to initialize the ocean model, one can use bottom_value instead of top_value.

INPUT

model	A parameter representing the component model in use. [integer]
n	Tracer number. [integer]

OUTPUT

surf_value	The surface value that will be initialized for the tracer [real]
multiplier	The vertical multiplier for the tracer Level(k-1) = multiplier * Level(k) [real]

r. Query_method

```
logical = query_method (method_type, model, n, name, control)
```

DESCRIPTION A function to query the "methods" associated with each tracer. The "methods" are the parameters of the component model that can be adjusted by user by placing formatted strings, associated with a particular tracer, within the field table. These methods can control the advection, wet deposition, dry deposition or initial profile of the tracer in question. Any parametrization can use this function as long as a routine for parsing the name and control strings are provided by that routine.

INPUT

`method_type`	The method that is being requested. [character]
`model`	A parameter representing the component model in use. [integer]
`n`	Tracer number that you want the family name for. [integer]

OUTPUT

`name`	A string containing the modified name to be used with method_type. i.e. "2nd_order" might be the default for advection. One could use "4th_order" here to modify that behaviour. [character]
`control`	A string containing the modified parameters that are associated with the method_type and name. [character, optional]
`query_method`	A flag to show whether method_type exists with regard to tracer n. If method_type is not present then one must have default values. [logical]

NOTE

At present the tracer manager module allows the initialization of a tracer profile if a restart does not exist for that tracer. Options for this routine are as follows

Tracer profile setup

```
=================================================================
|method_type |method_name |method_control |
=================================================================
|profile_type |fixed |surface_value = X | |profile_type |profile
|surface_value = X, top_value = Y |(atmosphere) |profile_type
|profile |surface_value = X, bottom_value = Y |(ocean)
=================================================================
```

s. Query_combined

```
logical = query_combined (model, index)
```

DESCRIPTION A function to query whether families of tracers have been combined already. This function should only be used in conjunction with add_members_to_family and split_family_into_members.

INPUT

model	A parameter representing the component model in use. [integer]
index	Tracer number. [integer]

OUTPUT

query_combined	A flag to show whether the tracer family has been combined. [logical]

t. **Set_tracer_atts**

```
call set_tracer_atts (model, name, longname, units)
```

DESCRIPTION A function to allow the user set the tracer longname and units from the tracer initialization routine. It seems sensible that the user who is coding the tracer code will know what units they are working in and it is probably safer to set the value in the tracer code rather than in the field table.

INPUT

model	A parameter representing the component model in use. [integer]
name	Tracer name. [character]

OUTPUT

longname	A string describing the longname of the tracer for output to NetCDF files [character, optional]
units	A string describing the units of the tracer for output to NetCDF files [character, optional]
set_tracer_atts	A flag to show that [character, optional]

6.205.6 Data sets

None.

6.205.7 Error messages

FATAL in register_tracers invalid model type The index for the model type is invalid.

NOTE in register_tracers No tracers are available to be registered. No tracers are available to be registered. This means that the field table does not exist or is empty.

FATAL in register_tracers MAX_TRACER_FIELDS exceeded The maximum number of tracer fields has been exceeded.

NOTE in register_tracers There is only 1 tracer for tracer family X. Making an orphan. A tracer has been given a family name but that family has only this member. Therefore it should be an orphan.

FATL in register_tracers MAX_TRACER_FIELDS needs to be increased The number of tracer fields has exceeded the maximum allowed. The parameter MAX_TRACER_FIELDS needs to be increased.

FATAL in get_number_tracers Model number is invalid. The index of the component model is invalid.

Fatal in get_tracer_indices index array size too small in get_tracer_indices The global index array is too small and cannot contain all the tracer numbers.

FATAL in get_tracer_indices family array size too small in get_tracer_indices The family index array is too small and cannot contain all the tracer numbers.

FATAL in get_tracer_indices prognostic array size too small in get_tracer_indices The prognostic index array is too small and cannot contain all the tracer numbers.

FATAL in get_tracer_indices diagnostic array size too small in get_tracer_indices The diagnostic index array is too small and cannot contain all the tracer numbers.

NOTE in get_tracer_index tracer with this name not found: X

FATAL in assign_tracer_field invalid index The index that has been passed to this routine is invalid.

FATAL in assign_tracer_field At least one of data, data_tlevels or tendency must be passed in here. At least one of data, data_tlevels or tendency must be passed to assign_tracer_field Otherwise there is not much point in calling this routine.

FATAL in get_tracer_field invalid index The index that has been passed to this routine is invalid. Check the index that is being passed corresponds to a valid tracer name.

FATAL in get_tracer_field invalid index The index that has been passed to this routine is invalid. Check the index that is being passed corresponds to a valid tracer name.

FATAL in get_tracer_field tracer field array not allocated The tracer array has not been allocated. This means that a call to assign_tracer_field is absent in the code.

FATAL in get_tracer_tlevels invalid index The index that has been passed to this routine is invalid. Check the index that is being passed corresponds to a valid tracer name.

FATAL in get_tracer_tlevels invalid index The index that has been passed to this routine is invalid. Check the index that is being passed corresponds to a valid tracer name.

FATAL in get_tracer_tlevels tracer field array not allocated The tracer array has not been allocated. This means that a call to assign_tracer_field is absent in the code.

FATAL in get_tracer_tendency invalid index The index that has been passed to this routine is invalid. Check the index that is being passed corresponds to a valid tracer name.

FATAL in get_tracer_tendency invalid index The index that has been passed to this routine is invalid. Check the index that is being passed corresponds to a valid tracer name.

FATAL in get_tracer_tendency tracer tendency field array not allocated The tracer array has not been allocated. This means that a call to assign_tracer_field is absent in the code.

6.206 module mpp_domains_mod

6.206.1 Overview

`mpp_domains_mod` is a set of simple calls for domain decomposition and domain updates on rectilinear grids. It requires the module *module mpp_mod*, upon which it is built.

Scalable implementations of finite-difference codes are generally based on decomposing the model domain into subdomains that are distributed among processors. These domains will then be obliged to exchange data at their boundaries if data dependencies are merely nearest-neighbour, or may need to acquire information from the global domain if there are extended data dependencies, as in the spectral transform. The domain decomposition is a key operation in the development of parallel codes.

`mpp_domains_mod` provides a domain decomposition and domain update API for *rectilinear* grids, built on top of the *module mpp_mod* API for message passing. Features of `mpp_domains_mod` include:

Simple, minimal API, with free access to underlying API for more complicated stuff.

Design toward typical use in climate/weather CFD codes.

6.206.2 Domains

I have assumed that domain decomposition will mainly be in 2 horizontal dimensions, which will in general be the two fastest-varying indices. There is a separate implementation of 1D decomposition on the fastest-varying index, and 1D decomposition on the second index, treated as a special case of 2D decomposition, is also possible. We define *domain* as the grid associated with a *task*. We define the *compute domain* as the set of gridpoints that are computed by a task, and the *data domain* as the set of points that are required by the task for the calculation. There can in general be more than 1 task per PE, though often the number of domains is the same as the processor count. We define the *global domain* as the global computational domain of the entire model (i.e, the same as the computational domain if run on a single processor). 2D domains are defined using a derived type `domain2D`, constructed as follows (see comments in code for more details):

```
type, public :: domain_axis_spec
    private
    integer :: begin, end, size, max_size
    logical :: is_global
    end type domain_axis_spec
    type, public :: domain1D
    private
    type(domain_axis_spec) :: compute, data, global, active
    logical :: mustputb, mustgetb, mustputf, mustgetf, folded
    type(domain1D), pointer, dimension(:) :: list
    integer :: pe              !PE to which this domain is assigned
    integer :: pos
    end type domain1D
domaintypes of higher rank can be constructed from type domain1D
typically we only need 1 and 2D, but could need higher (e.g 3D LES)
some elements are repeated below if they are needed once per domain
    type, public :: domain2D
    private
    type(domain1D) :: x
    type(domain1D) :: y
    type(domain2D), pointer, dimension(:) :: list
    integer :: pe              !PE to which this domain is assigned
    integer :: pos
    end type domain2D
    type(domain1D), public :: NULL_DOMAIN1D
    type(domain2D), public :: NULL_DOMAIN2D
```

The `domain2D` type contains all the necessary information to define the global, compute and data domains of each task, as well as the PE associated with the task. The PEs from which remote data may be acquired to update the data domain are also contained in a linked list of neighbours.

6.206.3 Other modules used

```
mpp_mod
```

6.206.4 Public interface

```
use mpp_domains_mod [, only:  mpp_define_domains,
                              mpp_update_domains,
                              mpp_redistribute,
                              mpp_global_field,
                              mpp_global_max,
                              mpp_global_sum,
                              operator,
                              mpp_get_compute_domain,
                              mpp_get_compute_domains,
                              mpp_get_data_domain,
                              mpp_get_global_domain,
                              mpp_define_layout,
                              mpp_get_pelist,
                              mpp_get_layout,
                              mpp_domains_init,
                              mpp_domains_set_stack_size,
                              mpp_domains_exit,
                              mpp_get_domain_components ]
```

mpp_define_domains: Set up a domain decomposition.

mpp_update_domains: Halo updates.

mpp_redistribute: Reorganization of distributed global arrays.

mpp_global_field: Fill in a global array from domain-decomposed arrays.

mpp_global_max: Global max/min of domain-decomposed arrays.

mpp_global_sum: Global sum of domain-decomposed arrays.

operator: Equality/inequality operators for domaintypes.

mpp_get_compute_domain: These routines retrieve the axis specifications associated with the compute domains.

mpp_get_compute_domains: Retrieve the entire array of compute domain extents associated with a decomposition.

mpp_get_data_domain: These routines retrieve the axis specifications associated with the data domains.

mpp_get_global_domain: These routines retrieve the axis specifications associated with the global domains.

mpp_define_layout: Retrieve layout associated with a domain decomposition.

mpp_get_pelist: Retrieve list of PEs associated with a domain decomposition.

mpp_get_layout: Retrieve layout associated with a domain decomposition.

mpp_domains_init: Initialize domain decomp package.

mpp_domains_set_stack_size: Set user stack size.

mpp_domains_exit: Exit `mpp_domains_mod`.

mpp_get_domain_components: Retrieve 1D components of 2D decomposition.

6.206.5 Public data

None.

6.206.6 Public routines

a. **Mpp_define_domains**

```
call mpp_define_domains ( global_indices, ndivs, domain, & pelist, flags, halo,␣
↪extent, maskmap )
```

```
call mpp_define_domains ( global_indices, layout, domain, pelist, & xflags,␣
↪yflags, xhalo, yhalo, & xextent, yextent, maskmap, name )
```

DESCRIPTION There are two forms for the `mpp_define_domains` call. The 2D version is generally to be used but is built by repeated calls to the 1D version, also provided.

INPUT

`global_indices`	Defines the global domain. [integer, dimension(2)] [integer, dimension(4)]
`ndivs`	Is the number of domain divisions required. [integer]
`pelist`	List of PEs to which the domains are to be assigned. [integer, dimension(0:)] [integer, dimension(0:)]
`flags`	An optional flag to pass additional information about the desired domain topology. Useful flags in a 1D decomposition include `GLOBAL_DATA_DOMAIN` and `CYCLIC_GLOBAL_DOMAIN`. Flags are integers: multiple flags may be added together. The flag values are public parameters available by use association. [integer]
`halo`	Width of the halo. [integer]
`extent`	Normally `mpp_define_domains` attempts an even division of the global domain across `ndivs` domains. The `extent` array can be used by the user to pass a custom domain division. The `extent` array has `ndivs` elements and holds the compute domain widths, which should add up to cover the global domain exactly. [integer, dimension(0:)]
`maskmap`	Some divisions may be masked (`maskmap=.FALSE.`) to exclude them from the computation (e.g for ocean model domains that are all land). The `maskmap` array is dimensioned `ndivs` and contains `.TRUE.` values for any domain that must be *included* in the computation (default all). The `pelist` array length should match the number of domains included in the computation. [logical, dimension(0:)] [logical, dimension(:,:)]
`layout`	[integer, dimension(2)]
`xflags`, `yflags`	[integer]
`xhalo`, `yhalo`	[integer]
`xextent`, `yextent`	[integer, dimension(0:)]
`name`	[character(len=*)]

INPUT/OUTPUT

`domain`	Holds the resulting domain decomposition. [type(domain1D)] [type(domain2D)]

NOTE

For example:

```
call mpp_define_domains( (/1,100/), 10, domain, &
     flags=GLOBAL_DATA_DOMAIN+CYCLIC_GLOBAL_DOMAIN, halo=2 )
```

defines 10 compute domains spanning the range [1,100] of the global domain. The compute domains are non-overlapping blocks of 10. All the data domains are global, and with a halo of 2 span the range [-1:102]. And since the global domain has been declared to be cyclic, `domain(9)%next => domain(0)` and `domain(0)%prev => domain(9)`. A field is allocated on the data domain, and computations proceed on the compute domain. A call to ` <#mpp_update_domains>`__ would fill in the values in the halo region:

```
call mpp_get_data_domain( domain, isd, ied ) !returns -1 and 102
    call mpp_get_compute_domain( domain, is, ie ) !returns (1,10) on PE 0 ...
    allocate( a(isd:ied) )
    do i = is,ie
       a(i) = <perform computations>
    end do
    call mpp_update_domains( a, domain )
```

The call to `mpp_update_domains` fills in the regions outside the compute domain. Since the global domain is cyclic, the values at `i=(-1,0)` are the same as at `i=(99,100)`; and `i=(101,102)` are the same as `i=(1,2)`.

The 2D version is just an extension of this syntax to two dimensions.

The 2D version of the above should generally be used in codes, including 1D-decomposed ones, if there is a possibility of future evolution toward 2D decomposition. The arguments are similar to the 1D case, except that now we have optional arguments `flags`, `halo`, `extent` and `maskmap` along two axes.

`flags` can now take an additional possible value to fold one or more edges. This is done by using flags `FOLD_WEST_EDGE`, `FOLD_EAST_EDGE`, `FOLD_SOUTH_EDGE` or `FOLD_NORTH_EDGE`. When a fold exists (e.g cylindrical domain), vector fields reverse sign upon crossing the fold. This parity reversal is performed only in the vector version of ` <#mpp_update_domains>`__. In addition, shift operations may need to be applied to vector fields on staggered grids, also described in the vector interface to `mpp_update_domains`.

`name` is the name associated with the decomposition, e.g `'Ocean model'`. If this argument is present, `mpp_define_domains` will print the domain decomposition generated to `stdlog`.

Examples:

```
call mpp_define_domains( (/1,100,1,100/), (/2,2/), domain, xhalo=1 )
```

will create the following domain layout:

```
                     |---------|-----------|-----------|-------------|
                     |domain(1)|domain(2)  |domain(3)  |domain(4)    |
|--------------------|---------|-----------|-----------|-------------|
|Compute domain|1,50,1,50|51,100,1,50|1,50,51,100|51,100,51,100|
```

(continues on next page)

(continued from previous page)

```
|--------------|---------|-----------|-----------|-------------|
|Data domain   |0,51,1,50|50,101,1,50|0,51,51,100|50,101,51,100|
|--------------|---------|-----------|-----------|-------------|
```

Again, we allocate arrays on the data domain, perform computations on the compute domain, and call `mpp_update_domains` to update the halo region.

If we wished to perfom a 1D decomposition along `Y` on the same global domain, we could use:

```
call mpp_define_domains( (/1,100,1,100/), layout=(/4,1/), domain, xhalo=1 )
```

This will create the following domain layout:

```
               |----------|-----------|-----------|------------|
               |domain(1) |domain(2)  |domain(3)  |domain(4)   |
|--------------|----------|-----------|-----------|------------|
|Compute domain|1,100,1,25|1,100,26,50|1,100,51,75|1,100,76,100|
|--------------|----------|-----------|-----------|------------|
|Data domain   |0,101,1,25|0,101,26,50|0,101,51,75|1,101,76,100|
|--------------|----------|-----------|-----------|------------|
```

b. Mpp_update_domains

```
call mpp_update_domains ( field, domain, flags )
```

```
call mpp_update_domains ( fieldx, fieldy, domain, flags, gridtype )
```

DESCRIPTION `mpp_update_domains` is used to perform a halo update of a domain-decomposed array on each PE. `MPP_TYPE_` can be of type `complex`, `integer`, `logical` or `real`; of 4-byte or 8-byte kind; of rank up to 5. The vector version (with two input data fields) is only present for `real` types. For 2D domain updates, if there are halos present along both `x` and `y`, we can choose to update one only, by specifying `flags=XUPDATE` or `flags=YUPDATE`. In addition, one-sided updates can be performed by setting `flags` to any combination of `WUPDATE`, `EUPDATE`, `SUPDATE` and `NUPDATE`, to update the west, east, north and south halos respectively. Any combination of halos may be used by adding the requisite flags, e.g: `flags=XUPDATE+SUPDATE` or `flags=EUPDATE+WUPDATE+SUPDATE` will update the east, west and south halos. If a call to `mpp_update_domains` involves at least one E-W halo and one N-S halo, the corners involved will also be updated, i.e, in the example above, the SE and SW corners will be updated. If `flags` is not supplied, that is equivalent to `flags=XUPDATE+YUPDATE`. The vector version is passed the `x` and `y` components of a vector field in tandem, and both are updated upon return. They are passed together to treat parity issues on various grids. For example, on a cubic sphere projection, the `x` and `y` components may be interchanged when passing from an equatorial cube face to a polar face. For grids with folds, vector components change sign on crossing the fold. Special treatment at boundaries such as folds is also required for staggered grids. The following types of staggered grids are recognized: 1) `AGRID`: values are at grid centers. 2) `BGRID_NE`: vector fields are at the NE vertex of a grid cell, i.e: the array elements `u(i,j)` and `v(i,j)` are actually at (i+½,j+½) with respect to the grid centers. 3) `BGRID_SW`: vector fields are at the SW vertex of a grid cell, i.e: the array elements `u(i,j)` and `v(i,j)` are actually at (i-½,j-½) with respect to the grid centers. 4) `CGRID_NE`: vector fields are at the N and E faces of a grid cell, i.e: the array elements `u(i,j)` and `v(i,j)` are actually at (i+½,j) and (i,j+½) with respect to the grid centers. 5) `CGRID_SW`: vector fields are at the S and W faces of a grid cell, i.e: the array elements `u(i,j)` and `v(i,j)` are actually at (i-½,j) and (i,j-½) with respect to the grid centers. The gridtypes listed above are all available by use association as integer parameters. The scalar version of `mpp_update_domains` assumes that the values of a scalar field are always at `AGRID` locations, and no special boundary treatment is required. If vector fields are at staggered locations, the optional argument gridtype must be appropriately set for correct treatment at boundaries. It is safe to apply vector field updates to the appropriate arrays irrespective of the domain topology: if the topology requires no special treatment of vector fields, specifying gridtype will do no harm. mpp_update_domains internally buffers the date being sent and received into single messages for efficiency. A turnable internal buffer area in memory is provided for this purpose by mpp_domains_mod. The size of this buffer area can be set by the user by calling mpp_domains_set_stack_size in *module mpp_domains_mod*.

c. Mpp_redistribute

```
call mpp_redistribute ( domain_in, field_in, domain_out, field_out )
```

DESCRIPTION mpp_redistribute is used to reorganize a distributed array. MPP_TYPE_ can be of type integer, complex, or real; of 4-byte or 8-byte kind; of rank up to 5.

INPUT

field_in	field_in is dimensioned on the data domain of domain_in.

OUTPUT

field_out	field_out on the data domain of domain_out.

d. Mpp_global_field

```
call mpp_global_field ( domain, local, global, flags )
```

DESCRIPTION mpp_global_field is used to get an entire domain-decomposed array on each PE. MPP_TYPE_ can be of type complex, integer, logical or real; of 4-byte or 8-byte kind; of rank up to 5. All PEs in a domain decomposition must call mpp_global_field, and each will have a complete global field at the end. Please note that a global array of rank 3 or higher could occupy a lot of memory.

INPUT

domain	
local	local is dimensioned on either the compute domain or the data domain of domain.
flags	flags can be given the value XONLY or YONLY, to specify a globalization on one axis only.

OUTPUT

global	global is dimensioned on the corresponding global domain.

e. Mpp_global_max

```
mpp_global_max ( domain, field, locus )
```

DESCRIPTION mpp_global_max is used to get the maximum value of a domain-decomposed array on each PE. MPP_TYPE_ can be of type integer or real; of 4-byte or 8-byte kind; of rank up to 5. The dimension of locus must equal the rank of field. All PEs in a domain decomposition must call mpp_global_max, and each will have the result upon exit. The function mpp_global_min, with an identical syntax. is also available.

INPUT

domain	
field	field is dimensioned on either the compute domain or the data domain of domain.

OUTPUT

locus	locus, if present, can be used to retrieve the location of the maximum (as in the MAXLOC intrinsic of f90).

f. Mpp_global_sum

```
call mpp_global_sum ( domain, field, flags )
```

DESCRIPTION mpp_global_sum is used to get the sum of a domain-decomposed array on each PE. MPP_TYPE_ can be of type integer, complex, or real; of 4-byte or 8-byte kind; of rank up to 5.

INPUT

domain	
field	field is dimensioned on either the compute domain or the data domain of domain.
flags	flags, if present, must have the value BITWISE_EXACT_SUM. This produces a sum that is guaranteed to produce the identical result irrespective of how the domain is decomposed. This method does the sum first along the ranks beyond 2, and then calls ` <#mpp_global_field>`__ to produce a global 2D array which is then summed. The default method, which is considerably faster, does a local sum followed by mpp_sum in *module mpp_mod* across the domain decomposition.

NOTE All PEs in a domain decomposition must call mpp_global_sum, and each will have the result upon exit.

g. Operator

DESCRIPTION

The module provides public operators to check for equality/inequality of domaintypes, e.g:

```
type(domain1D) :: a, b
type(domain2D) :: c, d
...
if( a.NE.b )then
```

(continues on next page)

(continued from previous page)

```
    ...
end if
if( c==d )then
    ...
end if
```

Domains are considered equal if and only if the start and end indices of each of their component global, data and compute domains are equal.

h. Mpp_get_compute_domain

```
call mpp_get_compute_domain
```

DESCRIPTION The domain is a derived type with private elements. These routines retrieve the axis specifications associated with the compute domains The 2D version of these is a simple extension of 1D.

i. Mpp_get_compute_domains

```
call mpp_get_compute_domains ( domain, xbegin, xend, xsize, & ybegin, yend, ysize␣
↪)
```

DESCRIPTION Retrieve the entire array of compute domain extents associated with a decomposition.

INPUT `domain`

OUTPUT `xbegin,ybegin, xend,yend, xsize,ysize`

j. Mpp_get_data_domain

```
call mpp_get_data_domain
```

DESCRIPTION The domain is a derived type with private elements. These routines retrieve the axis specifications associated with the data domains. The 2D version of these is a simple extension of 1D.

k. Mpp_get_global_domain

```
call mpp_get_global_domain
```

DESCRIPTION The domain is a derived type with private elements. These routines retrieve the axis specifications associated with the global domains. The 2D version of these is a simple extension of 1D.

l. Mpp_define_layout

```
call mpp_define_layout ( global_indices, ndivs, layout )
```

DESCRIPTION Given a global 2D domain and the number of divisions in the decomposition (`ndivs`: usually the PE count unless some domains are masked) this calls returns a 2D domain layout. By default, `mpp_define_layout` will attempt to divide the 2D index space into domains that maintain the aspect ratio of the global domain. If this cannot be done, the algorithm favours domains that are longer in `x` than `y`, a preference that could improve vector performance.

INPUT

`global_indices`	[integer, dimension(4)]
`ndivs`	[integer]

OUTPUT

`layout`	[integer, dimension(2)]

m. Mpp_get_pelist

DESCRIPTION The 1D version of this call returns an array of the PEs assigned to this 1D domain decomposition. In addition the optional argument `pos` may be used to retrieve the 0-based position of the domain local to the calling PE, i.e `domain%list(pos)%pe` is the local PE, as returned by mpp_pe in *module mpp_mod*. The 2D version of this call is identical to 1D version.

INPUT

`domain`	[type(domain1D)] [type(domain2D)]

OUTPUT

`pelist`	[integer, dimension(:)] [integer, dimension(:)]
`pos`	[integer] [integer]

n. Mpp_get_layout

```
call mpp_get_layout ( domain, layout )
```

DESCRIPTION The 1D version of this call returns the number of divisions that was assigned to this decomposition axis. The 2D version of this call returns an array of dimension 2 holding the results on two axes.

INPUT

`domain`	[type(domain1D)] [type(domain2D)]

OUTPUT

`layout`	[integer] [integer, dimension(2)]

o. Mpp_domains_init

```
call mpp_domains_init (flags)
```

DESCRIPTION Called to initialize the `mpp_domains_mod` package. `flags` can be set to `MPP_VERBOSE` to have `mpp_domains_mod` keep you informed of what it's up to. `MPP_DEBUG` returns even more information for debugging. `mpp_domains_init` will call `mpp_init`, to make sure *module mpp_mod* is initialized. (Repeated calls to `mpp_init` do no harm, so don't worry if you already called it).

INPUT

`flags`	[integer]

p. Mpp_domains_set_stack_size

```
call mpp_domains_set_stack_size (n)
```

DESCRIPTION This sets the size of an array that is used for internal storage by `mpp_domains`. This array is used, for instance, to buffer the data sent and received in halo updates. This call has implied global synchronization. It should be placed somewhere where all PEs can call it.

INPUT

`n`	[integer]

q. Mpp_domains_exit

```
call mpp_domains_exit ()
```

DESCRIPTION Serves no particular purpose, but is provided should you require to re-initialize `mpp_domains_mod`, for some odd reason.

r. Mpp_get_domain_components

```
call mpp_get_domain_components ( domain, x, y )
```

DESCRIPTION It is sometime necessary to have direct recourse to the domain1D types that compose a domain2D object. This call retrieves them.

INPUT

`domain`	[type(domain2D)]

OUTPUT

`x,y`	[type(domain1D)]

6.206.7 Data sets

None.

6.206.8 Error messages

None.

6.206.9 References

None.

6.206.10 Compiler specifics

Any module or program unit using `mpp_domains_mod` must contain the line

```
use mpp_domains_mod
```

`mpp_domains_mod` uses *module mpp_mod*, and therefore is subject to the compiling and linking requirements of *module mpp_mod*.

6.206.11 Precompiler options

`mpp_domains_mod` uses standard f90, and has no special requirements. There are some OS-dependent pre-processor directives that you might need to modify on non-SGI/Cray systems and compilers. The portability, as described in *module mpp_mod* obviously is a constraint, since this module is built on top of it. Contact me, Balaji, SGI/GFDL, with questions.

6.206.12 Loader options

The source consists of the main source file and also requires the following include files: GFDL users can check it out of the main CVS repository as part of the CVS module. The current public tag is . External users can download the latest package . Public access to the GFDL CVS repository will soon be made available.

6.206.13 Test PROGRAM

None.

6.206.14 Notes

None.

6.207 module mpp_io_mod

6.207.1 Overview

mpp_io_mod, is a set of simple calls for parallel I/O on distributed systems. It is geared toward the writing of data in netCDF format. It requires the modules *module mpp_domains_mod* and *module mpp_mod*, upon which it is built.

In massively parallel environments, an often difficult problem is the reading and writing of data to files on disk. MPI-IO and MPI-2 IO are moving toward providing this capability, but are currently not widely implemented. Further, it is a rather abstruse API. mpp_io_mod is an attempt at a simple API encompassing a certain variety of the I/O tasks that will be required. It does not attempt to be an all-encompassing standard such as MPI, however, it can be implemented in MPI if so desired. It is equally simple to add parallel I/O capability to mpp_io_mod based on vendor-specific APIs while providing a layer of insulation for user codes.

The mpp_io_mod parallel I/O API built on top of the *module mpp_domains_mod* and *module mpp_mod* API for domain decomposition and message passing. Features of mpp_io_mod include:

1. Simple, minimal API, with free access to underlying API for more complicated stuff.
2. Self-describing files: comprehensive header information (metadata) in the file itself.
3. Strong focus on performance of parallel write: the climate models for which it is designed typically read a minimal amount of data (typically at the beginning of the run); but on the other hand, tend to write copious amounts of data during the run. An interface for reading is also supplied, but its performance has not yet been optimized.
4. Integrated netCDF capability: netCDF is a data format widely used in the climate/weather modeling community. netCDF is considered the principal medium of data storage for mpp_io_mod. But I provide a raw unformatted fortran I/O capability in case netCDF is not an option, either due to unavailability, inappropriateness, or poor performance.
5. May require off-line post-processing: a tool for this purpose, mppnccombine, is available. GFDL users may use ~hnv/pub/mppnccombine. Outside users may obtain the source here. It can be compiled on any C compiler and linked with the netCDF library. The program is free and is covered by the GPL license.

The internal representation of the data being written out is assumed be the default real type, which can be 4 or 8-byte. Time data is always written as 8-bytes to avoid overflow on climatic time scales in units of seconds.

The I/O activity critical to performance in the models for which mpp_io_mod is designed is typically the writing of large datasets on a model grid volume produced at intervals during a run. Consider a 3D grid volume, where model arrays are stored as (i,j,k). The domain decomposition is typically along i or j: thus to store data to disk as a global volume, the distributed chunks of data have to be seen as non-contiguous. If we attempt to have all PEs write this data into a single file, performance can be seriously compromised because of the data reordering that will be required. Possible options are to have one PE acquire all the data and write it out, or to have all the PEs write independent files, which are recombined offline. These three modes of operation are described in the `mpp_io_mod` terminology in terms of two parameters, *threading* and *fileset*, as follows:

Single-threaded I/O: a single PE acquires all the data and writes it out.
Multi-threaded, single-fileset I/O: many PEs write to a single file.
Multi-threaded, multi-fileset I/O: many PEs write to independent files. This is also called *distributed I/O*.

The middle option is the most difficult to achieve performance. The choice of one of these modes is made when a file is opened for I/O, in ` <#mpp_open>`__.

A requirement of the design of `mpp_io_mod` is that the file must be entirely self-describing: comprehensive header information describing its contents is present in the header of every file. The header information follows the model of netCDF. Variables in the file are divided into *axes* and *fields*. An axis describes a co-ordinate variable, e.g `x,y,z,t`. A field consists of data in the space described by the axes. An axis is described in `mpp_io_mod` using the defined type `axistype`:

```
type, public :: axistype
  sequence
  character(len=128) :: name
  character(len=128) :: units
  character(len=256) :: longname
  character(len=8) :: cartesian
  integer :: len
  integer :: sense           !+/-1, depth or height?
  type(domain1D), pointer :: domain
  real, dimension(:), pointer :: data
  integer :: id, did
  integer :: type  ! external NetCDF type format for axis data
  integer :: natt
  type(atttype), pointer :: Att(:) ! axis attributes
end type axistype
```

A field is described using the type `fieldtype`:

```
type, public :: fieldtype
  sequence
  character(len=128) :: name
  character(len=128) :: units
  character(len=256) :: longname
  real :: min, max, missing, fill, scale, add
  integer :: pack
  type(axistype), dimension(:), pointer :: axes
  integer, dimension(:), pointer :: size
  integer :: time_axis_index
  integer :: id
  integer :: type ! external NetCDF format for field data
  integer :: natt, ndim
  type(atttype), pointer :: Att(:) ! field metadata
end type fieldtype
```

An attribute (global, field or axis) is described using the `atttype`:

```
type, public :: atttype
  sequence
  integer :: type, len
  character(len=128) :: name
  character(len=256)  :: catt
  real(FLOAT_KIND), pointer :: fatt(:)
end type atttype
```

This default set of field attributes corresponds closely to various conventions established for netCDF files. The `pack` attribute of a field defines whether or not a field is to be packed on output. Allowed values of `pack` are 1,2,4 and 8. The value of `pack` is the number of variables written into 8 bytes. In typical use, we write 4-byte reals to netCDF output; thus the default value of `pack` is 2. For `pack` = 4 or 8, packing uses a simple-minded linear scaling scheme using the `scale` and `add` attributes. There is thus likely to be a significant loss of dynamic range with packing. When a field is declared to be packed, the `missing` and `fill` attributes, if supplied, are packed also.

Please note that the pack values are the same even if the default real is 4 bytes, i.e `PACK=1` still follows the definition above and writes out 8 bytes.

A set of *attributes* for each variable is also available. The variable definitions and attribute information is written/read by calling ` <#mpp_write_meta>`__ or ` <#mpp_read_meta>`__. A typical calling sequence for writing data might be:

```
...
  type(domain2D), dimension(:), allocatable, target :: domain
  type(fieldtype) :: field
  type(axistype) :: x, y, z, t
...
  call mpp_define_domains( (/1,nx,1,ny/), domain )
  allocate( a(domain(pe)%x%data%start_index:domain(pe)%x%data%end_index, &
               domain(pe)%y%data%start_index:domain(pe)%y%data%end_index,nz) )
...
  call mpp_write_meta( unit, x, 'X', 'km', 'X distance', &
      domain=domain(pe)%x, data=(/(float(i),i=1,nx)/) )
  call mpp_write_meta( unit, y, 'Y', 'km', 'Y distance', &
      domain=domain(pe)%y, data=(/(float(i),i=1,ny)/) )
  call mpp_write_meta( unit, z, 'Z', 'km', 'Z distance', &
      data=(/(float(i),i=1,nz)/) )
  call mpp_write_meta( unit, t, 'Time', 'second', 'Time' )

  call mpp_write_meta( unit, field, (/x,y,z,t/), 'a', '(m/s)', AAA', &
      missing=-1e36 )
...
  call mpp_write( unit, x )
  call mpp_write( unit, y )
  call mpp_write( unit, z )
...
```

In this example, `x` and `y` have been declared as distributed axes, since a domain decomposition has been associated. `z` and `t` are undistributed axes. `t` is known to be a *record* axis (netCDF terminology) since we do not allocate the `data` element of the `axistype`. *Only one record axis may be associated with a file.* The call to ` <#mpp_write_meta>`__ initializes the axes, and associates a unique variable ID with each axis. The call to `mpp_write_meta` with argument `field` declared `field` to be a 4D variable that is a function of `(x,y,z,t)`, and a unique variable ID is associated with it. A 3D field will be written at each call to `mpp_write(field)`.

The data to any variable, including axes, is written by `mpp_write`.

Any additional attributes of variables can be added through subsequent `mpp_write_meta` calls, using the variable ID as a handle. *Global* attributes, associated with the dataset as a whole, can also be written thus. See the ` <#mpp_write_meta>`__ call syntax below for further details.

You cannot interleave calls to `mpp_write` and `mpp_write_meta`: the first call to `mpp_write` implies that metadata specification is complete.

A typical calling sequence for reading data might be:

```
...
  integer :: unit, natt, nvar, ntime
  type(domain2D), dimension(:), allocatable, target :: domain
  type(fieldtype), allocatable, dimension(:) :: fields
  type(atttype), allocatable, dimension(:) :: global_atts
  real, allocatable, dimension(:) :: times
...
  call mpp_define_domains( (/1,nx,1,ny/), domain )

  call mpp_read_meta(unit)
  call mpp_get_info(unit,natt,nvar,ntime)
  allocate(global_atts(natt))
  call mpp_get_atts(unit,global_atts)
  allocate(fields(nvar))
  call mpp_get_vars(unit, fields)
  allocate(times(ntime))
  call mpp_get_times(unit, times)

  allocate( a(domain(pe)%x%data%start_index:domain(pe)%x%data%end_index, &
              domain(pe)%y%data%start_index:domain(pe)%y%data%end_index,nz) )
...
  do i=1, nvar
    if (fields(i)%name == 'a')  call mpp_read(unit,fields(i),domain(pe), a,
                                          tindex)
  enddo
...
```

In this example, the data are distributed as in the previous example. The call to ` <#mpp_read_meta>`__ initializes all of the metadata associated with the file, including global attributes, variable attributes and non-record dimension data. The call to `mpp_get_info` returns the number of global attributes (`natt`), variables (`nvar`) and time levels (`ntime`) associated with the file identified by a unique ID (`unit`). `mpp_get_atts` returns all global attributes for the file in the derived type `atttype(natt)`. `mpp_get_vars` returns variable types (`fieldtype(nvar)`). Since the record dimension data are not allocated for calls to ` <#mpp_write>`__, a separate call to `mpp_get_times` is required to access record dimension data. Subsequent calls to `mpp_read` return the field data arrays corresponding to the fieldtype. The `domain` type is an optional argument. If `domain` is omitted, the incoming field array should be dimensioned for the global domain, otherwise, the field data is assigned to the computational domain of a local array.

Multi-fileset reads are not supported with `mpp_read`.

6.207.2 Other modules used

```
        mpp_mod
mpp_domains_mod
```

6.207.3 Public interface

```
use mpp_io_mod [, only:  mpp_write_meta,
                         mpp_write,
                         mpp_read,
                         mpp_get_atts,
                         mpp_io_init,
                         mpp_io_exit,
                         mpp_open,
                         mpp_close,
                         mpp_read_meta,
                         mpp_get_info,
                         mpp_get_times,
                         mpp_flush,
                         mpp_get_ncid ]
```

mpp_write_meta: Write metadata.

mpp_write: Write to an open file.

mpp_read: Read from an open file.

mpp_get_atts: Get file global metdata.

mpp_io_init: Initialize `mpp_io_mod`.

mpp_io_exit: Exit `mpp_io_mod`.

mpp_open: Open a file for parallel I/O.

mpp_close: Close an open file.

mpp_read_meta: Read metadata.

mpp_get_info: Get some general information about a file.

mpp_get_times: Get file time data.

mpp_flush: Flush I/O buffers to disk.

mpp_get_ncid: Get netCDF ID of an open file.

6.207.4 Public data

None.

6.207.5 Public routines

a. **Mpp_write_meta**

```
call mpp_write_meta ( unit, axis, name, units, longname, cartesian, sense, domain,
↪ data )
```

```
call mpp_write_meta ( unit, field, axes, name, units, longname, min, max, missing,
↪ fill, scale, add, pack )
```

```
call mpp_write_meta ( unit, id, name, rval=rval, pack=pack )
```

```
call mpp_write_meta ( unit, id, name, ival=ival )
```

```
call mpp_write_meta ( unit, id, name, cval=cval )
```

```
call mpp_write_meta ( unit, name, rval=rval, pack=pack )
```

```
call mpp_write_meta ( unit, name, ival=ival )
```

```
call mpp_write_meta ( unit, name, cval=cval )
```

DESCRIPTION This routine is used to write the metadata describing the contents of a file being written. Each file can contain any number of fields, which are functions of 0-3 space axes and 0-1 time axes. (Only one time axis can be defined per file). The basic metadata defined above for `axistype` and `fieldtype` are written in the first two forms of the call shown below. These calls will associate a unique variable ID with each variable (axis or field). These can be used to attach any other real, integer or character attribute to a variable. The last form is used to define a *global* real, integer or character attribute that applies to the dataset as a whole.

INPUT `unit`, `name`, `units`, `longname`, `cartesian`, `sense`, `domain`, `data`, `min`, `max`, `missing`, `fill`, `scale`, `add`, `pack`, `id`, `cval`, `ival`, `rval`

OUTPUT `axis`, `field`

NOTE The first form defines a time or space axis. Metadata corresponding to the type above are written to the file on <unit>. A unique ID for subsequen references to this axis is returned in axis%id. If the <domain> element is present, this is recognized as a distributed data axis and domain decomposition information is also written if required (the domain decomposition info is required for multi-fileset multi-threaded I/O). If the <data> element is allocated, it is considered to be a space axis, otherwise it is a time axis with an unlimited dimension. Only one time axis is allowed per file. The second form defines a field. Metadata corresponding to the type above are written to the file on <unit>. A unique ID for subsequen references to this field is returned in field%id. At least one axis must be associated, 0D variables are not considered. mpp_write_meta must previously have been called on all axes associated with this field. The third form (3 - 5) defines metadata associated with a previously defined axis or field, identified to mpp_write_meta by its unique ID <id>. The attribute is named <name> and can take on a real, integer or character value. <rval> and <ival> can be scalar or 1D arrays. This need not be called for attributes already contained in the type. The last form (6 - 8) defines global metadata associated with the file as a whole. The attribute is named <name> and can take on a real, integer or character value. <rval> and <ival> can be scalar or 1D arrays. Note that `mpp_write_meta` is expecting axis data on the *global* domain even if it is a domain-decomposed axis. You cannot interleave calls to `mpp_write` and `mpp_write_meta`: the first call to `mpp_write` implies that metadata specification is complete.

b. Mpp_write

```
mpp_write ( unit, axis )
```

```
mpp_write ( unit, field, data, tstamp )
```

```
mpp_write ( unit, field, domain, data, tstamp )
```

DESCRIPTION `mpp_write` is used to write data to the file on an I/O unit using the file parameters supplied by ` <#mpp_open>`__. Axis and field definitions must have previously been written to the file using ` <#mpp_write_meta>`__. There are three forms of mpp_write, one to write axis data, one to write distributed field data, and one to write non-distributed field data. *Distributed* data refer to arrays whose two fastest-varying indices are domain-decomposed. Distributed data must be 2D or 3D (in space). Non-distributed data can be 0-3D. The data argument for distributed data is expected by mpp_write to contain data specified on the *data* domain, and will write the data belonging to the *compute* domain, fetching or sending data as required by the parallel I/O mode specified in the mpp_open call. This is consistent with our definition in *module mpp_domains_mod*, where all arrays are expected to be dimensioned on the data domain, and all operations performed on the compute domain. The type of the data argument must be a *default real*, which can be 4 or 8 byte.

INPUT

tstamp	tstamp is an optional argument. It is to be omitted if the field was defined not to be a function of time. Results are unpredictable if the argument is supplied for a time- independent field, or omitted for a time-dependent field. Repeated writes of a time-independent field are also not recommended. One time level of one field is written per call. tstamp must be an 8-byte real, even if the default real type is 4-byte.

NOTE

The type of write performed by mpp_write depends on the file characteristics on the I/O unit specified at the ` <#mpp_open>`__ call. Specifically, the format of the output data (e.g netCDF or IEEE), the threading and fileset flags, etc., can be changed there, and require no changes to the mpp_write calls.

Packing is currently not implemented for non-netCDF files, and the pack attribute is ignored. On netCDF files, NF_DOUBLEs (8-byte IEEE floating point numbers) are written for pack=1 and NF_FLOATs for pack=2. (pack=2 gives the customary and default behaviour). We write NF_SHORTs (2-byte integers) for pack=4, or NF_BYTEs (1-byte integers) for pack=8. Integer scaling is done using the scale and add attributes at pack=4 or 8, satisfying the relation

```
data = packed_data*scale + add
```

NOTE: mpp_write does not check to see if the scaled data in fact fits into the dynamic range implied by the specified packing. It is incumbent on the user to supply correct scaling attributes.

You cannot interleave calls to mpp_write and mpp_write_meta: the first call to mpp_write implies that metadata specification is complete.

c. Mpp_read

```
call mpp_read ( unit, field, data, time_index )
```

```
call mpp_read ( unit, field, domain, data, time_index )
```

DESCRIPTION mpp_read is used to read data to the file on an I/O unit using the file parameters supplied by ` <#mpp_open>`__. There are two forms of mpp_read, one to read distributed field data, and one to read non-distributed field data. *Distributed* data refer to arrays whose two fastest-varying indices are domain-decomposed. Distributed data must be 2D or 3D (in space). Non-distributed data can be 0-3D. The data argument for distributed data is expected by mpp_read to contain data specified on the *data* domain, and will read the data belonging to the *compute* domain, fetching data as required by the parallel I/O mode specified in the mpp_open call. This is consistent with our definition in *module mpp_domains_mod*, where all arrays are expected to be dimensioned on the data domain, and all operations performed on the compute domain.

INPUT

`unit`	[integer]
`field`	[type(fieldtype)]
`domain`	
`time_index`	time_index is an optional argument. It is to be omitted if the field was defined not to be a function of time. Results are unpredictable if the argument is supplied for a time- independent field, or omitted for a time-dependent field.

INPUT/OUTPUT

`data`	[real, dimension(:,:,:)]

NOTE The type of read performed by `mpp_read` depends on the file characteristics on the I/O unit specified at the ` <#mpp_open>`__ call. Specifically, the format of the input data (e.g netCDF or IEEE) and the `threading` flags, etc., can be changed there, and require no changes to the `mpp_read` calls. (`fileset` = MPP_MULTI is not supported by `mpp_read`; IEEE is currently not supported). Packed variables are unpacked using the `scale` and `add` attributes. `mpp_read_meta` must be called prior to calling `mpp_read.`

d. Mpp_get_atts

```
call mpp_get_atts ( unit, global_atts)
```

DESCRIPTION Get file global metdata.

INPUT

`unit`	[integer]
`global_atts`	[atttype, dimension(:)]

e. Mpp_io_init

```
call mpp_io_init ( flags, maxunit )
```

DESCRIPTION Called to initialize the `mpp_io_mod` package. Sets the range of valid fortran units and initializes the `mpp_file` array of `type(filetype)`. `mpp_io_init` will call `mpp_init` and `mpp_domains_init`, to make sure its parent modules have been initialized. (Repeated calls to the `init` routines do no harm, so don't worry if you already called it).

INPUT

`flags`	[integer]
`maxunit`	[integer]

f. Mpp_io_exit

```
call mpp_io_exit ()
```

DESCRIPTION It is recommended, though not at present required, that you call this near the end of a run. This will close all open files that were opened with ` <#mpp_open>`__. Files opened otherwise are not affected.

g. Mpp_open

```
call mpp_open ( unit, file, action, form, access, threading, fileset, iospec,␣
↪nohdrs, recl, pelist )
```

DESCRIPTION Open a file for parallel I/O.

INPUT

`file`	file is the filename: REQUIRED we append .nc to filename if it is a netCDF file we append .<pppp> to filename if fileset is private (pppp is PE number) [character(len=*)]
`action`	action is one of MPP_RDONLY, MPP_APPEND, MPP_WRONLY or MPP_OVERWR. [integer]
`form`	form is one of MPP_ASCII: formatted read/write MPP_NATIVE: unformatted read/write with no conversion MPP_IEEE32: unformatted read/write with conversion to IEEE32 MPP_NETCDF: unformatted read/write with conversion to netCDF [integer]
`access`	access is one of MPP_SEQUENTIAL or MPP_DIRECT (ignored for netCDF). RECL argument is REQUIRED for direct access IO. [integer]
`threading`	threading is one of MPP_SINGLE or MPP_MULTI single-threaded IO in a multi-PE run is done by PE0. [integer]
`fileset`	fileset is one of MPP_MULTI and MPP_SINGLE fileset is only used for multi-threaded I/O if all I/O PEs in <pelist> use a single fileset, they write to the same file if all I/O PEs in <pelist> use a multi fileset, they each write an independent file [integer]
`pelist`	pelist is the list of I/O PEs (currently ALL). [integer]
`recl`	recl is the record length in bytes. [integer]
`iospec`	iospec is a system hint for I/O organization, e.g assign(1) on SGI/Cray systems. [character(len=*)]
`nohdrs`	nohdrs has no effect when action=MPP_RDONLY|MPP_APPEND or when form=MPP_NETCDF [logical]

OUTPUT

`unit`	unit is intent(OUT): always _returned_by_ mpp_open(). [integer]

NOTE

The integer parameters to be passed as flags (`MPP_RDONLY`, etc) are all made available by use association. The `unit` returned by `mpp_open` is guaranteed unique. For non-netCDF I/O it is a valid fortran unit number and fortran I/O can be directly called on the file.

`MPP_WRONLY` will guarantee that existing files named `file` will not be clobbered. `MPP_OVERWR` allows overwriting of files.

Files opened read-only by many processors will give each processor an independent pointer into the file, i.e:

```
   namelist / nml / ...
...
   call mpp_open( unit, 'input.nml', action=MPP_RDONLY )
   read(unit,nml)
```

will result in each PE independently reading the same namelist.

Metadata identifying the file and the version of `mpp_io_mod` are written to a file that is opened `MPP_WRONLY` or `MPP_OVERWR`. If this is a multi-file set, and an additional global attribute `NumFilesInSet` is written to be used by post-processing software.

If `nohdrs=.TRUE.` all calls to write attributes will return successfully *without* performing any writes to the file. The default is `.FALSE.`.

For netCDF files, headers are always written even if `nohdrs=.TRUE.` The string `iospec` is passed to the OS to characterize the I/O to be performed on the file opened on `unit`. This is typically used for I/O optimization. For example, the FFIO layer on SGI/Cray systems can be used for controlling synchronicity of reads and writes, buffering of data between user space and disk for I/O optimization, striping across multiple disk partitions, automatic data conversion and the like (`man intro_ffio`). All these actions are controlled through the `assign` command. For example, to specify asynchronous caching of data going to a file open on `unit`, one would do:

```
call mpp_open( unit, ... iospec='-F cachea' )
```

on an SGI/Cray system, which would pass the supplied `iospec` to the `assign(3F)` system call.

Currently `iospec` performs no action on non-SGI/Cray systems. The interface is still provided, however: users are cordially invited to add the requisite system calls for other systems.

h. Mpp_close

```
call mpp_close ( unit, action )
```

DESCRIPTION Closes the open file on `unit`. Clears the `type(filetype)` object `mpp_file(unit)` making it available for reuse.

INPUT

`unit`	[integer]
`action`	[integer]

i. Mpp_read_meta

```
call mpp_read_meta (unit)
```

DESCRIPTION This routine is used to read the metadata describing the contents of a file. Each file can contain any number of fields, which are functions of 0-3 space axes and 0-1 time axes. (Only one time axis can be defined per file). The basic metadata defined above for `axistype` and `fieldtype` are stored in `mpp_io_mod` and can be accessed outside of `mpp_io_mod` using calls to `mpp_get_info`, `mpp_get_atts`, `mpp_get_vars` and `mpp_get_times`.

INPUT

unit	[integer]

NOTE mpp_read_meta must be called prior to mpp_read.

j. **Mpp_get_info**

```
call mpp_get_info ( unit, ndim, nvar, natt, ntime )
```

DESCRIPTION Get some general information about a file.

INPUT

unit	[integer]

OUTPUT

ndim	[integer]
nvar	[integer]
natt	[integer]
ntime	[integer]

k. **Mpp_get_times**

```
call mpp_get_times ( unit, time_values )
```

DESCRIPTION Get file time data.

INPUT

unit	[integer]

INPUT/OUTPUT

time_values	[real(DOUBLE_KIND), dimension(:)]

l. **Mpp_flush**

```
call mpp_flush (unit)
```

DESCRIPTION Flushes the open file on unit to disk. Any outstanding asynchronous writes will be completed. Any buffer layers between the user and the disk (e.g the FFIO layer on SGI/Cray systems) will be flushed. Calling mpp_flush on a unit opened with the MPP_RDONLY attribute is likely to lead to erroneous behaviour.

INPUT

unit	[integer]

m. **Mpp_get_ncid**

```
mpp_get_ncid (unit)
```

DESCRIPTION This returns the `ncid` associated with the open file on `unit`. It is used in the instance that the user desires to perform netCDF calls upon the file that are not provided by the `mpp_io_mod` API itself.

INPUT

`unit`	[integer]

6.207.6 Data sets

None.

6.207.7 Error messages

None.

6.207.8 References

None.

6.207.9 Compiler specifics

Any module or program unit using `mpp_io_mod` must contain the line

```
use mpp_io_mod
```

If netCDF output is desired, the cpp flag `-Duse_netCDF` must be turned on. The loader step requires an explicit link to the netCDF library (typically something like `-L/usr/local/lib -lnetcdf`, depending on the path to the netCDF library). netCDF release 3 for fortran is required. Please also consider the compiling and linking requirements of linking as described in *module mpp_domains_mod* and *module mpp_mod*, which are `use`d by this module.

6.207.10 Precompiler options

`mpp_io_mod` uses standard f90. On SGI/Cray systems, certain I/O characteristics are specified using `assign(3F)`. On other systems, the user may have to provide similar capability if required.
There are some OS-dependent pre-processor directives that you might need to modify on non-SGI/Cray systems and compilers.

6.207.11 Loader options

The source consists of the main source file and also requires the following include files: (when compiled with) GFDL users can check it out of the main CVS repository as part of the CVS module. The current public tag is . External users can download the latest package . Public access to the GFDL CVS repository will soon be made available.

6.207.12 Test PROGRAM

None.

6.207.13 Notes

None.

6.208 module mpp_mod

6.208.1 Overview

`mpp_mod`, is a set of simple calls to provide a uniform interface to different message-passing libraries. It currently can be implemented either in the SGI/Cray native SHMEM library or in the MPI standard. Other libraries (e.g MPI-2, Co-Array Fortran) can be incorporated as the need arises.

The data transfer between a processor and its own memory is based on `load` and `store` operations upon memory. Shared-memory systems (including distributed shared memory systems) have a single address space and any processor can acquire any data within the memory by `load` and `store`. The situation is different for distributed parallel systems. Specialized MPP systems such as the T3E can simulate shared-memory by direct data acquisition from remote memory. But if the parallel code is distributed across a cluster, or across the Net, messages must be sent and received using the protocols for long-distance communication, such as TCP/IP. This requires a ``handshaking" between nodes of the distributed system. One can think of the two different methods as involving `puts` or `gets` (e.g the SHMEM library), or in the case of negotiated communication (e.g MPI), `sends` and `recvs`.

The difference between SHMEM and MPI is that SHMEM uses one-sided communication, which can have very low-latency high-bandwidth implementations on tightly coupled systems. MPI is a standard developed for distributed computing across loosely-coupled systems, and therefore incurs a software penalty for negotiating the communication. It is however an open industry standard whereas SHMEM is a proprietary interface. Besides, the puts or gets on which it is based cannot currently be implemented in a cluster environment (there are recent announcements from Compaq that occasion hope).

The message-passing requirements of climate and weather codes can be reduced to a fairly simple minimal set, which is easily implemented in any message-passing API. mpp_mod provides this API.

Features of mpp_mod include:

1. Simple, minimal API, with free access to underlying API for more complicated stuff.
2. Design toward typical use in climate/weather CFD codes.
3. Performance to be not significantly lower than any native API.

This module is used to develop higher-level calls for *module mpp_domains_mod* and *module mpp_io_mod*.

Parallel computing is initially daunting, but it soon becomes second nature, much the way many of us can now write vector code without much effort. The key insight required while reading and writing parallel code is in arriving at a mental grasp of several independent parallel execution streams through the same code (the SPMD model). Each variable you examine may have different values for each stream, the processor ID being an obvious example. Subroutines and function calls are particularly subtle, since it is not always obvious from looking at a call what synchronization between execution streams it implies. An example of erroneous code would be a global barrier call (see ` <#mpp_sync>`__ below) placed within a code block that not all PEs will execute, e.g:

```
if( pe.EQ.0 )call mpp_sync()
```

Here only PE 0 reaches the barrier, where it will wait indefinitely. While this is a particularly egregious example to illustrate the coding flaw, more subtle versions of the same are among the most common errors in parallel code.

It is therefore important to be conscious of the context of a subroutine or function call, and the implied synchronization. There are certain calls here (e.g mpp_declare_pelist, mpp_init, mpp_malloc, mpp_set_stack_size) which must be called by all PEs. There are others which must be called by a subset of PEs (here called a pelist) which must be called by all the PEs in the pelist (e.g mpp_max, mpp_sum, mpp_sync). Still others imply no synchronization at all. I will make every effort to highlight the context of each call in the MPP modules, so that the implicit synchronization is spelt out.

For performance it is necessary to keep synchronization as limited as the algorithm being implemented will allow. For instance, a single message between two PEs should only imply synchronization across the PEs in question. A *global* synchronization (or *barrier*) is likely to be slow, and is best avoided. But codes first parallelized on a Cray T3E tend to have many global syncs, as very fast barriers were implemented there in hardware.

Another reason to use pelists is to run a single program in MPMD mode, where different PE subsets work on different portions of the code. A typical example is to assign an ocean model and atmosphere model to different PE subsets, and couple them concurrently instead of running them serially. The MPP module provides the notion of a *current pelist*, which is set when a group of PEs branch off into a subset. Subsequent calls that omit the pelist optional argument (seen below in many of the individual calls) assume that the implied synchronization is across the current pelist. The calls mpp_root_pe and mpp_npes also return the values appropriate to the current pelist. The mpp_set_current_pelist call is provided to set the current pelist.

6.208.2 Other modules used

```
shmem_interface
            mpi
```

6.208.3 Public interface

F90 is a strictly-typed language, and the syntax pass of the compiler requires matching of type, kind and rank (TKR). Most calls listed here use a generic type, shown here as `MPP_TYPE_`. This is resolved in the pre-processor stage to any of a variety of types. In general the MPP operations work on 4-byte and 8-byte variants of `integer, real, complex, logical` variables, of rank 0 to 5, leading to 48 specific module procedures under the same generic interface. Any of the variables below shown as `MPP_TYPE_` is treated in this way.

```
use mpp_mod [, only:  mpp_max,
                      mpp_sum,
                      mpp_transmit,
                      mpp_broadcast,
                      mpp_chksum,
                      mpp_error,
                      mpp_init,
                      stdin,
                      mpp_exit,
                      mpp_pe,
                      mpp_npes,
                      mpp_declare_pelist,
                      mpp_set_current_pelist,
                      mpp_clock_set_grain,
                      mpp_sync,
                      mpp_sync_self,
                      mpp_malloc,
                      mpp_set_stack_size ]
```

mpp_max: Reduction operations.

mpp_sum: Reduction operation.

mpp_transmit: Basic message-passing call.

mpp_broadcast: Parallel broadcasts.

mpp_chksum: Parallel checksums.

mpp_error: Error handler.

mpp_init: Initialize `mpp_mod`.

stdin: Standard fortran unit numbers.

mpp_exit: Exit `mpp_mod`.

mpp_pe: Returns processor ID.

mpp_npes: Returns processor count for current pelist.

mpp_declare_pelist: Declare a pelist.

mpp_set_current_pelist: Set context pelist.

mpp_clock_set_grain: Set the level of granularity of timing measurements.

mpp_sync: Global synchronization.

mpp_sync_self: Local synchronization.

mpp_malloc: Symmetric memory allocation.

mpp_set_stack_size: Allocate module internal workspace.

6.208.4 Public data

None.

6.208.5 Public routines

a. **Mpp_max**

```
call mpp_max ( a, pelist )
```

DESCRIPTION Find the max of scalar a the PEs in pelist result is also automatically broadcast to all PEs

INPUT

a	real or integer, of 4-byte of 8-byte kind.
pelist	If pelist is omitted, the context is assumed to be the current pelist. This call implies synchronization across the PEs in pelist, or the current pelist if pelist is absent.

b. **Mpp_sum**

```
call mpp_sum ( a, length, pelist )
```

DESCRIPTION MPP_TYPE_ corresponds to any 4-byte and 8-byte variant of integer, real, complex variables, of rank 0 or 1. A contiguous block from a multi-dimensional array may be passed by its starting address and its length, as in f77. Library reduction operators are not required or guaranteed to be bit-reproducible. In any case, changing the processor count changes the data layout, and thus very likely the order of operations. For bit-reproducible sums of distributed arrays, consider using the mpp_global_sum routine provided by the *module mpp_domains_mod* module. The bit_reproducible flag provided in earlier versions of this routine has been removed. If pelist is omitted, the context is assumed to be the current pelist. This call implies synchronization across the PEs in pelist, or the current pelist if pelist is absent.

INPUT length pelist

INPUT/OUTPUT a

c. Mpp_transmit

```
call mpp_transmit ( put_data, put_len, put_pe, get_data, get_len, get_pe )
```

DESCRIPTION

MPP_TYPE_ corresponds to any 4-byte and 8-byte variant of integer, real, complex, logical variables, of rank 0 or 1. A contiguous block from a multi-dimensional array may be passed by its starting address and its length, as in f77.

mpp_transmit is currently implemented as asynchronous outward transmission and synchronous inward transmission. This follows the behaviour of shmem_put and shmem_get. In MPI, it is implemented as mpi_isend and mpi_recv. For most applications, transmissions occur in pairs, and are here accomplished in a single call.

The special PE designations NULL_PE, ANY_PE and ALL_PES are provided by use association.

NULL_PE: is used to disable one of the pair of transmissions.

ANY_PE: is used for unspecific remote destination. (Please note that put_pe=ANY_PE has no meaning in the MPI context, though it is available in the SHMEM invocation. If portability is a concern, it is best avoided).

ALL_PES: is used for broadcast operations.

It is recommended that ` <#mpp_broadcast>`__ be used for broadcasts.

The following example illustrates the use of NULL_PE and ALL_PES:

```
real, dimension(n) :: a
if( pe.EQ.0 )then
    do p = 1,npes-1
       call mpp_transmit( a, n, p, a, n, NULL_PE )
    end do
else
    call mpp_transmit( a, n, NULL_PE, a, n, 0 )
end if

call mpp_transmit( a, n, ALL_PES, a, n, 0 )
```

The do loop and the broadcast operation above are equivalent.

Two overloaded calls mpp_send and mpp_recv have also been provided. mpp_send calls mpp_transmit with get_pe=NULL_PE. mpp_recv calls mpp_transmit with put_pe=NULL_PE. Thus the do loop above could be written more succinctly:

```
if( pe.EQ.0 )then
    do p = 1,npes-1
       call mpp_send( a, n, p )
    end do
else
    call mpp_recv( a, n, 0 )
end if
```

d. Mpp_broadcast

```
call mpp_broadcast ( data, length, from_pe, pelist )
```

DESCRIPTION The `mpp_broadcast` call has been added because the original syntax (using `ALL_PES` in `mpp_transmit`) did not support a broadcast across a pelist. `MPP_TYPE_` corresponds to any 4-byte and 8-byte variant of `integer, real, complex, logical` variables, of rank 0 or 1. A contiguous block from a multi-dimensional array may be passed by its starting address and its length, as in `f77`. Global broadcasts through the `ALL_PES` argument to ` <#mpp_transmit>`__ are still provided for backward-compatibility. If `pelist` is omitted, the context is assumed to be the current pelist. `from_pe` must belong to the current pelist. This call implies synchronization across the PEs in `pelist`, or the current pelist if `pelist` is absent.

INPUT `length`, `from_pe`, `pelist`

INPUT/OUTPUT `data(*)`

e. Mpp_chksum

```
mpp_chksum ( var, pelist )
```

DESCRIPTION

`mpp_chksum` is a parallel checksum routine that returns an identical answer for the same array irrespective of how it has been partitioned across processors. `LONG_KIND`is the `KIND` parameter corresponding to long integers (see discussion on OS-dependent preprocessor directives) defined in the header file `os.h`. `MPP_TYPE_` corresponds to any 4-byte and 8-byte variant of `integer, real, complex, logical` variables, of rank 0 to 5.

Integer checksums on FP data use the F90 `TRANSFER()` intrinsic.

The serial checksum module is superseded by this function, and is no longer being actively maintained. This provides identical results on a single-processor job, and to perform serial checksums on a single processor of a parallel job, you only need to use the optional `pelist` argument.

```
use mpp_mod
integer :: pe, chksum
real :: a(:)
pe = mpp_pe()
chksum = mpp_chksum( a, (/pe/) )
```

The additional functionality of `mpp_chksum` over serial checksums is to compute the checksum across the PEs in `pelist`. The answer is guaranteed to be the same for the same distributed array irrespective of how it has been partitioned.

If `pelist` is omitted, the context is assumed to be the current pelist. This call implies synchronization across the PEs in `pelist`, or the current pelist if `pelist` is absent.

INPUT `pelist`, `var`

f. Mpp_error

```
call mpp_error ( errortype, routine, errormsg )
```

DESCRIPTION

It is strongly recommended that all error exits pass through mpp_error to assure the program fails cleanly. An individual PE encountering a STOP statement, for instance, can cause the program to hang. The use of the STOP statement is strongly discouraged.

Calling mpp_error with no arguments produces an immediate error exit, i.e:

```
call mpp_error
call mpp_error(FATAL)
```

are equivalent.

The argument order

```
call mpp_error( routine, errormsg, errortype )
```

is also provided to support legacy code. In this version of the call, none of the arguments may be omitted.

The behaviour of mpp_error for a WARNING can be controlled with an additional call mpp_set_warn_level.

```
call mpp_set_warn_level(ERROR)
```

causes mpp_error to treat WARNING exactly like FATAL.

```
call mpp_set_warn_level(WARNING)
```

resets to the default behaviour described above.

mpp_error also has an internal error state which maintains knowledge of whether a warning has been issued. This can be used at startup in a subroutine that checks if the model has been properly configured. You can generate a series of warnings using mpp_error, and then check at the end if any warnings has been issued using the function mpp_error_state(). If the value of this is WARNING, at least one warning has been issued, and the user can take appropriate action:

```
if( ... )call mpp_error( WARNING, '...' )
if( ... )call mpp_error( WARNING, '...' )
if( ... )call mpp_error( WARNING, '...' )
...
if( mpp_error_state().EQ.WARNING )call mpp_error( FATAL, '...' )
```

INPUT errortype. One of NOTE, WARNING or FATAL (these definitions are acquired by use association). NOTE writes errormsg to STDOUT. WARNING writes errormsg to STDERR. FATAL writes errormsg to STDERR, and induces a clean error exit with a call stack traceback.

g. Mpp_init

```
call mpp_init ( flags )
```

DESCRIPTION Called to initialize the mpp_mod package. It is recommended that this call be the first executed line in your program. It sets the number of PEs assigned to this run (acquired from the command line, or through the environment variable NPES), and associates an ID number to each PE. These can be accessed by calling ` <#mpp_npes>`__ and ` <#mpp_pe>`__.

INPUT flags<flags can be set to MPP_VERBOSE to have mpp_mod keep you informed of what it's up to. [integer]

h. Stdin

```
stdin ()
```

DESCRIPTION This function, as well as stdout(), stderr(), stdlog(), returns the current standard fortran unit numbers for input, output, error messages and log messages. Log messages, by convention, are written to the file logfile.out.

i. Mpp_exit

```
call mpp_exit ()
```

DESCRIPTION Called at the end of the run, or to re-initialize mpp_mod, should you require that for some odd reason. This call implies synchronization across all PEs.

j. Mpp_pe

```
mpp_pe ()
```

DESCRIPTION This returns the unique ID associated with a PE. This number runs between 0 and npes-1, where npes is the total processor count, returned by mpp_npes. For a uniprocessor application this will always return 0.

k. Mpp_npes

```
mpp_npes ()
```

DESCRIPTION This returns the number of PEs in the current pelist. For a uniprocessor application, this will always return 1.

l. Mpp_declare_pelist

```
call mpp_declare_pelist ( pelist,name )
```

DESCRIPTION This call is written specifically to accommodate a MPI restriction that requires a parent communicator to create a child communicator, In other words: a pelist cannot go off and declare a communicator, but every PE in the parent, including those not in pelist(:), must get together for the `MPI_COMM_CREATE` call. The parent is typically `MPI_COMM_WORLD`, though it could also be a subset that includes all PEs in `pelist`. The restriction does not apply to SMA but to have uniform code, you may as well call it. This call implies synchronization across the PEs in the current pelist, of which `pelist` is a subset.

INPUT `pelist` [integer, dimension(:)]

m. Mpp_set_current_pelist

```
call mpp_set_current_pelist ( pelist )
```

DESCRIPTION This call sets the value of the current pelist, which is the context for all subsequent "global" calls where the optional `pelist` argument is omitted. All the PEs that are to be in the current pelist must call it. In MPI, this call may hang unless `pelist` has been previous declared using ` <#mpp_declare_pelist>`__. If the argument `pelist` is absent, the current pelist is set to the "world" pelist, of all PEs in the job.

INPUT `pliest` [integer]

n. Mpp_clock_set_grain

```
call mpp_clock_set_grain ( grain )
```

DESCRIPTION This routine and three other routines, mpp_clock_id, mpp_clock_begin(id), and mpp_clock_end(id) may be used to time parallel code sections, and extract parallel statistics. Clocks are identified by names, which should be unique in the first 32 characters. The `mpp_clock_id` call initializes a clock of a given name and returns an integer `id`. This `id` can be used by subsequent `mpp_clock_begin` and `mpp_clock_end` calls set around a code section to be timed. Example:

```
integer :: id
id = mpp_clock_id( 'Atmosphere' )
call mpp_clock_begin(id)
call atmos_model()
call mpp_clock_end()
```

Two flags may be used to alter the behaviour of `mpp_clock`. If the flag `MPP_CLOCK_SYNC` is turned on by `mpp_clock_id`, the clock calls `mpp_sync` across all the PEs in the current pelist at the top of the timed code section, but allows each PE to complete the code section (and reach `mpp_clock_end`) at different times. This allows us to measure load imbalance for a given code section. Statistics are written to `stdout` by `mpp_exit`.

The flag `MPP_CLOCK_DETAILED` may be turned on by `mpp_clock_id` to get detailed communication profiles. Communication events of the types `SEND, RECV, BROADCAST, REDUCE` and `WAIT` are separately measured for data volume and time. Statistics are written to `stdout` by `mpp_exit`, and individual PE info is also written to the file `mpp_clock.out.####` where `####` is the PE id given by `mpp_pe`.

The flags MPP_CLOCK_SYNC and MPP_CLOCK_DETAILED are integer parameters available by use association, and may be summed to turn them both on.

While the nesting of clocks is allowed, please note that turning on the non-optional flags on inner clocks has certain subtle issues. Turning on MPP_CLOCK_SYNC on an inner clock may distort outer clock measurements of load imbalance. Turning on MPP_CLOCK_DETAILED will stop detailed measurements on its outer clock, since only one detailed clock may be active at one time. Also, detailed clocks only time a certain number of events per clock (currently 40000) to conserve memory. If this array overflows, a warning message is printed, and subsequent events for this clock are not timed.

Timings are done using the f90 standard SYSTEM_CLOCK intrinsic.

The resolution of SYSTEM_CLOCK is often too coarse for use except across large swaths of code. On SGI systems this is transparently overloaded with a higher resolution clock made available in a non-portable fortran interface made available by nsclock.c. This approach will eventually be extended to other platforms.

New behaviour added at the Havana release allows the user to embed profiling calls at varying levels of granularity all over the code, and for any particular run, set a threshold of granularity so that finer-grained clocks become dormant.

The threshold granularity is held in the private module variable clock_grain. This value may be modified by the call mpp_clock_set_grain, and affect clocks initiated by subsequent calls to mpp_clock_id. The value of clock_grain is set to an arbitrarily large number initially.

Clocks initialized by mpp_clock_id can set a new optional argument grain setting their granularity level. Clocks check this level against the current value of clock_grain, and are only triggered if they are *at or below ("coarser than")* the threshold. Finer-grained clocks are dormant for that run.

Note that subsequent changes to clock_grain do not change the status of already initiated clocks, and that if the optional grain argument is absent, the clock is always triggered. This guarantees backward compatibility.

INPUT grain [integer]

o. Mpp_sync

```
call mpp_sync ( pelist )
```

DESCRIPTION Synchronizes PEs at this point in the execution. If pelist is omitted all PEs are synchronized. This can be expensive on many systems, and should be avoided if possible. Under MPI, we do not call MPI_BARRIER, as you might expect. This is because this call can be prohibitively slow on many systems. Instead, we perform the same operation as mpp_sync_self, i.e all participating PEs wait for completion of all their outstanding non-blocking operations. If pelist is omitted, the context is assumed to be the current pelist. This call implies synchronization across the PEs in pelist, or the current pelist if pelist is absent.

INPUT pelist [integer, dimension(:)]

p. Mpp_sync_self

DESCRIPTION mpp_transmit is implemented as asynchronous put/send and synchronous get/recv. mpp_sync_self guarantees that outstanding asynchronous operations from the calling PE are complete. If pelist is supplied, mpp_sync_self checks only for outstanding puts to the PEs in pelist. If pelist is omitted, the context is assumed to be the current pelist. This call implies synchronization across the PEs in pelist, or the current pelist if pelist is absent.

INPUT pelist [integer, dimension(:)]

q. Mpp_malloc

```
call mpp_malloc ( ptr, newlen, len )
```

DESCRIPTION This routine is used on SGI systems when mpp_mod is invoked in the SHMEM library. It ensures that dynamically allocated memory can be used with shmem_get and shmem_put. This is called *symmetric allocation* and is described in the intro_shmem man page. ptr is a *Cray pointer* (see the section on portability). The operation can be expensive (since it requires a global barrier). We therefore attempt to re-use existing allocation whenever possible. Therefore len and ptr must have the SAVE attribute in the calling routine, and retain the information about the last call to mpp_malloc. Additional memory is symmetrically allocated if and only if newlen exceeds len. This is never required on Cray PVP or MPP systems. While the T3E manpages do talk about symmetric allocation, mpp_mod is coded to remove this restriction. It is never required if mpp_mod is invoked in MPI. This call implies synchronization across all PEs.

INPUT ptr, a cray pointer, points to a dummy argument in this routine. newlen, the required allocation length for the pointer ptr [integer]. len, the current allocation (0 if unallocated). [integer].

r. Mpp_set_stack_size

```
call mpp_set_stack_size (n)
```

DESCRIPTION mpp_mod maintains a private internal array called mpp_stack for private workspace. This call sets the length, in words, of this array. The mpp_init call sets this workspace length to a default of 32768, and this call may be used if a longer workspace is needed. This call implies synchronization across all PEs. This workspace is symmetrically allocated, as required for efficient communication on SGI and Cray MPP systems. Since symmetric allocation must be performed by *all* PEs in a job, this call must also be called by all PEs, using the same value of n. Calling mpp_set_stack_size from a subset of PEs, or with unequal argument n, may cause the program to hang. If any MPP call using mpp_stack overflows the declared stack array, the program will abort with a message specifying the stack length that is required. Many users wonder why, if the required stack length can be computed, it cannot also be specified at that point. This cannot be automated because there is no way for the program to know if all PEs are present at that call, and with equal values of n. The program must be rerun by the user with the correct argument to mpp_set_stack_size, called at an appropriate point in the code where all PEs are known to be present.

INPUT

n	[integer]

6.208.6 Data sets

None.

6.208.7 Error messages

None.

6.208.8 References

None.

6.208.9 Compiler specifics

Any module or program unit using `mpp_mod` must contain the line

```
use mpp_mod
```

The source file for `mpp_mod` is ` <ftp://ftp.gfdl.gov/pub/vb/mpp/mpp.F90>`__. Activate the preprocessor flag `-Duse_libSMA` to invoke the SHMEM library, or `-Duse_libMPI` to invoke the MPI library. Global translation of preprocessor macros is required. This required the activation of the `-F` flag on Cray systems and the `-ftpp -macro_expand` flags on SGI systems. On non-SGI/Cray systems, please consult the f90 manpage for the equivalent flag. On Cray PVP systems, *all* routines in a message-passing program must be compiled with `-a taskcommon`. On SGI systems, it is required to use 4-byte integers and 8-byte reals, and the 64-bit ABI (`-i4 -r8 -64 -mips4`). It is also required on SGI systems to link the following libraries explicitly: one of `-lmpi` and `-lsma`, depending on whether you wish to use the SHMEM or MPI implementations; and `-lexc`). On Cray systems, all the required flags are default. On SGI, use MIPSPro f90 7.3.1.2 or higher. On Cray, use cf90 3.0.0.0 or higher. On either, use the message-passing toolkit MPT 1.2 or higher. The declaration `MPI_INTEGER8` for 8-byte integers was provided by `mpp_mod` because it was absent in early releases of the Message Passing Toolkit. It has since been included there, and the declaration in `mpp_mod` commented out. This declaration may need to be reinstated if you get a compiler error from this (i.e you are using a superseded version of the MPT). By turning on the cpp flag `-Dtest_mpp` and compiling `mpp_mod` by itself, you may create a test program to exercise certain aspects of `mpp_mod`, e.g

```
f90 -F -Duse_libSMA -Dtest_mpp mpp.F90
mpprun -n4 a.out
```

runs a 4-PE test on a t3e.

6.208.10 Precompiler options

While the SHMEM library is currently available only on SGI/Cray systems, mpp_mod can be used on any other system with a standard-compliant f90 compiler and MPI library. SHMEM is now becoming available on other systems as well. There are some OS-dependent pre-processor directives that you might need to modify on non-SGI/Cray systems and compilers. On SGI systems, the f90 standard SYSTEM_CLOCK intrinsic is overloaded with a non-portable fortran interface to a higher-precision clock. This is distributed with the MPP package as nsclock.c. This approach will eventually be extended to other platforms, since the resolution of the default clock is often too coarse for our needs.

6.208.11 Test PROGRAM

None.

6.208.12 Notes

None.

6.209 module fft_mod

6.209.1 Overview

Performs simultaneous fast Fourier transforms (FFTs) between real grid space and complex Fourier space.

This routine computes multiple 1-dimensional FFTs and inverse FFTs. There are 2d and 3d versions between type real grid point space and type complex Fourier space. There are single (32-bit) and full (64-bit) versions. On Cray and SGI systems, vendor-specific scientific library routines are used, otherwise a user may choose a NAG library version or stand-alone version using Temperton's FFT.

6.209.2 Other modules used

```
platform_mod
     fms_mod
   fft99_mod
```

6.209.3 Public interface

```
use fft_mod [, only:  fft_grid_to_fourier,
                      fft_fourier_to_grid,
                      fft_init,
                      fft_end ]
```

fft_grid_to_fourier: Given multiple sequences of real data values, this routine computes the complex Fourier transform for all sequences.

fft_fourier_to_grid: Given multiple sequences of Fourier space transforms, this routine computes the inverse transform and returns the real data values for all sequences.

fft_init: This routine must be called to initialize the size of a single transform and setup trigonometric constants.

fft_end: This routine is called to unset the transform size and deallocate memory.

6.209.4 Public data

None.

6.209.5 Public routines

a. **Fft_grid_to_fourier**

```
fourier = fft_grid_to_fourier ( grid )
```

DESCRIPTION Given multiple sequences of real data values, this routine computes the complex Fourier transform for all sequences.

INPUT

grid	Multiple sequence of real data values. The first dimension must be n+1 (where n is the size of a single sequence). [real(R4_KIND), dimension(:,:)] [real(R8_KIND), dimension(:,:)] [real(R4_KIND), dimension(:,:,:)] [real(R8_KIND), dimension(:,:,:)]

OUTPUT

fourier	Multiple sequences of transformed data in complex Fourier space. The first dimension must equal n/2+1 (where n is the size of a single sequence). The remaining dimensions must be the same size as the input argument “grid”. [complex(R4_KIND), dimension(lenc,size(grid,2))] [complex(R8_KIND), dimension(lenc,size(grid,2))] [complex(R4_KIND), dimension(lenc,size(grid,2),size(grid,3))] [complex(R8_KIND), dimension(lenc,size(grid,2),size(grid,3))]

NOTE The complex Fourier components are passed in the following format.

```
fourier (1)      = cmplx ( a(0), b(0) )
fourier (2)      = cmplx ( a(1), b(1) )
    :                :
    :                :
fourier (n/2+1) = cmplx ( a(n/2), b(n/2) )
```

where n = length of each real transform

b. Fft_fourier_to_grid

```
grid = fft_fourier_to_grid ( fourier )
```

DESCRIPTION Given multiple sequences of Fourier space transforms, this routine computes the inverse transform and returns the real data values for all sequences.

INPUT

`fourier`	Multiple sequence complex Fourier space transforms. The first dimension must equal n/2+1 (where n is the size of a single real data sequence). [real(R4_KIND), dimension(:,:)] [real(R8_KIND), dimension(:,:)] [real(R4_KIND), dimension(:,:,:)] [real(R8_KIND), dimension(:,:,:)]

OUTPUT

`grid`	Multiple sequence of real data values. The first dimension must be n+1 (where n is the size of a single sequence). The remaining dimensions must be the same size as the input argument “fourier”. [complex(R4_KIND), dimension(leng1,size(fourier,2))] [complex(R8_KIND), dimension(leng1,size(fourier,2))] [complex(R4_KIND), dimension(leng1,size(fourier,2),size(fourier,3))] [complex(R8_KIND), dimension(leng1,size(fourier,2),size(fourier,3))]

c. Fft_init

```
call fft_init ( n )
```

DESCRIPTION This routine must be called once to initialize the size of a single transform. To change the size of the transform the routine fft_exit must be called before re-initialing with fft_init.

INPUT

`n`	The number of real values in a single sequence of data. The resulting transformed data will have n/2+1 pairs of complex values. [integer]

d. **Fft_end**

```
call fft_end
```

DESCRIPTION This routine is called to unset the transform size and deallocate memory. It can not be called unless fft_init has already been called. There are no arguments.

6.209.6 Data sets

None.

6.209.7 Error messages

Error in fft_grid_to_fourier fft_init must be called The initialization routine fft_init must be called before routines fft_grid_to_fourier.

Error in fft_grid_to_fourier size of first dimension of input data is wrong The real grid point field must have a first dimension equal to n+1 (where n is the size of each real transform). This message occurs when using the SGI/Cray fft.

Error in fft_grid_to_fourier length of input data too small The real grid point field must have a first dimension equal to n (where n is the size of each real transform). This message occurs when using the NAG or Temperton fft.

Error in fft_grid_to_fourier float kind not supported for nag fft 32-bit real data is not supported when using the NAG fft. You may try modifying this part of the code by uncommenting the calls to the NAG library or less consider using the Temperton fft.

Error in fft_fourier_to_grid fft_init must be called The initialization routine fft_init must be called before routines fft_fourier_to_grid.

Error in fft_fourier_to_grid size of first dimension of input data is wrong The complex Fourier field must have a first dimension equal to n/2+1 (where n is the size of each real transform). This message occurs when using the SGI/Cray fft.

Error in fft_fourier_to_grid length of input data too small The complex Fourier field must have a first dimension greater than or equal to n/2+1 (where n is the size of each real transform). This message occurs when using the NAG or Temperton fft.

Error in fft_fourier_to_grid float kind not supported for nag fft float kind not supported for nag fft 32-bit real data is not supported when using the NAG fft. You may try modifying this part of the code by uncommenting the calls to the NAG library or less consider using the Temperton fft.

FATAL in fft_init attempted to reinitialize fft You must call fft_exit before calling fft_init for a second time.

Error in fft_end attempt to un-initialize fft that has not been initialized You can not call fft_end unless fft_init has been called.

6.209.8 References

1. For the SGI/Cray version refer to the manual pages for DZFFTM, ZDFFTM, SCFFTM, and CSFFTM.
2. For the NAG version refer to the NAG documentation for routines C06FPF, C06FQF, and C06GQF.

6.209.9 Compiler specifics

None.

6.209.10 Precompiler options

-D NAGFFT -D NAGFFT On non-Cray/SGI machines, set to use the NAG library FFT routines. Otherwise the Temperton FFT is used by default.

-D test_fft Provides source code for a simple test program. The program generates several sequences of real data. This data is transformed to Fourier space and back to real data, then compared to the original real data.

6.209.11 Loader options

On SGI machines the scientific library needs to be loaded by linking with:

```
-lscs
```

If using the NAG library, the following loader options (or something similar) may be necessary:

```
-L/usr/local/lib -lnag
```

6.209.12 Test PROGRAM

None.

6.209.13 Notes

The routines are overloaded for 2d and 3d versions. The 2d versions copy data into 3d arrays then calls the 3d interface. On SGI/Cray machines: There are single (32-bit) and full (64-bit) versions. For Cray machines the single precision version does not apply. On non-SGI/CRAY machines: The NAG library option uses the "full" precision NAG routines (C06FPF,C06FQF,C06GQF). Users may have to specify a 64-bit real compiler option (e.g., -r8). The stand-alone Temperton FFT option works for the real precision specified at compile time. If you compiled with single (32-bit) real precision then FFT's cannot be computed at full (64-bit) precision.

6.210 Module sat_vapor_pres_mod

6.210.1 Overview

Routines for determining the saturation vapor pressure (`ES`) and the derivative of `ES` with respect to temperature.

This module contains routines for determining the saturation vapor pressure (`ES`) from lookup tables constructed using equations given in the Smithsonian tables. The `ES` lookup tables are valid between -160C and +100C (approx 113K to 373K). The values of `ES` are computed over ice from -160C to -20C, over water from 0C to 100C, and a blended value (over water and ice) from -20C to 0C. This version was written for non-vector machines. See the notes section for details on vectorization.

6.210.2 Other modules used

```
constants_mod
      fms_mod
```

6.210.3 Public interface

Description summarizing public interface.

lookup_es: For the given temperatures, returns the saturation vapor pressures.

lookup_des: For the given temperatures, returns the derivative of saturation vapor pressure with respect to temperature.

compute_es: For the given temperatures, computes the saturation vapor pressures.

sat_vapor_pres_init: Initializes the lookup tables for saturation vapor pressure.

6.210.4 Public data

None.

6.210.5 Public routines

a. **Lookup_es**

```
call lookup_es ( temp, esat )
```

DESCRIPTION For the given temperatures these routines return the saturation vapor pressure (esat). The return values are derived from lookup tables (see notes below).

INPUT

`temp`	Temperature in degrees Kelvin. [real, dimension(scalar)] [real, dimension(:)] [real, dimension(:,:)] [real, dimension(:,:,:)]

OUTPUT

`esat`	Saturation vapor pressure in pascals. May be a scalar, 1d, 2d, or 3d array. Must have the same order and size as temp. [real, dimension(scalar)] [real, dimension(:)] [real, dimension(:,:)] [real, dimension(:,:,:)]

b. **Lookup_des**

```
call lookup_des ( temp, desat )
```

DESCRIPTION For the given temperatures these routines return the derivative of esat w.r.t. temperature (desat). The return values are derived from lookup tables (see notes below).

INPUT

`temp`	Temperature in degrees Kelvin. [real, dimension(scalar)] [real, dimension(:)] [real, dimension(:,:)] [real, dimension(:,:,:)]

OUTPUT

`desat`	Derivative of saturation vapor pressure w.r.t. temperature in pascals/degree. May be a scalar, 1d, 2d, or 3d array. Must have the same order and size as temp. [real, dimension(scalar)] [real, dimension(:)] [real, dimension(:,:)] [real, dimension(:,:,:)]

c. Compute_es

```
es = compute_es ( temp )
```

DESCRIPTION Computes saturation vapor pressure for the given temperature using the equations given in the Smithsonian Meteorological Tables. Between -20C and 0C a blended value over ice and water is returned.

INPUT

`temp`	Temperature in degrees Kelvin. [real, dimension(:)] [real, dimension(scalar)] [real, dimension(:,:)] [real, dimension(:,:,:)]

OUTPUT

`es`	Saturation vapor pressure in pascals. May be a scalar, 1d, 2d, or 3d array. Must have the same order and size as temp. [real, dimension(:)] [real, dimension(scalar)] [real, dimension(:,:)] [real, dimension(:,:,:)]

d. Sat_vapor_pres_init

```
call sat_vapor_pres_init
```

DESCRIPTION Initializes the lookup tables for saturation vapor pressure. This routine will be called automatically the first time **lookup_es** or **lookup_des** is called, the user does not need to call this routine. There are no arguments.

6.210.6 Data sets

None.

6.210.7 Error messages

FATAL in lookup_es table overflow, nbad=## Temperature(s) provided to the saturation vapor pressure lookup are outside the valid range of the lookup table (-160 to 100 deg C). This may be due to a numerical instability in the model. Information should have been printed to standard output to help determine where the instability may have occurred. If the lookup table needs a larger temperature range, then parameters in the module header must be modified.

6.210.8 References

1. Smithsonian Meteorological Tables Page 350.

6.210.9 Compiler specifics

None.

6.210.10 Precompiler options

None.

6.210.11 Loader options

None.

6.210.12 Test PROGRAM

test_sat_vapor_pres

```
use sat_vapor_pres_mod
implicit none

integer, parameter :: ipts=500, jpts=100, kpts=50, nloop=1
real, dimension(ipts,jpts,kpts) :: t,es,esn,des,desn
integer :: n

 generate temperatures between 120K and 340K
  call random_number (t)
  t = 130. + t * 200.

 initialize the tables (optional)
  call sat_vapor_pres_init

 compute actual es and "almost" actual des
   es = compute_es  (t)
  des = compute_des (t)

do n = 1, nloop
 es and des
  call lookup_es  (t, esn)
  call lookup_des (t,desn)
enddo

 terminate, print deviation from actual
  print *, 'size=',ipts,jpts,kpts,nloop
  print *, 'err es  = ', sum((esn-es)**2)
  print *, 'err des = ', sum((desn-des)**2)

contains
```

(continues on next page)

(continued from previous page)

```
----------------------------------
 routine to estimate derivative

 function compute_des (tem) result (des)
 real, intent(in) :: tem(:,:,:)
 real, dimension(size(tem,1),size(tem,2),size(tem,3)) :: des,esp,esm
 real, parameter :: tdel = .01
    esp = compute_es (tem+tdel)
    esm = compute_es (tem-tdel)
    des = (esp-esm)/(2*tdel)
 end function compute_des
----------------------------------

end program test_sat_vapor_pres
```

6.210.13 Notes

1. **Vectorization** To create a vector version the lookup routines need to be modified. The local variables: tmp, del, ind, should be changed to arrays with the same size and order as input array temp. 2. **Construction of the ``ES`` tables** The tables are constructed using the saturation vapor pressure (`ES`) equations in the Smithsonian tables. The tables are valid between -160C to +100C with increments at 1/10 degree. Between -160C and -20C values of `ES` over ice are used, between 0C and 100C values of `ES` over water are used, between -20C and 0C blended values of `ES` (over water and over ice) are used. There are three tables constructed: `ES`, first derivative (`ES'`), and second derivative (`ES`"). The ES table is constructed directly from the equations in the Smithsonian tables. The `ES`' table is constructed by bracketing temperature values at +/- 0.01 degrees. The `ES`" table is estimated by using centered differencing of the `ES`' table. 3. **Determination of ``es`` and ``es'`` from lookup tables** Values of the saturation vapor pressure (`es`) and the derivative (`es'`) are determined at temperature (T) from the lookup tables (`ES`, `ES'`, `ES''`) using the following formula.

```
es (T) = ES(t) + ES'(t) * dt + 0.5 * ES''(t) * dt**2
es'(T) = ES'(t) + ES''(t) * dt

where     t = lookup table temperature closest to T
         dt = T - t
```

4. Internal (private) parameters These parameters can be modified to increase/decrease the size/range of the lookup tables.

```
tcmin   The minimum temperature (in deg C) in the lookup tables.
              [integer, default: tcmin = -160]

    tcmax   The maximum temperature (in deg C) in the lookup tables.
              [integer, default: tcmin = +100]
```

6.211 module topography_mod

6.211.1 Overview

Routines for creating land surface topography fields and land-water masks for latitude-longitude grids.

This module generates realistic mountains and land-water masks on a specified latitude-longitude grid by interpolating from the 1/6 degree Navy mean topography and percent water data sets. The fields that can be generated are mean and standard deviation of topography within the specified grid boxes; and land-ocean (or water) mask and land-ocean (or water) fractional area. The interpolation scheme conserves the area-weighted average of the input data by using module horiz_interp. The interfaces get_gaussian_topog and gaussian_topog_init are documented in *module gaussian_topog_mod*.

6.211.2 Other modules used

```
gaussian_topog_mod
  horiz_interp_mod
           fms_mod
```

6.211.3 Public interface

```
use topography_mod [, only:  get_topog_mean,
                             get_topog_stdev,
                             get_ocean_frac,
                             get_ocean_mask,
                             get_water_frac,
                             get_water_mask ]
```

get_topog_mean: Returns a “realistic” mean surface height field.

get_topog_stdev: Returns a standard deviation of higher resolution topography with the given model grid boxes.

get_ocean_frac: Returns fractional area covered by ocean in a grid box.

get_ocean_mask: Returns a land-ocean mask in a grid box.

get_water_frac: Returns fractional area covered by water.

get_water_mask: Returns a land-water mask in a grid box.

6.211.4 Public data

None.

6.211.5 Public routines

a. **Get_topog_mean**

```
flag = <B> get_topog_mean </B> ( blon, blat, zmean )
```

DESCRIPTION Returns realistic mountains on a latitude-longtude grid. The returned field is the mean topography for the given grid boxes. Computed using a conserving area-weighted interpolation. The current input data set is the 1/6 degree Navy mean topography.

INPUT

`blon`	The longitude (in radians) at grid box boundaries. [real, dimension(:)]
`blat`	The latitude (in radians) at grid box boundaries. [real, dimension(:)]

OUTPUT

`zmean`	The mean surface height (meters). The size of this field must be size(blon)-1 by size(blat)-1. [real, dimension(:,:)]
`get_topog_mean`	A logical value of TRUE is returned if the surface height field was successfully created. A value of FALSE may be returned if the input topography data set was not readable. [logical]

b. **Get_topog_stdev**

```
flag = <B> get_topog_stdev </B> ( blon, blat, stdev )
```

DESCRIPTION Returns the standard deviation of the “finer” input topography data set, currently the Navy 1/6 degree mean topography data, within the boundaries of the given input grid.

INPUT

`blon`	The longitude (in radians) at grid box boundaries. [real, dimension(:)]
`blat`	The latitude (in radians) at grid box boundaries. [real, dimension(:)]

OUTPUT

`stdev`	The standard deviation of surface height (in meters) within given input model grid boxes. The size of this field must be size(blon)-1 by size(blat)-1. [real, dimension(:,:)]
`get_topog_stdev`	A logical value of TRUE is returned if the output field was successfully created. A value of FALSE may be returned if the input topography data set was not readable. [logical]

c. Get_ocean_frac

```
flag = <B> get_ocean_frac </B> ( blon, blat, ocean_frac )
```

DESCRIPTION Returns fractional area covered by ocean in the given model grid boxes.

INPUT

`blon`	The longitude (in radians) at grid box boundaries. [real, dimension(:)]
`blat`	The latitude (in radians) at grid box boundaries. [real, dimension(:)]

OUTPUT

`ocean_frac`	The fractional amount (0 to 1) of ocean in a grid box. The size of this field must be size(blon)-1 by size(blat)-1. [real, dimension(:,:)]
`get_ocean_frac`	A logical value of TRUE is returned if the output field was successfully created. A value of FALSE may be returned if the Navy 1/6 degree percent water data set was not readable. [logical]

d. Get_ocean_mask

```
flag = <B> get_ocean_mask </B> ( blon, blat, ocean_mask )
```

DESCRIPTION Returns a land-ocean mask in the given model grid boxes.

INPUT

`blon`	The longitude (in radians) at grid box boundaries. [real, dimension(:)]
`blat`	The latitude (in radians) at grid box boundaries. [real, dimension(:)]

OUTPUT

`ocean_frac`	The fractional amount (0 to 1) of ocean in a grid box. The size of this field must be size(blon)-1 by size(blat)-1. [real, dimension(:,:)]
`get_ocean_mask`	A logical value of TRUE is returned if the output field was successfully created. A value of FALSE may be returned if the Navy 1/6 degree percent water data set was not readable. [logical]

e. Get_water_frac

```
flag = <B> get_water_frac </B> ( blon, blat, water_frac )
```

DESCRIPTION Returns the percent of water in a grid box.

INPUT

`blon`	The longitude (in radians) at grid box boundaries. [real, dimension(:)]
`blat`	The latitude (in radians) at grid box boundaries. [real, dimension(:)]

OUTPUT

water_frac	The fractional amount (0 to 1) of water in a grid box. The size of this field must be size(blon)-1 by size(blat)-1. [real, dimension(:,:)]
get_water_frac	A logical value of TRUE is returned if the output field was successfully created. A value of FALSE may be returned if the Navy 1/6 degree percent water data set was not readable. [logical]

f. Get_water_mask

```
flag = <B> get_water_mask </B> ( blon, blat, water_mask )
```

DESCRIPTION Returns a land-water mask in the given model grid boxes.

INPUT

blon	The longitude (in radians) at grid box boundaries. [real, dimension(:)]
blat	The latitude (in radians) at grid box boundaries. [real, dimension(:)]

OUTPUT

water_mask	A binary mask for water (true) or land (false). The size of this field must be size(blon)-1 by size(blat)-1. [real, dimension(:,:)]
get_water_mask	A logical value of TRUE is returned if the output field was successfully created. A value of FALSE may be returned if the Navy 1/6 degree percent water data set was not readable. [logical]

6.211.6 Namelist

&topography_nml topog_file Name of topography file. [character, default: DATA/navy_topography.data] water_file Name of percent water file. [character, default: DATA/navy_pctwater.data]

6.211.7 Data sets

This module uses the 1/6 degree U.S. Navy mean topography and percent water data sets. These data sets have been re-formatted to separate 32-bit IEEE files. The names of these files is specified by the namelist input. The format for both files is as follows:

```
record = 1    nlon, nlat
record = 2    blon, blat
record = 3    data
```

where:

```
nlon, nlat = The number of longitude and latitude points
             in the horizontal grid.  For the 1/6 degree
             data sets this is 2160 x 1080. [integer]
```

(continues on next page)

(continued from previous page)

```
blon, blat = The longitude and latitude grid box boundaries in degrees.
                [real :: blon(nlon+1), blat(nlat+1)]

data       = The topography or percent water data.
               [real :: data(nlon,nlat)]
```

6.211.8 Error messages

FATAL in get_topog_mean shape(zmean) is not equal to (/size(blon)-1,size(blat)-1/)) Check the input grid size and output field size.

FATAL in get_water_frac shape(water_frac) is not equal to (/size(blon)-1,size(blat)-1/)) Check the input grid size and output field size.

6.211.9 References

None.

6.211.10 Compiler specifics

None.

6.211.11 Precompiler options

None.

6.211.12 Loader options

None.

6.211.13 Test PROGRAM

To run this program you will need the topography and percent water data sets and use the following namelist (in file input.nml). &gaussian_topog_nml height = 5000., 3000., 3000., 3000., olon = 90., 255., 285., 0., olat = 45., 45., -15., -90., wlon = 15., 10., 5., 180., wlat = 15., 25., 25., 20., / program test test program for topography and gaussian_topog modules

```
 use topography_mod
 implicit none

 integer, parameter :: nlon=24, nlat=18
 real :: x(nlon), y(nlat), xb(nlon+1), yb(nlat+1), z(nlon,nlat)
 real :: hpi, rtd
 integer :: i,j
 logical :: a

gaussian mountain parameters
 real, parameter :: ht=4000.
 real, parameter :: x0=90., y0=45. ! origin in degrees
 real, parameter :: xw=15., yw=15. ! half-width in degees
 real, parameter :: xr=30., yr= 0. ! ridge-width in degrees

create lat/lon grid in radians
   hpi = acos(0.0)
   rtd = 90./hpi ! rad to deg
   do i=1,nlon
     xb(i) = 4.*hpi*real(i-1)/real(nlon)
   enddo
     xb(nlon+1) = xb(1)+4.*hpi
     yb(1) = -hpi
   do j=2,nlat
     yb(j) = yb(j-1) + 2.*hpi/real(nlat)
   enddo
     yb(nlat+1) = hpi
mid-point of grid boxes
   x(1:nlon) = 0.5*(xb(1:nlon)+xb(2:nlon+1))
   y(1:nlat) = 0.5*(yb(1:nlat)+yb(2:nlat+1))
test topography_mod routines
   a = get_topog_mean(xb,yb,z)
   call printz ('get_topog_mean')

   a = get_water_frac(xb,yb,z)
   z = z*100. ! in percent
   call printz ('get_water_frac')

   a = get_ocean_frac(xb,yb,z)
   z = z*100. ! in percent
   call printz ('get_ocean_frac')

test gaussian_topog_mod routines
   a = .true.
   z = get_gaussian_topog(x,y,ht,x0,y0,xw,yw,xr,yr)
   call printz ('get_gaussian_topog')

   call gaussian_topog_init (x,y,z)
   call printz ('gaussian_topog_init')

 contains
```

(continues on next page)

(continued from previous page)

```
simple printout of topog/water array
   subroutine printz (lab)
   character(len=*), intent(in) :: lab
    if (a) then
       print '(/a)', trim(lab)
    else
       print '(/a)', 'no data available: '//trim(lab)
       return
    endif
print full grid
       print '(3x,25i5)', (nint(x(i)*rtd),i=1,nlon)
     do j=nlat,1,-1
       print '(i3,25i5)', nint(y(j)*rtd), (nint(z(i,j)),i=1,nlon)
     enddo
   end subroutine printz

 end program test
```

6.211.14 Notes

None.

6.212 module gaussian_topog_mod

6.212.1 Overview

Routines for creating Gaussian-shaped land surface topography for latitude-longitude grids.

Interfaces generate simple Gaussian-shaped mountains from parameters specified by either argument list or namelist input. The mountain shapes are controlled by the height, half-width, and ridge-width parameters.

6.212.2 Other modules used

```
      fms_mod
constants_mod
```

6.212.3 Public interface

```
use gaussian_topog_mod [, only:  gaussian_topog_init,
                                 get_gaussian_topog ]
```

gaussian_topog_init: Returns a surface height field that consists of the sum of one or more Gaussian-shaped mountains.

get_gaussian_topog: Returns a simple surface height field that consists of a single Gaussian-shaped mountain.

6.212.4 Public data

None.

6.212.5 Public routines

a. **Gaussian_topog_init**

```
<B>call gaussian_topog_init </B> ( lon, lat, zsurf )
```

DESCRIPTION Returns a land surface topography that consists of a "set" of simple Gaussian-shaped mountains. The height, position, width, and elongation of the mountains can be controlled by variables in namelist &gaussian_topog_nml.

INPUT

lon	The mean grid box longitude in radians. [real, dimension(:)]
lat	The mean grid box latitude in radians. [real, dimension(:)]

OUTPUT

zsurf	The surface height (in meters). The size of this field must be size(lon) by size(lat). [real, dimension(:,:)]

b. Get_gaussian_topog

```
zsurf = <B> get_gaussian_topog </B> ( lon, lat, height [, olond, olatd, wlond, wlatd, rlond, rlatd ] )
```

DESCRIPTION Returns a single Gaussian-shaped mountain. The height, position, width, and elongation of the mountain is controlled by optional arguments.

INPUT

`lon`	The mean grid box longitude in radians. [real, dimension(:)]
`lat`	The mean grid box latitude in radians. [real, dimension(:)]
`height`	Maximum surface height in meters. [real, dimension(scalar)]
`olond, olatd`	Position/origin of mountain in degrees longitude and latitude. This is the location of the maximum height. [real, dimension(scalar)]
`wlond, wlatd`	Gaussian half-width of mountain in degrees longitude and latitude. [real, dimension(scalar)]
`rlond, rlatd`	Ridge half-width of mountain in degrees longitude and latitude. This is the elongation of the maximum height. [real, dimension(scalar)]

OUTPUT

`zsurf`	The surface height (in meters). The size of the returned field is size(lon) by size(lat). [real, dimension(:,:)]

NOTE Mountains do not wrap around the poles.

6.212.6 Namelist

&gaussian_topog_nml `height` Height in meters of the Gaussian mountains. [real, dimension(mxmtns), units: meter, default: 0.] `olon, olat` The longitude and latitude of mountain origins (in degrees). [real, dimension(mxmtns), units: degree, default: 0.] `wlon, wlat` The longitude and latitude half-width of mountain tails (in degrees). [real, dimension(mxmtns), units: degree, default: 0.] `rlon, rlat` The longitude and latitude half-width of mountain ridges (in degrees). For a “standard” Gaussian mountain set rlon=rlat=0. [real, dimension(mxmtns), units: degree, default: 0.] `NOTE` The variables in this namelist are only used when routine <TT>gaussian_topog_init</TT> is called. The namelist variables are dimensioned (by 10), so that multiple mountains can be generated. Internal parameter mxmtns = 10. By default no mountains are generated. []

6.212.7 Data sets

None.

6.212.8 Error messages

FATAL in get_gaussian_topog shape(zsurf) is not equal to (/size(lon),size(lat)/) Check the input grid size and output field size. The input grid is defined at the midpoint of grid boxes.

6.212.9 References

None.

6.212.10 Compiler specifics

None.

6.212.11 Precompiler options

None.

6.212.12 Loader options

None.

6.212.13 Test PROGRAM

None.

6.212.14 Notes

NAMELIST FOR GENERATING GAUSSIAN MOUNTAINS * multiple mountains can be generated * the final mountains are the sum of all height = height in meters olon, olat = longitude,latitude origin (degrees) rlon, rlat = longitude,latitude half-width of ridge (degrees) wlon, wlat = longitude,latitude half-width of tail (degrees) Note: For the standard gaussian mountain set rlon = rlat = 0 .

```
   height -->   ___________________________
               /                           \
              /              |              \
gaussian     /               |               \
  sides --> /                |                \
           /               olon                \
     _____/                olat                 \______

          |    |             |
          |<-->|<----------->|
          |wlon|    rlon     |
           wlat     rlat
```

See the topography module documentation for a test program.

6.213 module diag_manager_mod

6.213.1 Overview

<TT>diag_manager_mod</TT> is a set of simple calls for parallel diagnostics on distributed systems. It is geared toward the writing of data in netCDF format.

<TT>diag_manager_mod</TT> provides a convenient set of interfaces for writing data to disk. It is built upon the parallel I/O interface <TT>mpp_io</TT>. A single group of calls to the <TT>diag_manager_mod</TT> interfaces provides data to disk at any number of sampling and/or averaging intervals specified at run-time. Run-time specification of diagnostics are input through the diagnostics table, which is described in the *diag_table_tk* documentation. <B>Features of <TT>diag_manager_mod</TT> include:</B> Simple, minimal API. Run-time choice of diagnostics. Self-describing files: comprehensive header information (metadata) in the file itself. Strong parallel write performance. Integrated netCDF capability: netCDF is a data format widely used in the climate/weather modeling community. netCDF is considered the principal medium of data storage for <TT>diag_manager_mod</TT>. Raw unformatted fortran I/O capability is also available. Requires off-line post-processing: a tool for this purpose, <TT>mppnccombine</TT>, is available. GFDL users may use <TT>~hnv/pub/mppnccombine</TT>. Outside users may obtain the source here. It can be compiled on any C compiler and linked with the netCDF library. The program is free and is covered by the GPL license.

6.213.2 Other modules used

```
time_manager_mod
      mpp_io_mod
         fms_mod
   diag_axis_mod
 diag_output_mod
```

6.213.3 Public interface

```
use diag_manager_mod [, only:  send_data,
                               register_diag_field,
                               register_static_field,
                               diag_manager_end,
                               diag_manager_init,
                               get_base_time,
                               get_base_date,
                               need_data ]
```

send_data: Send data over to output fields.

register_diag_field: Register Diagnostic Field.

register_static_field: Register Static Field.

diag_manager_end: Exit Diagnostics Manager.

diag_manager_init: Initialize Diagnostics Manager.

get_base_time: Return base time for diagnostics.

get_base_date: Return base date for diagnostics.

need_data: Determine whether data is needed for the current model time step.

6.213.4 Public data

None.

6.213.5 Public routines

a. **Send_data**

DESCRIPTION send_data is overloaded for 1 to 3-d arrays. diag_field_id corresponds to the id returned from a previous call to register_diag_field. The field array is restricted to the computational range of the array. Optional argument is_in can be used to update sub-arrays of the entire field. Additionally, an optional logical or real mask can be used to apply missing values to the array. For the real mask, the mask is applied if the mask value is less than 0.5. The weight array is currently not implemented.

INPUT

`diag_field_id`	[integer] [integer] [integer] [integer]
`field`	[real] [real, dimension(:)] [real, dimension(:,:)] [real, dimension(:,:,:)]
`time`	[time_type] [time_type] [time_type] [time_type]

b. Register_diag_field

```
register_diag_field (module_name, field_name, axes, init_time, & long_name, units,
↪ missing_value, range)
```

DESCRIPTION Return field index for subsequent calls to send_data

INPUT

module_name	[character(len=*)]
field_name	[character(len=*)]
axes	[integer, dimension(:)]
init_time	[time_type]
long_name	[character(len=*)]
units	[character(len=*)]
missing_value	[real]
range	[real, dimension(2)]

c. Register_static_field

```
register_static_field (module_name, field_name, axes, & long_name, units, missing_
↪value, range, require)
```

DESCRIPTION Return field index for subsequent call to send_data.

INPUT

module_name	[character(len=*)]
field_name	[character(len=*)]
axes	[integer, dimension(:)]
long_name	[character(len=*)]
units	[character(len=*)]
missing_value	[real]
range	[real, dimension(2)]

d. Diag_manager_end

```
call diag_manager_end (time)
```

DESCRIPTION Flushes diagnostic buffers where necessary. Close diagnostics files.

INPUT

TIME	[time_type]

e. **Diag_manager_init**

```
call diag_manager_init ()
```

DESCRIPTION Open and read diag_table. Select fields and files for diagnostic output.

f. **Get_base_time**

```
call get_base_time ()
```

DESCRIPTION Return base time for diagnostics (note: base time must be >= model time).

g. **Get_base_date**

```
call get_base_date (year, month, day, hour, minute, second)
```

DESCRIPTION Return date information for diagnostic reference time.

h. **Need_data**

```
need_data (diag_field_id,next_model_time)
```

DESCRIPTION Determine whether data is needed for the current model time step. Since diagnostic data are buffered, the "next" model time is passed instead of the current model time. This call can be used to minimize overhead for complicated diagnostics.

INPUT

`inext_model_time`	next_model_time = current model time + model time_step [time_type]
`diag_field_id`	[integer]

6.213.6 Data sets

None.

6.213.7 Error messages

None.

6.213.8 References

None.

6.213.9 Compiler specifics

COMPILING AND LINKING SOURCE Any module or program unit using <TT>diag_manager_mod</TT> must contain the line

```
use diag_manager_mod
```

If netCDF output is desired, the cpp flag <TT>-Duse_netCDF</TT> must be turned on. The loader step requires an explicit link to the netCDF library (typically something like <TT>-L/usr/local/lib -lnetcdf</TT>, depending on the path to the netCDF library). netCDF release 3 for fortran is required.

6.213.10 Precompiler options

PORTABILITY <TT>diag_manager_mod</TT> uses standard f90.

6.213.11 Loader options

GFDL users can checkout diag_manager_mod using the cvs command <TT>setenv CVSROOT '/home/fms/cvs';cvs co diag_manager</TT>.

```
ACQUIRING SOURCE
```

6.213.12 Test PROGRAM

None.

6.213.13 Notes

None.

6.214 diag_table_tk

6.214.1 Overview

The script diag_table_tk is a GUI written in Perl/Tk for building diagnostics tables, which are used by the *module diag_manager_mod* for run-time specification of diagnostics.

The diagnostics table allows users to specify sampling rates and the choice of fields at run time. The table consists of comma-separated ASCII values and may be hand-edited. The preferred method of building a table is to use the provided GUI interface diag_table_tk. A default diag table is provided with each runscript.

The table is separated into three sections.

1. **Global section:** The first two lines of the table contain the experiment title and base date. The base date is the reference time used for the time units. The base date must be greater than or equal to the model start date. The date consists of six space-separated integers: year, month, day, hour, minute, and second.
2. **File section:** File lines contain 6 fields - file name, output frequency, output frequency units, file format (currently only support NetCDF), time units and long name for time axis. The format is as follows:

```
"file_name", output_freq, "output_freq_units", format, "time_units", "time_long_
↪name"


output_freq:
         > 0  output frequency in "output_units"
         = 0  output frequency every time step
         =-1  output frequency at end of run

output_freq_units = units used for output frequency
        (years, months, days, minutes, hours, seconds)

format:   1 NetCDF

time_units   = units used to label the time axis
         (days, minutes, hours, seconds)
```

3. **Field section:** Field lines contain 8 fields - module name, field name, output field name, file name, time sampling (for averaging, currently only support all timesteps), time average, other operations (currently not implemented) and pack value (1,2,4 or 8). The format is as follows:

```
"module_name", "field_name", "output_name", "file_name" "time_sampling",
time_avg, "other_opts", packing

module_name :  e.g. "atmos_mod", "land_mod"

time_avg = .true. or .false.

packing  = 1  double precision
         = 2  float
         = 4  packed 16-bit integers
         = 8  packed 1-byte (not tested?)
```

6.214.2 Installation

diag_table_tk requires the following perl modules:

```
use English;
use Tk;
use Cwd;
require Tk::FileSelect;
require Tk::Text;
use Tk::widgets qw/Dialog ErrorDialog ROText/;
use Tk::FileDialog;
use Tk::Balloon;
use File::Find;
```

Most of these are built by default in perl 5.004 and above; however, you may need to install the perl Tk modules.

Obtain Tk and Tk-FileDialog from:
http://www.cpan.org

Obtain Tk and Tk-FileDialog in RPM (red hat package manager) format from:
http://rpmfind.net/linux/rpm2html/search.php?query=perl-Tk
http://rpmfind.net/linux/rpm2html/search.php?query=perl-Tk-FileDialog

6.214.3 Usage

1. **Load and edit previously saved diag tables.** Choose “Load Table” from the “File” menu.
2. **Quick parsing of f90 source code for fields which may be registered. Fields are grouped by module name.** To obtain a list of available diagnostic fields, choose “Modify Output Field Entry” from the main menu. Enter the path to the directory containing your source code, and click “Search”. After the search is complete, you can look in the “Field” menu for the list of available fields.
3. **Easy table editing, including ability to delete or edit selected lines.** To edit the text of an entry, click the “Show Table” button, Select the entry you wish to edit by clicking on the “Entry List” button, then click “Edit Entry”. A new window will open in which you can make changes. Click “Save Changes” when you are finished.
4. **Error checks to help ensure that your diag table will work properly.** Ensures proper spacing and formatting.
5. **Online Help is available.** Choose “Help” from the menubar.

6.214.4 Bugs and future plans

The “cancel” button doesn’t seem to work. The Show Table window should be opened by default when you click “modify table a” or “modify table b”. Visual feedback is good.

It should warn you if you make changes and quit without saving.

6.215 module bgrid_polar_filter_mod

6.216 module bgrid_halo_mod

6.217 module bgrid_horiz_mod

6.218 module bgrid_cold_start_mod

6.219 module bgrid_prog_var_mod

6.220 module bgrid_diagnostics_mod

6.221 module bgrid_integrals

6.222 module bgrid_change_grid_mod

6.223 module bgrid_masks_mod

6.224 module bgrid_vert_mod

6.225 module atmosphere_mod

6.226 module bgrid_core_mod

6.227 module bgrid_core_driver_mod

6.228 Module hs_forcing_mod

6.229 program atmos_model

6.230 PROGRAM `replace_wrf_fields`

6.230.1 Overview

Program to copy various fields from one WRF netCDF file to another.

There are many existing utilities to process netCDF files, i.e. the NCO operators and NCL scripts, which have more functionality than this program. The only purpose for having this one is that it is a standalone program with no prerequisites or dependencies other than the netCDF libraries. If you already have other tools available they can do the same functions that this program does.

This program copies the given data fields from the input file to the output file, failing if their sizes, shapes, or data types do not match exactly. The expected use is to copy fields which are updated by the WRF program but are not part of the DART state vector, for example, sea surface temperature or soil fields. After DART has updated the WRF restart `wrfinput_d01` file, this program can be used to update other fields in the file before running the model.

6.230.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&replace_wrf_fields_nml
   fieldnames = 'SST',
   fieldlist_file = '',
   fail_on_missing_field = .true.
   debug = .false.,
   /
```

Item	Type	Description
fieldnames	character(len=129)(:)	An array of ASCII field names to be copied from the input netCDF file to the output netCDF file. The names must match exactly, and the size and shape of the data must be the same in the input and output files for the data to be copied. If the field names are set here, the fieldlist_file item must be ' '.
fieldlist_file	character(len=129)	An alternative to an explicit list of field names to copy. This is a single string, the name of a file which contains a single field name, one per line. If this option is set, the fieldnames namelist item must be ' '.
fail_on_missing_field	logical	If any fields in the input list are not found in either the input or output netcdf files, fail if this is set to true. If false, a warning message will be printed but execution will continue.
debug	logical	If true, print out debugging messages about which fields are found in the input and output files.

6.230.3 Modules used

```
types_mod
utilities_mod
parse_args_mod
```

6.230.4 Files

- input namelist ; `input.nml`
- Input - output WRF state netCDF files; `wrfinput_d01, wrfinput_d02, ...`
- fieldlist_file (if specified in namelist)

File formats

This utility works on any pair of netCDF files, doing a simple read and copy from one to the other.

6.230.5 References

- none

6.231 PROGRAM `wrf_dart_obs_preprocess`

6.231.1 Overview

Program to preprocess observations, with specific knowledge of the WRF domain.

This program will exclude all observations outside of the given WRF domain. There are options to exclude or increase the error values of obs close to the domain boundaries. The program can superob (average) aircraft and satellite wind obs if they are too dense.

This program can read up to 9 additional obs_seq files and merge their data in with the basic obs_sequence file which is the main input.

This program can reject surface observations if the elevation encoded in the observation is too different from the wrf surface elevation.

This program can exclude observations above a specified height or pressure.

This program can overwrite the incoming Data QC value with another.

6.231.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&wrf_obs_preproc_nml

  file_name_input          = 'obs_seq.old'
  file_name_output         = 'obs_seq.new'

  sonde_extra              = 'obs_seq.rawin'
  land_sfc_extra           = 'obs_seq.land_sfc'
  metar_extra              = 'obs_seq.metar'
  marine_sfc_extra         = 'obs_seq.marine'
  sat_wind_extra           = 'obs_seq.satwnd'
  profiler_extra           = 'obs_seq.profiler'
```

(continues on next page)

(continued from previous page)

```
  gpsro_extra              = 'obs_seq.gpsro'
  acars_extra              = 'obs_seq.acars'
  trop_cyclone_extra       = 'obs_seq.tc'

  overwrite_obs_time       = .false.

  obs_boundary             = 0.0
  increase_bdy_error       = .false.
  maxobsfac                = 2.5
  obsdistbdy               = 15.0

  sfc_elevation_check      = .false.
  sfc_elevation_tol        = 300.0
  obs_pressure_top         = 0.0
  obs_height_top           = 2.0e10

  include_sig_data         = .true.
  tc_sonde_radii           = -1.0

  superob_aircraft         = .false.
  aircraft_horiz_int       = 36.0
  aircraft_pres_int        = 2500.0

  superob_sat_winds        = .false.
  sat_wind_horiz_int       = 100.0
  sat_wind_pres_int        = 2500.0

  overwrite_ncep_satwnd_qc = .false.
  overwrite_ncep_sfc_qc    = .false.
/
```

Item

Type

Description

Generic parameters:

file_name_input

character(len=129)

The input obs_seq file.

file_name_output

character(len=129)

The output obs_seq file.

sonde_extra, land_sfc_extra, metar_extra, marine_sfc_extra, marine_sfc_extra, sat_wind_extra, profiler_extra, gpsro_extra, acars_extra, trop_cyclone_extra

character(len=129)

The names of additional input obs_seq files, which if they exist, will be merged in with the obs from the `file_name_input` obs_seq file. If the files do not exist, they are silently ignored without error.

overwrite_obs_time

logical

If true, replace the incoming observation time with the analysis time. Not recommended.

Boundary-specific parameters:

obs_boundary

real(r8)

Number of grid points around domain boundary which will be considered the new extent of the domain. Observations outside this smaller area will be excluded.

increase_bdy_error

logical

If true, observations near the domain boundary will have their observation error increased by `maxobsfac`.

maxobsfac

real(r8)

If `increase_bdy_error` is true, multiply the error by a ramped factor. This item sets the maximum error.

obsdistbdy

real(r8)

If `increase_bdy_error` is true, this defines the region around the boundary (in number of grid points) where the observation error values will be altered. This is ramped, so when you reach the innermost points the change in observation error is 0.0.

Parameters to reduce observation count :

sfc_elevation_check

logical

If true, check the height of surface observations against the surface height in the model.

sfc_elevation_tol

real(r8)

If `sfc_elevation_check` is true, the maximum difference between the elevation of a surface observation and the model surface height, in meters. If the difference is larger than this value, the observation is excluded.

obs_pressure_top

real(r8)

Observations with a vertical coordinate in pressure which are located above this pressure level (i.e. the obs vertical value is smaller than the given pressure) will be excluded.

obs_height_top

real(r8)

Observations with a vertical coordinate in height which are located above this height value (i.e. the obs vertical value is larger than the given height) will be excluded.

Radio/Rawinsonde-specific parameters :

include_sig_data

logical

If true, include significant level data from radiosondes.

tc_sonde_radii

real(r8)

If greater than 0.0 remove any sonde observations closer than this distance in Kilometers to the center of a Tropical Cyclone.

Aircraft-specific parameters :

superob_aircraft

logical

If true, average all aircraft observations within the given radius and output only a single observation. Any observation that is used in computing a superob observation is removed from the list and is not used in any other superob computation.

aircraft_horiz_int

real(r8)

If `superob_aircraft` is true, the horizontal distance in Kilometers which defines the superob area. All other unused aircraft observations within this radius will be averaged with the current observation.

aircraft_vert_int

real(r8)

If `superob_aircraft` is true, the vertical distance in Pascals which defines the maximum separation for including an observation in the superob computation.

Satellite Wind-specific parameters :

superob_sat_winds

logical

If true, average all sat_wind observations within the given radius and output only a single observation. Any observation that is used in computing a superob observation is removed from the list and is not used in any other superob computation.

sat_wind_horiz_int

real(r8)

If `superob_sat_winds` is true, the horizontal distance in Kilometers which defines the superob area. All other unused sat_wind observations within this radius will be averaged with the current observation.

sat_wind_vert_int

real(r8)

If `superob_sat_winds` is true, the vertical distance in Pascals which defines the maximum separation for including an observation in the superob computation.

overwrite_ncep_satwnd_qc

logical

If true, replace the incoming Data QC value in satellite wind observations with 2.0.

Surface Observation-specific parameters :

overwrite_ncep_sfc_qc

logical

If true, replace the incoming Data QC value in surface observations with 2.0.

6.231.3 Modules used

```
types_mod
obs_sequence_mod
utilities_mod
obs_kind_mod
time_manager_mod
model_mod
netcdf
```

6.231.4 Files

- Input namelist ; `input.nml`
- Input WRF state netCDF files; `wrfinput_d01, wrfinput_d02, ...`
- Input obs_seq files (as specified in namelist)
- Output obs_seq file (as specified in namelist)

File formats

This utility can read one or more obs_seq files and combine them while doing the rest of the processing. It uses the standard DART observation sequence file format.

6.231.5 References

- Generously contributed by Ryan Torn.

6.232 PROGRAM `netcdf_to_gitm_blocks`

The Global Ionosphere Thermosphere Model (GITM) is a 3-dimensional spherical code that models the Earth's thermosphere and ionosphere system using a stretched grid in latitude and altitude. For a fuller description of using GITM within DART, please see the *GITM*.

`netcdf_to_gitm_blocks` is the program that updates the GITM restart files (i.e. `b?????.rst`) with the information from a DART output/restart file (e.g. `perfect_ics, filter_ics, ...`).

The list of variables used to create the DART state vector are specified in the `input.nml` file.

Conditions required for successful execution of `netcdf_to_gitm_blocks`:

- a valid `input.nml` namelist file for DART

- a valid UAM.in control file for GITM
- a set of b?????.rst data files for GITM
- a header.rst file for GITM
- the DART/GITM interfaces must be compiled in a manner consistent with the GITM data and control files. The following GITM source files are required to build *any* DART interface:
  - models/gitm/GITM2/src/ModConstants.f90
  - models/gitm/GITM2/src/ModEarth.f90
  - models/gitm/GITM2/src/ModKind.f90
  - models/gitm/GITM2/src/ModSize.f90
  - models/gitm/GITM2/src/ModTime.f90
  - models/gitm/GITM2/src/time_routines.f90

  Versions of these are included in the DART release. ModSize.f90, in particular, must match what was used to create the b????.rst files.

The individual model instances are run in unique directories. This is also where the converter routines gitm_to_dart and netcdf_to_gitm_blocks are run. This makes it easy to use a single 'static' name for the input and output filenames. advance_model.csh is responsibile for linking the appropriate files to these static filenames.

The simplest way to test the converter is to compile GITM and run a single model state forward using work/clean.sh. To build GITM ... download GITM and unpack the code into DART/models/gitm/GITM2 and follow these instructions:

```
cd models/gitm/GITM2
./Config.pl -install -compiler=ifortmpif90 -earth
make
cd ../work
./clean.sh 1 1 0 150.0 170.0 1.0
```

And then manually run netcdf_to_gitm_blocks on the result.

6.232.1 Namelist

We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&netcdf_to_gitm_blocks_nml
   netcdf_to_gitm_blocks_output_file = 'dart_restart',
   advance_time_present      = .false.
   /

&model_nml
   gitm_restart_dirname          = 'advance_temp_e1/UA/restartOUT',
   assimilation_period_days      = 0,
   assimilation_period_seconds   = 1800,
   model_perturbation_amplitude = 0.2,
   output_state_vector           = .false.,
   calendar                      = 'Gregorian',
   debug                         = 0,
```

(continues on next page)

(continued from previous page)

```
   gitm_state_variables = 'Temperature',           'QTY_TEMPERATURE',
                          'eTemperature',          'QTY_TEMPERATURE_ELECTRON',
                          'ITemperature',          'QTY_TEMPERATURE_ION',
                          'iO_3P_NDensityS',       'QTY_DENSITY_NEUTRAL_O3P',
   ...
```

Contents	Type	Description
netcdf_to_gitm_blocks_output_file	character(len=128)	The name of the DART file containing the model state derived from the GITM restart files.
advance_time_present	logical	If you are manually converting a DART initial conditions or restart file this should be `.false.`; these files have a single timestamp describing the valid time of the model state. If `.true.`, TWO timestamps are expected in the DART file header and `DART_GITM_time_control.txt`) is created with the settings appropriate to advance GITM to the time requested by DART.

The full description of the `model_nml` namelist is documented in the gitm model_mod, but the most important variable for `netcdf_to_gitm_blocks` is repeated here.

Contents	Type	Description
gitm_restart_dirname	character(len=256)	The name of the directory containing the GITM restart files and runtime control information.
gitm_state_variables	character(len=32), dimension(2,80)	The list of variable names in the gitm restart file to use to create the DART state vector and their corresponding DART kind. The default list is specified in model_mod.nml

6.232.2 Modules used

```
obs_def_upper_atm_mod.f90
assim_model_mod.f90
types_mod.f90
location/threed_sphere/location_mod.f90
models/gitm/GITM2/src/ModConstants.f90
models/gitm/GITM2/src/ModEarth.f90
models/gitm/GITM2/src/ModKind.f90
models/gitm/GITM2/src/ModSize.f90
models/gitm/GITM2/src/ModTime.f90
models/gitm/GITM2/src/time_routines.f90
models/gitm/dart_gitm_mod.f90
models/gitm/netcdf_to_gitm_blocks.f90
models/gitm/model_mod.f90
null_mpi_utilities_mod.f90
obs_kind_mod.f90
random_seq_mod.f90
time_manager_mod.f90
utilities_mod.f90
```

6.232.3 Files read

- gitm restart files: `b????.rst`
- gitm control files: `header.rst`
- gitm control files: `UAM.in.rst`
- DART namelist file: `input.nml`

6.232.4 Files written

- DART initial conditions/restart file; e.g. `dart_ics`

6.232.5 References

- The official `GITM` site is: can be found at ccmc.gsfc.nasa.gov/models/modelinfo.php?model=GITM

6.233 PROGRAM `gitm_blocks_to_netcdf`

The Global Ionosphere Thermosphere Model (GITM) is a 3-dimensional spherical code that models the Earth's thermosphere and ionosphere system using a stretched grid in latitude and altitude. For a fuller description of using GITM within DART, please see the *GITM*.

`gitm_blocks_to_netcdf` is the program that reads GITM restart files (i.e. `b?????.rst`) and creates a DART output/restart file (e.g. `perfect_ics, filter_ics, ...`).

The list of variables used to create the DART state vector are specified in the `input.nml` file.

Conditions required for successful execution of `gitm_blocks_to_netcdf`:

- a valid `input.nml` namelist file for DART
- a valid `UAM.in` control file for GITM
- a set of `b?????.rst` data files for GITM
- a `header.rst` file for GITM
- the DART/GITM interfaces must be compiled in a manner consistent with the GITM data and control files. The following GITM source files are required to build *any* DART interface:
  - models/gitm/GITM2/src/ModConstants.f90
  - models/gitm/GITM2/src/ModEarth.f90
  - models/gitm/GITM2/src/ModKind.f90
  - models/gitm/GITM2/src/ModSize.f90
  - models/gitm/GITM2/src/ModTime.f90
  - models/gitm/GITM2/src/time_routines.f90

  Versions of these are included in the DART release. `ModSize.f90`, in particular, must match what was used to create the `b????.rst` files.

The individual model instances are run in unique directories. This is also where the converter routines gitm_blocks_to_netcdf and dart_to_gitm are run. This makes it easy to use a single 'static' name for the input and output filenames. advance_model.csh is responsibile for linking the appropriate files to these static filenames.

The simplest way to test the converter is to compile GITM and run a single model state forward using work/clean.sh. To build GITM … download GITM and unpack the code into DART/models/gitm/GITM2 and follow these instructions:

```
cd models/gitm/GITM2
./Config.pl -install -compiler=ifortmpif90 -earth
make
cd ../work
./clean.sh 1 1 0 150.0 170.0 1.0
```

6.233.1 Namelist

We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/' for all our namelist input. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&gitm_blocks_to_netcdf_nml
   gitm_blocks_to_netcdf_output_file = 'dart_ics',
   /

&model_nml
   gitm_restart_dirname          = 'advance_temp_e1/UA/restartOUT',
   assimilation_period_days      = 0,
   assimilation_period_seconds   = 1800,
   model_perturbation_amplitude = 0.2,
   output_state_vector           = .false.,
   calendar                      = 'Gregorian',
   debug                         = 0,
   gitm_state_variables = 'Temperature',            'QTY_TEMPERATURE',
                          'eTemperature',           'QTY_TEMPERATURE_ELECTRON',
                          'ITemperature',           'QTY_TEMPERATURE_ION',
                          'iO_3P_NDensityS',        'QTY_DENSITY_NEUTRAL_O3P',
   ...
```

Contents	Type	Description
gitm_blocks_to_netcdf_output_file	character(len=128)	The name of the DART file containing the model state derived from the GITM restart files.

The full description of the model_nml namelist is documented in the gitm model_mod, but the most important variable for gitm_blocks_to_netcdf is repeated here.

Contents	Type	Description
gitm_restart_dirname	character(len=256)	The name of the directory containing the GITM restart files and runtime control information.
gitm_state_variables	character(len=32), dimension(2,80)	The list of variable names in the gitm restart file to use to create the DART state vector and their corresponding DART kind. The default list is specified in model_mod.nml

6.233.2 Modules used

```
obs_def_upper_atm_mod.f90
assim_model_mod.f90
types_mod.f90
location/threed_sphere/location_mod.f90
models/gitm/GITM2/src/ModConstants.f90
models/gitm/GITM2/src/ModEarth.f90
models/gitm/GITM2/src/ModKind.f90
models/gitm/GITM2/src/ModSize.f90
models/gitm/GITM2/src/ModTime.f90
models/gitm/GITM2/src/time_routines.f90
models/gitm/dart_gitm_mod.f90
models/gitm/gitm_blocks_to_netcdf.f90
models/gitm/model_mod.f90
null_mpi_utilities_mod.f90
obs_kind_mod.f90
random_seq_mod.f90
time_manager_mod.f90
utilities_mod.f90
```

6.233.3 Files read

- gitm restart files: `b????.rst`
- gitm control files: `header.rst`
- gitm control files: `UAM.in.rst`
- DART namelist file: `input.nml`

6.233.4 Files written

- DART initial conditions/restart file; e.g. `dart_ics`

6.233.5 References

- The official `GITM` site is: can be found at ccmc.gsfc.nasa.gov/models/modelinfo.php?model=GITM

6.234 MODULE model_mod

6.234.1 Overview

Every model that is DART compliant must provide an set of interfaces that will be called by DART code. For models which have no special code for some of these routines, they can pass through the call to this default module, which satisfies the call but does no work. To use these routines in a `model_mod.f90`, add at the top:

```
use default_model_mod, only : xxx, yyy
```

and then leave them in the public list.

6.234.2 Namelist

The default routines have no namelist.

6.234.3 Other modules used

```
types_mod
time_manager_mod
location_mod
utilities_mod
netcdf_utilities_mod
ensemble_manager_mod
dart_time_io_mod
```

6.234.4 Public interfaces

use model_mod, only :	get_model_size
	adv_1step
	get_state_meta_data
	model_interpolate
	shortest_time_between_assimilations
	static_init_model
	init_time
	fail_init_time
	init_conditions
	fail_init_conditions
	nc_write_model_atts
	nc_write_model_vars
	pert_model_copies
	get_close_obs
	get_close_state
	convert_vertical_obs
	convert_vertical_state
	read_model_time
	write_model_time
	end_model

A note about documentation style. Optional arguments are enclosed in brackets *[like this]*.

model_size = get_model_size( )

```
integer(i8) :: get_model_size
```

Returns the length of the model state vector as 1. Probably not what you want. The model_mod should set this to the right size and not use this routine.

`model_size`	The length of the model state vector.

call adv_1step(x, time)

```
real(r8), dimension(:), intent(inout) :: x
type(time_type),        intent(in)    :: time
```

Throws a fatal error. If the model_mod can advance the model it should provide a real routine. This default routine is intended for use by models which cannot advance themselves from inside filter.

`x`	State vector of length model_size.
`time`	Current time of model state.

call get_state_meta_data (index_in, location, [, var_type])

```
integer,             intent(in)  :: index_in
type(location_type), intent(out) :: location
integer, optional,   intent(out) ::  var_type
```

Sets the location to missing and the variable type to 0. The model_mod should provide a routine that sets a real location and a state vector type for the requested item in the state vector.

`index_in`	Index of state vector element about which information is requested.
`location`	The location of state variable element.
var_type	The generic quantity of the state variable element.

call model_interpolate(state_handle, ens_size, location, obs_quantity, expected_obs, istatus)

```
type(ensemble_type),    intent(in)  :: state_handle
integer,                intent(in)  :: ens_size
type(location_type),    intent(in)  :: location
integer,                intent(in)  :: obs_quantity
real(r8),               intent(out) :: expected_obs(ens_size)
integer,                intent(out) :: istatus(ens_size)
```

Sets the expected obs to missing and returns an error code for all obs. This routine should be supplied by the model_mod.

`state_handle`	The handle to the state structure containing information about the state vector about which information is requested.
`ens_size`	The ensemble size.
`location`	Location to which to interpolate.
`obs_quantity`	Quantity of state field to be interpolated.
`expected_obs`	The interpolated values from the model.
`istatus`	Integer values return 0 for success. Other positive values can be defined for various failures.

var = shortest_time_between_assimilations()

```
type(time_type) :: shortest_time_between_assimilations
```

Returns 1 day.

`var`	Smallest advance time of the model.

call static_init_model()

Does nothing.

call init_time(time)

```
type(time_type), intent(out) :: time
```

Returns a time of 0.

`time`	Initial model time.

call fail_init_time(time)

```
type(time_type), intent(out) :: time
```

Throws a fatal error. This is appropriate for models that cannot start from arbitrary initial conditions.

`time`	NOT SET. Initial model time.

call init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

Returns x(:) = 0.0

x	Initial conditions for state vector.

call fail_init_conditions(x)

```
real(r8), dimension(:), intent(out) :: x
```

Throws a fatal error. This is appropriate for models that cannot start from arbitrary initial conditions.

x	NOT SET: Initial conditions for state vector.

call nc_write_model_atts(ncFileID, domain_id)

```
integer, intent(in) :: ncFileID
integer, intent(in) :: domain_id
```

Does nothing.

ncFileID	Integer file descriptor to previously-opened netCDF file.
domain_id	integer describing the domain (which can be a nesting level, a component model …) Models with nested grids are decomposed into 'domains' in DART. The concept is extended to refer to 'coupled' models where one model component may be the atmosphere, another component may be the ocean, or land, or ionosphere … these would be referenced as different domains.

call nc_write_model_vars(ncFileID, domain_id, state_ens_handle [, memberindex] [, timeindex])

```
integer,             intent(in) :: ncFileID
integer,             intent(in) :: domain_id
type(ensemble_type), intent(in) :: state_ens_handle
integer, optional,   intent(in) :: memberindex
integer, optional,   intent(in) :: timeindex
```

Does nothing

`ncFileID`	file descriptor to previously-opened netCDF file.
`domain_id`	integer describing the domain (which can be a nesting level, a component model . . .)
`state_ens_handle`	The handle to the state structure containing information about the state vector about which information is requested.
`memberindex`	Integer index of ensemble member to be written.
`timeindex`	The timestep counter for the given state.

call pert_model_copies(state_ens_handle, ens_size, pert_amp, interf_provided)

```
type(ensemble_type), intent(inout) :: state_ens_handle
integer,             intent(in)    :: ens_size
real(r8),            intent(in)    :: pert_amp
logical,             intent(out)   :: interf_provided
```

Returns 'interface provided' flag as false, so the default perturb routine in DART will add small amounts of gaussian noise to all parts of the state vector.

`state_ens_handle`	The handle containing an ensemble of state vectors to be perturbed.
`ens_size`	The number of ensemble members to perturb.
`pert_amp`	the amplitude of the perturbations. The interpretation is based on the model-specific implementation.
`interf_provided`	Returns false if model_mod cannot do this, else true.

call get_close_obs(gc, base_loc, base_type, locs, loc_qtys, loc_types, num_close, close_ind [, dist] [, state_handle)

```
type(get_close_type),           intent(in)  :: gc
type(location_type),            intent(in)  :: base_loc
integer,                        intent(in)  :: base_type
type(location_type),            intent(in)  :: locs(:)
integer,                        intent(in)  :: loc_qtys(:)
integer,                        intent(in)  :: loc_types(:)
integer,                        intent(out) :: num_close
integer,                        intent(out) :: close_ind(:)
real(r8),            optional, intent(out) :: dist(:)
type(ensemble_type), optional, intent(in)  :: state_handle
```

Passes the call through to the location module code.

`gc`	The get_close_type which stores precomputed information about the locations to speed up searching
`base_loc`	Reference location. The distances will be computed between this location and every other location in the obs list
`base_type`	The DART quantity at the `base_loc`
`locs(:)`	Compute the distance between the `base_loc` and each of the locations in this list
`loc_qtys(:)`	The corresponding quantity of each item in the `locs` list
`loc_types(:)`	The corresponding type of each item in the `locs` list. This is not available in the default implementation but may be used in custom implementations.
`num_close`	The number of items from the `locs` list which are within maxdist of the base location
`close_ind(:)`	The list of index numbers from the `locs` list which are within maxdist of the base location
`dist(:)`	If present, return the distance between each entry in the close_ind list and the base location. If not present, all items in the obs list which are closer than maxdist will be added to the list but the overhead of computing the exact distances will be skipped.
`state_handle`	The handle to the state structure containing information about the state vector about which information is requested.

call get_close_state(gc, base_loc, base_type, state_loc, state_qtys, state_indx, num_close, close_ind, dist, state_handle)

```
type(get_close_type), intent(in)    :: gc
type(location_type),  intent(inout) :: base_loc
integer,              intent(in)    :: base_type
type(location_type),  intent(inout) :: state_loc(:)
integer,              intent(in)    :: state_qtys(:)
integer(i8),          intent(in)    :: state_indx(:)
integer,              intent(out)   :: num_close
integer,              intent(out)   :: close_ind(:)
real(r8),             intent(out)   :: dist(:)
type(ensemble_type),  intent(in)    :: state_handle
```

Passes the call through to the location module code.

`gc`	The get_close_type which stores precomputed information about the locations to speed up searching
`base_loc`	Reference location. The distances will be computed between this location and every other location in the obs list
`base_type`	The DART quantity at the `base_loc`
`state_loc(:)`	Compute the distance between the `base_loc` and each of the locations in this list
`state_qtys(:)`	The corresponding quantity of each item in the `state_loc` list
`state_indx(:)`	The corresponding DART index of each item in the `state_loc` list. This is not available in the default implementation but may be used in custom implementations.
`num_close`	The number of items from the `state_loc` list which are within maxdist of the base location
`close_ind(:)`	The list of index numbers from the `state_loc` list which are within maxdist of the base location
`dist(:)`	If present, return the distance between each entry in the `close_ind` list and the base location. If not present, all items in the `state_loc` list which are closer than maxdist will be added to the list but the overhead of computing the exact distances will be skipped.
`state_handle`	The handle to the state structure containing information about the state vector about which information is requested.

call convert_vertical_obs(state_handle, num, locs, loc_qtys, loc_types, which_vert, status)

```
type(ensemble_type), intent(in)  :: state_handle
integer,             intent(in)  :: num
type(location_type), intent(in)  :: locs(:)
integer,             intent(in)  :: loc_qtys(:)
integer,             intent(in)  :: loc_types(:)
integer,             intent(in)  :: which_vert
integer,             intent(out) :: status(:)
```

Passes the call through to the location module code.

`state_handle`	The handle to the state.
`num`	the number of observation locations
`locs`	the array of observation locations
`loc_qtys`	the array of observation quantities.
`loc_types`	the array of observation types.
`which_vert`	the desired vertical coordinate system. There is a table in the `location_mod.f90` that relates integers to vertical coordinate systems.
`status`	Success or failure of the vertical conversion. If `istatus = 0`, the conversion was a success. Any other value is a failure.

call convert_vertical_state(state_handle, num, locs, loc_qtys, loc_types, which_vert, status)

```
type(ensemble_type), intent(in)  :: state_handle
integer,             intent(in)  :: num
type(location_type), intent(in)  :: locs(:)
integer,             intent(in)  :: loc_qtys(:)
integer,             intent(in)  :: loc_types(:)
integer,             intent(in)  :: which_vert
integer,             intent(out) :: status(:)
```

Passes the call through to the location module code.

`state_handle`	The handle to the state.
`num`	the number of state locations
`locs`	the array of state locations
`loc_qtys`	the array of state quantities.
`loc_types`	the array of state types.
`which_vert`	the desired vertical coordinate system. There is a table in the `location_mod.f90` that relates integers to vertical coordinate systems.
`status`	Success or failure of the vertical conversion. If `istatus = 0`, the conversion was a success. Any other value is a failure.

model_time = read_model_time(filename)

```
character(len=*), intent(in) :: filename
type(time_type)              :: model_time
```

Passes the call through to the dart_time_io module code.

filename	netCDF file name
model_time	The current time of the model state.

call write_model_time(ncid, dart_time)

```
integer,          intent(in) :: ncid
type(time_type),  intent(in) :: dart_time
```

Passes the call through to the dart_time_io module code.

ncid	handle to an open netCDF file
dart_time	The current time of the model state.

call end_model()

Does nothing.

6.234.5 Files

none

6.234.6 References

1. none

6.234.7 Private components

N/A

6.235 PROGRAM `cam_to_dart`

6.235.1 Overview

`cam_to_dart` is the program that reads a CAM restart file (usually `caminput.nc`) and creates a single DART output/restart file (e.g. `perfect_ics, filter_ics, ...`). If you have multiple input files, you will need to rename the output files as you create them.

The list of variables extracted from the CAM netCDF file and conveyed to DART is controlled by the set of `input.nml &model_nml:state_names_*` variables. The `date` and `datesec` variables in the CAM netcdf file are used to specify the valid time of the state vector. The time may be changed with the *PROGRAM restart_file_tool* if desired.

Some CAM restart files are from climatological runs and have a valid time that predates the use of the Gregorian calendar. In such instances, the year component of the original date is changed to be a valid Gregorian year (by adding 1601). A warning is issued to the screen and to the logfile. Please use the *PROGRAM restart_file_tool* to change this time.

Conditions required for successful execution of `cam_to_dart`:

- a valid `input.nml` namelist file for DART
- a CAM 'phis' netCDF file [default: `cam_phis.nc`]
- a CAM restart file [default: `caminput.nc`].

Since this program is called repeatedly for every ensemble member, we have found it convenient to link the CAM restart files to the default input filename (`caminput.nc`). The default DART output filename is `dart_ics` - this may be moved or linked as necessary.

6.235.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&cam_to_dart_nml
   cam_to_dart_input_file  = 'caminput.nc',
   cam_to_dart_output_file = 'dart_ics',
   /
```

Item	Type	Description
cam_to_dart_input_file	character(len=128)	The name of the DART file containing the CAM state.
cam_to_dart_output_file	character(len=128)	The name of the DART file containing the model state derived from the CAM restart file.

6.235.3 Modules used

```
assim_model_mod.f90
types_mod.f90
threed_sphere/location_mod.f90
model_mod.f90
null_mpi_utilities_mod.f90
obs_kind_mod.f90
random_seq_mod.f90
time_manager_mod.f90
utilities_mod.f90
```

6.235.4 Files read

- DART namelist file; `input.nml`
- CAM restart file; `caminput.nc`
- CAM "phis" file specified in `&model_nml::cam_phis` (normally `cam_phis.nc`)

6.235.5 Files written

- DART initial conditions/restart file; e.g. `dart_ics`

6.235.6 References

none

6.236 CAM

6.236.1 Overview

The DART system supports data assimilation into the Community Atmosphere Model (CAM) which is the atmospheric component of the Community Earth System Model (CESM). This DART interface is being used by graduate students, post-graduates, and scientists at universities and research labs to conduct data assimilation reseearch. Others are using the products of data assimilation (analyses), which were produced here at NCAR using CESM+DART, to conduct related research. The variety of research can be sampled on the DART Publications page.

"CAM" refers to a family of related atmospheric components, which can be built with two independent main characteristics. CESM labels these as:

resolution

where *resolution* refers to both the horizontal resolution of the grid (rather than the vertical resolution) **and** the dynamical core run on the specified grid. The dynamical core refers to the fluid dynamical equations run on the specified grid.

compset

where *compset* refers to the vertical grid **and** the parameterizations – the formulation of the subgrid-scale physics. These parameterizations encompass the equations describing physical processes such as convection, radiation, chemistry.

- The vertical grid is determined by the needs of the chosen parameterizations, thus the vertical spacing and the top level of the model domain, specified by a variable known as `ptop`, vary.
- The combinations of parameterizations and vertical grids are named: CAM3.5, CAM5, CAM#, . . . WACCM, WACCM#, WACCM-X, CAM-Chem.

There are minor characteristics choices within each of these, but only chemistry choices in WACCM and CAM-Chem have an impact on DART. As of April 2015, all of these variants are handled by the same `model_mod.f90`, namelist, and build scripts, with differences in the assimilation set up described in *Setup Variations*.

This DART+CAM interface has the following features.

- Assimilate within the CESM software framework by using the multi-instance capability of CESM1.1.1 (and later). This enables assimilation of suitable observations into multiple CESM components. The ability to assimilate in the previous mode, where DART called 'stand-alone' CAMs when needed, is not being actively supported for these CESM versions.
- Use either the eulerian, finite-volume (FV), or spectral-element (SE) dynamical core.
- Use any resolution of CAM, including refined mesh grids in CAM-SE. As of April, 2015 this is limited by the ability of the memory of a node of your hardware to contain the state vector of a single ensemble member. Work is under way to relax this restriction.
- Assimilate a variety of observations; to date the observations successfully assimilated include the NCEP reanalysis BUFR obs (T,U,V,Q), Global Positioning System radio occultation obs, and MOPITT carbon monoxide (when a chemistry model is incorporated into CAM-FV). Research has also explored assimilating surface observations, cloud liquid water, and aerosols. SABER and AURA observations have been assimilated into WACCM.
- Specify, via namelist entries, the CAM (initial file) variables which will be directly affected by the observations, that is, the state vector. This allows users to change the model state without recompiling (but other restrictions remain).
- Generate analyses on the CAM grid which have only CAM model error in them, rather than another model's.
- Generate such analyses with as few as 20 ensemble members.

In addition to the standard DART package there are ensembles of initial condition files at the large file website http://www.image.ucar.edu/pub/DART/CAM/ that are helpful for interfacing CAM with DART. In the current (2015) mode, CESM+DART can easily be started from a single model state, which is perturbed to create an ensemble of the desired size. A spin-up period is then required to allow the ensemble members to diverge.

Sample sets of observations, which can be used with CESM+DART assimilations, can be found at http://www.image.ucar.edu/pub/DART/Obs_sets/ of which the NCEP BUFR observations are the most widely used.

Experience on a variety of machines has shown that it is a very good idea to make sure your run-time environment has the following:

```
limit stacksize unlimited
limit datasize unlimited
```

This page contains the documentation for the DART interface module for the CAM and WACCM models, using the dynamical cores listed above. This implementation uses the CAM initial files (**not** restart files) for transferring the model state to/from the filter. This may change in future versions, but probably only for CAM-SE. The reasons for this include:

1. The contents of the restart files vary depending on both the model release version and the physics packages selected.
2. There is no metadata describing the variables in the restart files. Some information can be tracked down in the `atm.log` file, but not all of it.

3. The restart files (for non-chemistry model versions) are much larger than the initial files (and we need to deal with an ensemble of them).
4. The temperature on the restart files is virtual equivalent potential temperature, which requires (at least) surface pressure, specific humidity, and sensible temperature to calculate.
5. CAM does not call the initialization routines when restart files are used, so fields which are not modified by DART may be inconsistent with fields which are.
6. If DART modifies the contents of the `.r.` restart file, it might also need to modify the contents of the `.rs.` restart file, which has similar characteristics (1-3 above) to the `.r.` file.

The DART interfaces to CAM and many of the other CESM components have been integrated with the CESM set-up and run scripts.

6.236.2 Setup Scripts

Unlike previous versions of DART-CAM, CESM runs using its normal scripts, then stops and calls a DART script, which runs a single assimilation step, then returns to the CESM run script to continue the model advances. See the CESM interface documentation in `$DARTROOT/models/CESM` for more information on running DART with CESM. Due to the complexity of the CESM software environment, the versions of CESM which can be used for assimilation are more restricted than previously. Each supported CESM version has similar, but unique, sets of set-up scripts and CESM SourceMods. Those generally do not affect the `cam-fv/model_mod.f90` interface. Current (April, 2015) set-up scripts are:

- `CESM1_2_1_setup_pmo`: sets up a perfect_model_mod experiment, which creates synthetic observations from a free model run, based on the user's somewhat restricted choice of model, dates, etc. The restrictions are made in order to streamline the script, which will shorten the learning curve for new users.
- `CESM1_2_1_setup_pmo_advanced`: same as `CESM1_2_1_setup_pmo`, but can handle more advanced set-ups: recent dates (non-default forcing files), refined-grid CAM-SE, etc.
- `CESM1_2_1_setup_hybrid`: streamlined script (see `CESM1_2_1_setup_pmo`) which sets up an ensemble assimilation using CESM's multi-instance capability.
- `CESM1_2_1_setup_advanced`: like `CESM1_2_1_setup_pmo_advanced`, but for setting up an assimilation.

The DART state vector should include all prognostic variables in the CAM initial files which cannot be calculated directly from other prognostic variables. In practice the state vector sometimes contains derived quantities to enable DART to compute forward operators (expected observation values) efficiently. The derived quantities are often overwritten when the model runs the next timestep, so the work DART does to update them is wasted work.

Expected observation values on pressure, scale height, height or model levels can be requested from `model_interpolate`. Surface observations can not yet be interpolated, due to the difference between the model surface and the earth's surface where the observations are made. Model_interpolate can be queried for any (non-surface) variable in the state vector (which are variables native to CAM) plus pressure on height levels. The default state vector is PS, T, U, V, Q, CLDLIQ, CLDICE and any tracers or chemicals needed for a given study. Variables which are not in the initial file can be added (see the `./doc` directory but minor modifications to `model_mod.f90` and CAM may be necessary.

The 19 public interfaces in `model_mod` are standardized for all DART compliant models. These interfaces allow DART to get the model state and metadata describing this state, find state variables that are close to a given location, and do spatial interpolation for a variety of variables required by observational operators.

6.236.3 Namelist

The `&model_nml` namelist is read from the `input.nml` file. Namelists start with an ampersand `&` and terminate with a slash `/`. Character strings that contain a `/` must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&model_nml
   cam_template_filename               = 'caminput.nc'
   cam_phis_filename                   = 'cam_phis.nc'
   vertical_localization_coord         = 'PRESSURE'
   use_log_vertical_scale              = .false.
   no_normalization_of_scale_heights   = .true.
   no_obs_assim_above_level            = -1,
   model_damping_ends_at_level         = -1,
   state_variables                     = ''
   assimilation_period_days            = 0
   assimilation_period_seconds         = 21600
   suppress_grid_info_in_output        = .false.
   custom_routine_to_generate_ensemble = .true.
   fields_to_perturb                   = ''
   perturbation_amplitude              = 0.0_r8
   using_chemistry                     = .false.
   use_variable_mean_mass              = .false.
   debug_level                         = 0
/
```

The names of the fields to put into the state vector must match the CAM initial NetCDF file variable names.

Item	Type	Description
cam_template_file	character(len=128)	CAM initial file used to provide configuration information, such as the grid resolution, number of vertical levels, whether fields are staggered or not, etc.
cam_phis	character(len=128)	CAM topography file. Reads the "PHIS" NetCDF variable from this file. Typically this is a CAM History file because this field is not normally found in a CAM initial file.
vertical_localization_coord	character(len=128)	The vertical coordinate to which all vertical locations are converted in model_mod. Valid options are "pressure", "height", "scaleheight" or "level".
no_normalization_of_scale_heights	logical	If true the scale height is computed as the log of the pressure at the given location. If false the scale height is computed as a ratio of the log of the surface pressure and the log of the pressure aloft. In limited areas of high topography the ratio version might be advantageous, and in previous versions of filter this was the default. For global CAM the recommendation is to set this to .true. so the scale height is simply the log of the pressure at any location.
no_obs_assim_above_level	integer	Because the top of the model is highly damped it is recommended to NOT assimilate observations in the top model levels. The units here are CAM model level numbers. Set it to equal or below the lowest model level (the highest number) where damping is applied in the model.
model_damping_ends_at_level	integer	Set this to the lowest model level (the highest number) where model damping is applied. Observations below the 'no_obs_assim_above_level' cutoff but close enough to the model top to have an impact during the assimilation will have their impacts decreased smoothly to 0 at this given model level. The assimilation should make no changes to the model state above the given level.
state_variables	character(len=64), dimension(100)	Character string table that includes: Names of fields (NetCDF variable names) to be read into the state vector, the corresponding DART Quantity for that variable, if a bounded quantity the minimum and maximum valid values, and finally the string 'UPDATE' to indicate the updated values should be written back to the output file. 'NOUPDATE' will skip writing this field at the end of the assimilation.
assimilation_period_days	integer	Sets the assimilation window width, and should match the model advance time when cycling. The scripts distributed with DART always set this to 0 days, 21600 seconds (6 hours).
assimilation_period_seconds	integer	Sets the assimilation window width, and should match the model advance time when cycling. The scripts distributed with DART always set this to 0 days, 21600 seconds (6 hours).
suppress_grid_info_in_output	logical	Filter can update fields in existing files or create diagnostic/output files from scratch. By default files created from scratch include a full set of CAM grid information to make the file fully self-contained and plottable. However, to save disk space the grid variables can be suppressed in files created by filter by setting this to true.
custom_routine_to_generate_ensemble	logical	The default perturbation routine in filter adds gaussian noise equally to all fields in the state vector. It is recommended to set this option to true so code in the model_mod is called instead. This allows only a limited number of fields to be perturbed. For example, only perturbing the temperature field T with a small amount of noise and then running the model forward for a few days is often a recommended way to generate an ensemble from a single state.
fields_to_perturb	character(len=32), dimension(100)	If perturbing a single state to generate an ensemble, set 'custom_routine_to_generate_ensemble = .true.' and list list the field(s) to be perturbed here.
perturbation_amplitude	real(r8), dimension(100)	For each field name in the 'fields_to_perturb' list give the standard deviation for the gaussian noise to add to each field being perturbed.
pert_base_vals	real(r8), dimension(100)	If pert_sd is positive, this the list of values to which the field(s) listed in pert_names will be reset if filter is told to create an ensemble from a single state vector. Otherwise, it's is the list of values to use for each ensemble member when perturbing the single field named in pert_names. Unused unless pert_names is set and pert_base_vals is not the DART missing value.
using_chemistry	logical	If using CAM-CHEM, set this to .true.
us-	logical	If using any variant of WACCM with a very high model top, set this to .true.

Item	Type	Description
cam_template_file	character(len=128)	CAM initial file used to provide configuration information, such as the grid resolution, number of vertical levels, whether fields are staggered or not, etc.
cam_phis	character(len=128)	CAM topography file. Reads the “PHIS” NetCDF variable from this file. Typically this is a CAM History file because this field is not normally found in a CAM initial file.
vertical_localization_coord	character(len=128)	The vertical coordinate to which all vertical locations are converted in model_mod. Valid options are “pressure”, “height”, “scaleheight” or “level”.
no_normalization_of_scale_heights	logical	If true the scale height is computed as the log of the pressure at the given location. If false the scale height is computed as a ratio of the log of the surface pressure and the log of the pressure aloft. In limited areas of high topography the ratio version might be advantageous, and in previous versions of filter this was the default. For global CAM the recommendation is to set this to .true. so the scale height is simply the log of the pressure at any location.
no_obs_assim_above_level	integer	Because the top of the model is highly damped it is recommended to NOT assimilate observations in the top model levels. The units here are CAM model level numbers. Set it to equal or below the lowest model level (the highest number) where damping is applied in the model.
model_damping_ends_at_level	integer	Set this to the lowest model level (the highest number) where model damping is applied. Observations below the ‘no_obs_assim_above_level’ cutoff but close enough to the model top to have an impact during the assimilation will have their impacts decreased smoothly to 0 at this given model level. The assimilation should make no changes to the model state above the given level.
state_variables	character(len=64), dimension(100)	Character string table that includes: Names of fields (NetCDF variable names) to be read into the state vector, the corresponding DART Quantity for that variable, if a bounded quantity the minimum and maximum valid values, and finally the string ‘UPDATE’ to indicate the updated values should be written back to the output file. ‘NOUPDATE’ will skip writing this field at the end of the assimilation.
assimilation_period_days	integer	Sets the assimilation window width, and should match the model advance time when cycling. The scripts distributed with DART always set this to 0 days, 21600 seconds (6 hours).
assimilation_period_seconds	integer	Sets the assimilation window width, and should match the model advance time when cycling. The scripts distributed with DART always set this to 0 days, 21600 seconds (6 hours).
suppress_grid_info_in_output	logical	Filter can update fields in existing files or create diagnostic/output files from scratch. By default files created from scratch include a full set of CAM grid information to make the file fully self-contained and plottable. However, to save disk space the grid variables can be suppressed in files created by filter by setting this to true.
custom_routine_to_generate_ensemble	logical	The default perturbation routine in filter adds gaussian noise equally to all fields in the state vector. It is recommended to set this option to true so code in the model_mod is called instead. This allows only a limited number of fields to be perturbed. For example, only perturbing the temperature field T with a small amount of noise and then running the model forward for a few days is often a recommended way to generate an ensemble from a single state.
fields_to_perturb	character(len=32), dimension(100)	If perturbing a single state to generate an ensemble, set ‘custom_routine_to_generate_ensemble = .true.’ and list list the field(s) to be perturbed here.
perturbation_amplitude	real(r8), dimension(100)	For each field name in the ‘fields_to_perturb’ list give the standard deviation for the gaussian noise to add to each field being perturbed.
pert_base_vals	real(r8), dimension(100)	If pert_sd is positive, this the list of values to which the field(s) listed in pert_names will be reset if filter is told to create an ensemble from a single state vector. Otherwise, it’s is the list of values to use for each ensemble member when perturbing the single field named in pert_names. Unused unless pert_names is set and pert_base_vals is not the DART missing value.
using_chemistry	logical	If using CAM-CHEM, set this to .true.
us-	logical	If using any variant of WACCM with a very high model top, set this to .true.

6.236.4 Setup Variations

The variants of CAM require slight changes to the setup scripts (in $DARTROOT/models/cam-fv/shell_scripts) and in the namelists (in $DARTROOT/models/cam-fv/work/input.nml). From the DART side, assimilations can be started from a pre-existing ensemble, or an ensemble can be created from a single initial file before the first assimilation. In addition, there are setup differences between 'perfect model' runs, which are used to generate synthetic observations, and assimilation runs. Those differences are extensive enough that they've been coded into separate *Setup Scripts*.

Since the CESM compset and resolution, and the initial ensemble source are essentially independent of each other, changes for each of those may need to be combined to perform the desired setup.

Perturbed Ensemble

The default values in work/input.nml and shell_scripts/CESM1_2_1_setup_pmo and shell_scripts/CESM1_2_1_setup_hybrid are set up for a CAM-FV, single assimilation cycle using the default values as found in model_mod.f90 and starting from a single model state, which must be perturbed into an ensemble. The following are suggestions for setting it up for other assimilations. Namelist variables listed here might be in any namelist within input.nml.

CAM-FV

If built with the FV dy-core, the number of model top levels with extra diffusion in CAM is controlled by div24del2flag. The recommended minium values of highest_state_pressure_Pa come from that variable, and cutoff*vert_normalization_X:

```
2    ("div2") -> 2 levels  -> highest_state_pressure_Pa =  9400. Pa
4,24 ("del2") -> 3 levels  -> highest_state_pressure_Pa = 10500. Pa
```

and:

```
vert_coord           = 'pressure'
state_num_1d         = 0,
state_num_2d         = 1,
state_num_3d         = 6,
state_names_1d       = ''
state_names_2d       = 'PS'
state_names_3d       = 'T', 'US', 'VS', 'Q', 'CLDLIQ', 'CLDICE'
which_vert_1d        = 0,
which_vert_2d        = -1,
which_vert_3d        = 6*1,
highest_state_pressure_Pa = 9400. or 10500.
```

CAM-SE

There's an existing ensemble, so see the *Continuing after the first cycle* section to start from it instead of a single state. To set up a "1-degree" CAM-SE assimilation CESM1_2_1_setup_hybrid:

```
setenv resolution  ne30_g16
setenv refcase     SE30_Og16
setenv refyear     2005
setenv refmon      08
setenv refday      01
```

input.nml:

```
approximate_distance = .FALSE.
vert_coord          = 'pressure'
state_num_1d        = 1,
state_num_2d        = 6,
state_num_3d        = 0,
state_names_1d      = 'PS'
state_names_2d      = 'T','U','V','Q','CLDLIQ','CLDICE'
state_names_3d      = ''
which_vert_1d       = -1,
which_vert_2d       = 6*1,
which_vert_3d       = 0,
highest_obs_pressure_Pa   = 1000.,
highest_state_pressure_Pa = 10500.,
```

Variable resolution CAM-SE

To set up a variable resolution CAM-SE assimilation (as of April 2015) there are many changes to both the CESM code tree and the DART setup scripts. This is for very advanced users, so please contact dart @ ucar dot edu or raeder @ ucar dot edu for scripts and guidance.

WACCM

WACCM[#][-X] has a much higher top than the CAM versions, which requires the use of scale height as the vertical coordinate, instead of pressure, during assimilation. One impact of the high top is that the number of top model levels with extra diffusion in the FV version is different than in the low-topped CAM-FV, so the div24del2flag options lead to the following minimum values for highest_state_pressure_Pa:

```
2    ("div2") -> 3 levels  -> highest_state_pressure_Pa = 0.01 Pa
4,24 ("del2") -> 4 levels  -> highest_state_pressure_Pa = 0.02 Pa
```

The best choices of vert_normalization_scale_height, cutoff, and highest_state_pressure_Pa are still being investigated (April, 2015), and may depend on the observation distribution being assimilated.

WACCM is also typically run with coarser horizontal resolution. There's an existing 2-degree ensemble, so see the *Continuing after the first cycle* section to start from it, instead of a single state. If you use this, ignore any existing inflation restart file and tell DART to make its own in the first cycle in input.nml:

```
inf_initial_from_restart    = .false.,                 .false.,
inf_sd_initial_from_restart = .false.,                 .false.,
```

In any case, make the following changes (or similar) to convert from a CAM setup to a WACCM setup. CESM1_2_1_setup_hybrid:

```
setenv compset     F_2000_WACCM
setenv resolution  f19_f19
setenv refcase     FV1.9x2.5_WACCM4
setenv refyear     2008
setenv refmon      12
setenv refday      20
```

and the settings within input.nml:

```
vert_normalization_scale_height = 2.5
vert_coord                = 'log_invP'
highest_obs_pressure_Pa   = .001,
highest_state_pressure_Pa = .01,
```

If built with the SE dy-core (warning; experimental), then 4 levels will have extra diffusion, and also see the *CAM-SE* section.

If there are problems with instability in the WACCM foreasts, try changing some of the following parameters in either the user_nl_cam section of the setup script or input.nml.

- The default div24del2flag in WACCM is 4. Change it in the setup script to

  ```
  echo " div24del2flag         = 2 "                       >> ${fname}
  ```

  which will use the `cd_core.F90` in SourceMods, which has doubled diffusion in the top layers compared to CAM.

- Use a smaller dtime (1800 s is the default for 2-degree) in the setup script. This can also be changed in the ensemble of `user_nl_cam_####` in the `$CASEROOT` directory.

  ```
  echo " dtime         = 600 "                             >> ${fname}
  ```

- Increase highest_state_pressure_Pa in input.nml:

  ```
  div24del2flag = 2    ("div2") -> highest_state_pressure_Pa = 0.1 Pa
  div24del2flag = 4,24 ("del2") -> highest_state_pressure_Pa = 0.2 Pa
  ```

- Use a larger nsplit and/or nspltvrm in the setup script:

  ```
  echo " nsplit         = 16 "                             >> ${fname}
  echo " nspltvrm       =  4 "                             >> ${fname}
  ```

- Reduce `inf_damping` from the default value of `0.9` in `input.nml`:

  ```
  inf_damping           = 0.5,                   0,
  ```

6.236.5 Notes for Continuing an Integration

Continuing after the first cycle

After the first forecast+assimilation cycle, using an ensemble created from a single file, it is necessary to change to the 'continuing' mode, where CAM will not perform all of its startup procedures and DART will use the most recent ensemble. This example applies to an assimiation using prior inflation (`inf_...= .true.`). If posterior inflation were needed, then the 2nd column of `infl_...` would be set to `.true...` Here is an example snippet from `input.nml`:

```
start_from_restart      = .true.,
restart_in_file_name    = "filter_ics",
single_restart_file_in  = .false.,

inf_initial_from_restart    = .true.,                 .false.,
inf_sd_initial_from_restart = .true.,                 .false.,
```

Combining multiple cycles into one job

CESM1_2_1_setup_hybrid and CESM1_2_1_setup_pmo are set up in the default cycling mode, where each submitted job performs one model advance and one assimilation, then resubmits the next cycle as a new job. For long series of cycles, this can result in a lot of time waiting in the queue for short jobs to run. This can be prevented by using the 'cycles' scripts generated by CESM1_2_1_setup_advanced (instead of CESM1_2_1_setup_hybrid). This mode is described in $DARTROOT/models/cam-fv/doc/README_cam-fv.

6.236.6 Discussion

Many CAM initial file variables are already handled in the model_mod. Here is a list of others, which may be used in the future. Each would need to have a DART *KIND* associated with it in model_mod.

```
Atmos
   CLOUD:       "Cloud fraction" ;
   QCWAT:       "q associated with cloud water" ;
   TCWAT:       "T associated with cloud water" ;
   CWAT:        "Total Grid box averaged Condensate Amount (liquid + ice)" ;
   also? LCWAT

pbl
   PBLH:        "PBL height" ;
   QPERT:       "Perturbation specific humidity (eddies in PBL)" ;
   TPERT:       "Perturbation temperature (eddies in PBL)" ;

Surface
   LANDFRAC:    "Fraction of sfc area covered by land" ;
   LANDM:       "Land ocean transition mask: ocean (0), continent (1), transition (0-
↪1)" ;
      also LANDM_COSLAT
   ICEFRAC:     "Fraction of sfc area covered by sea-ice" ;
   SGH:         "Standard deviation of orography" ;
   Z0FAC:       "factor relating z0 to sdv of orography" ;
   TS:          "Surface temperature (radiative)" ;
   TSOCN:       "Ocean tempertare" ;
   TSICE:       "Ice temperature" ;
   TSICERAD:    "Radiatively equivalent ice temperature" ;

Land/under surface
   SICTHK:      "Sea ice thickness" ;
   SNOWHICE:    "Water equivalent snow depth" ;
   TS1:         "subsoil temperature" ;
   TS2:         "subsoil temperature" ;
   TS3:         "subsoil temperature" ;
   TS4:         "subsoil temperature" ;

Other fields are not included because they look more CLM oriented.

Other fields which users may add to the CAM initial files are not listed here.
```

6.236.7 Files

- `model_nml` in `input.nml`
- `cam_phis.nc` (CAM surface height file, often CAM's .h0. file in the CESM run environment)
- `caminput.nc` (CAM initial file)
- `clminput.nc` (CLM restart file)
- `iceinput.nc` (CICE restart file) by model_mod at the start of each assimilation)
- netCDF output state diagnostics files

6.236.8 Nitty gritty: Efficiency possibilities

- index_from_grid (and others?) could be more efficient by calculating and globally storing the beginning index of each cfld and/or the size of each cfld. Get_state_meta_data too. See `clm/model_mod.f90`.
- Global storage of height fields? but need them on staggered grids (only sometimes) Probably not; machines going to smaller memory and more recalculation.
- ! Some compilers can't handle passing a section of an array to a subroutine/function; I do this in `nc_write_model_vars(?)` and/or `write_cam_init(?)`; replace with an exactly sized array?
- Is the testing of resolution in read_cam_coord overkill in the line that checks the size of `(resol_n - resol_1)*resol`?
- Replace some do loops with forall (constructs)
- Subroutine `write_cam_times(model_time, adv_time)` is not needed in CESM+DART framework? Keep anyway?
- Remove the code that accommodates old CAM coordinate order (`lon,lev,lat`).
- Cubed sphere: Convert lon,lat refs into dim1,dim2 in more subroutines. get_val_heights is called with (`column_ind,1`) by CAM-SE code, and (`lon_ind, lat_ind`) otherwise).
- `cam_to_dart_kinds` and `dart_to_cam_types` are dimensioned 300, regardless of the number of fields in the state vector and/or *KIND*s .
- Describe:
  - The coordinate orders and translations; CAM initial file, `model_mod`, and `DART_Diag.nc`.
  - Motivations
    - There need to be 2 sets of arrays for dimensions and dimids;
      - one describing the caminput file (`f_...`)
      - and one for the state (`s_...`) (storage in this module).
      - Call them `f_dim_Nd`, `f_dimid_Nd`
      - `s_dim_Nd`, `s_dimid_Nd`
- Change (private only) subroutine argument lists; structures first, regardless of in/out then output, and input variables.
- Change declarations to have dummy argument integers used as dimensions first
- Implement a `grid_2d_type`? Convert phis to a `grid_2d_type`? ps, and staggered ps fields could also be this type.

- Deallocate grid_1d_arrays using end_1d_grid_instance in end_model. end_model is called by subroutines pert_model_state, nc_write_model_vars; any problem?
- ISSUE; In P[oste]rior_Diag.nc ensemble members are written out *between* the field mean/spread pair and the inflation mean/sd pair. Would it make more sense to put members after both pairs? Easy to do?
- ISSUE?; model_interpolate assumes that obs with a vertical location have 2 horizontal locations too. The state vector may have fields for which this isn't true, but no obs we've seen so far violate this assumption. It would have to be a synthetic/perfect_model obs, like some sort of average or parameter value.
- ISSUE; In convert_vert, if a 2D field has dimensions (lev, lat) then how is p_surf defined? Code would be needed to set the missing dimension to 1, or make different calls to coord_ind, etc.
- ISSUE; The QTY_ list from obs_def_mod must be updated when new fields are added to state vector. This could be done by the preprocessor when it inserts the code bits corresponding to the lists of observation types, but it currently (10/06) does not. Document accordingly.
- ISSUE: The CCM code (and Hui's packaging) for geopotentials and heights use different values of the physical constants than DART's. In one case Shea changed g from 9.81 to 9.80616, to get agreement with CCM(?...), so it may be important. Also, matching with Hui's tests may require using his values; change to DART after verifying?
- ISSUE: It's possible to figure out the model_version from the NetCDF file itself, rather than have that be user-provided (sometimes incorrect and hard to debug) meta-data. model_version is also misnamed; it's really the caminput.nc model version. The actual model might be a different version(?). The problem with removing it from the namelist is that the scripts need it too, so some rewriting there would be needed.
- ISSUE: max_neighbors is set to 6, but could be set to 4 for non-refined grids. Is there a good mechanism for this? Is it worth the file space savings?
- ISSUE: x_planar and y_planar could be reduced in rank, if no longer needed for testing and debugging.
- "Pobs" marks changes for providing expected obs of P break from past philosophy; P is not a native CAM variable (but is already calced here)
- NOVERT marks modifications for fields with no vertical location, i.e. GWD parameters.

6.236.9 References and Acknowledgements

- CAM homepage

Ave Arellano did the first work with CAM-Chem, assimilating MOPPITT CO observations into CAM-Chem. Jerome Barre and Benjamin Gaubert took up the development work from Ave, and prompted several additions to DART, as well as model_mod.f90.

Nick Pedatella developed the first vert_coord = 'log_invP' capability to enable assimilation using WACCM and scale height vertical locations.

6.237 PROGRAM dart_to_cam

6.237.1 Overview

dart_to_cam is the program that reads a DART restart or model advance file (e.g. perfect_ics, filter_ics, assim_model_state_id ...). and overwrites the part of the CAM data in a single CAM restart file (usually caminput.nc) which is in the DART state vector. If you have multiple input files, you will need to rename the output files as you create them.

The list of variables extracted from the DART state vector and exported to the CAM netCDF file is controlled by the set of `input.nml` `&model_nml:state_names_*` variables.

If the input is a model advance file, containing 2 timestamps (the current model time and a future time the model should run until), this program also creates a separate file named `times` that contains three lines: the advance-to time, the current model time, and the number of hours to advance. These will need to be extracted and inserted in a CAM namelist to indicate to CAM how long to run.

This program also updates the `date` and `datesec` variables in the CAM netcdf file. Generally these are identical times since the assimilation doesn't change the time of the data, but in case the original file had a different time that was overwritten in the state vector, it will update the time for consistency.

Conditions required for successful execution of `dart_to_cam`:

- a valid `input.nml` namelist file for DART
- a CAM 'phis' netCDF file [default: `cam_phis.nc`]
- a DART restart file [default: `dart_ics`] (read)
- a CAM restart file [default: `caminput.nc`] (read and written)

Since this program is called repeatedly for every ensemble member, we have found it convenient to link the DART input and CAM restart files to the default filenames `dart_ics` and `caminput.nc`). The output files may be moved or relinked as necessary.

6.237.2 Namelist

This namelist is read from the file `input.nml`. Namelists start with an ampersand '&' and terminate with a slash '/'. Character strings that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
&dart_to_cam_nml
   dart_to_cam_input_file  = 'dart_ics',
   dart_to_cam_output_file = 'caminput.nc',
   advance_time_present    = .true.,
   /
```

Item	Type	Description
dart_to_cam_input_file	character(len=128)	The name of the DART restart file containing the CAM state.
dart_to_cam_output_file	character(len=128)	The name of the CAM restart netcdf file.
advance_time_present	logical	Set to .false. for DART initial condition and restart files. Use the .true. setting for the files written by filter during a model advance.

6.237.3 Modules used

```
assim_model_mod.f90
types_mod.f90
threed_sphere/location_mod.f90
model_mod.f90
null_mpi_utilities_mod.f90
obs_kind_mod.f90
random_seq_mod.f90
time_manager_mod.f90
utilities_mod.f90
```

6.237.4 Files read

- DART namelist file; `input.nml`
- DART initial conditions/restart file; e.g. `dart_ics` (read)
- CAM restart file; `caminput.nc` (read and written)
- CAM "phis" file specified in `&model_nml::cam_phis` (normally `cam_phis.nc`)

6.237.5 Files written

- CAM restart file; `caminput.nc` (read and written)

6.237.6 References

none

6.238 PROGRAM `trans_pv_sv`

`trans_pv_sv` is responsible for converting the ocean model 'snapshot' files to a DART 'initial conditions' file. In order to do that, the valid time for the snapshot files must be calculated from several pieces of information: the filename contains a timestep index, the `data&PARM03` namelist contains information about the amount of time per timestep, and the `data.cal&CAL_NML` namelist contains the start time. Additionally, the grid characteristics must be read from `data&PARM04`. Consequently, the files `data`, and `data.cal` as well as the general `input.nml` are needed in addition to the snapshot files.

This program has a number of options that are driven from namelists and **one** piece of input read from STDIN: the integer representing the timestep index of the snapshot file set.

6.238.1 Usage

The output filename is hardwired to that expected by filter. This example creates an output file named assim_model_state_ud from the following files in the local directory:

S.0000000096.data
T.0000000096.data
U.0000000096.data
V.0000000096.data
Eta.0000000096.data

./trans_pv_sv < 96

6.238.2 Modules used

```
types_mod
utilities_mod
model_mod
assim_model_mod
time_manager_mod
```

6.238.3 Namelist

This program has no namelist of its own, but some of the underlying modules require namelists. To avoid duplication and, possibly, some inconsistency in the documentation, only a list of the required namelists is provided here, with a hyperlink to the full documentation for each namelist.

Namelist	Primary Purpose
utilities_nml	set the termination level and file name for the run-time log
assim_model_mod_nml	write DART restart files in binary or ASCII
model_nml	write netCDF files with prognostic variables
CAL_NML	determine start time of the ocean model
PARM03	the amount of time per model timestep for deciphering snapshot filenames
PARM04	ocean model grid parameters

6.238.4 Files

- input namelist files: data, data.cal, input.nml
- input snapshot files: [S,T,U,V,Eta].nnnnnnnnnn.[data[,.meta]]
- output initial conditions file: assim_model_state_ud

6.238.5 References

- none

6.238.6 Private components

N/A

6.239 PROGRAM `create_ocean_obs`

`create_ocean_obs` is responsible for converting an interim ASCII file of ocean observations into a DART observation sequence file. The interim ASCII file is a simple 'whitespace separated' table where each row is an observation and each column is specific information about the observation.

column number	quantity	description
1	longitude (in degrees)	longitude of the observation
2	latitude (in degrees)	latitude of the observation
3	depth (in meters)	depth of the observation
4	observation value	such as it is . . .
5	vertical coordinate flag	see location_mod:location_type for a full explanation. The short explanation is that *surface == -1*, and *depth == 3* There is a pathological difference between a surface observation and an observation with a depth of zero.
6	observation variance	good luck here . . .
7	Quality Control flag	integer value passed through to DART. There is a namelist parameter for `filter` to ignore any observation with a QC value <= input_qc_thre shold
8	obs_kind_name	a character string that must match a string in :doc:*../../observations/forward_operators/obs_def_MITgcm_ocean_model_mod*
9	startDate_1	the year-month-date of the observation (YYYYMMDD format)
10	startDate_2	the hour-minute-second of the observation (HHMMSS format)

For example:

```
273.7500 21.3500 -2.5018 28.0441  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 21.4500 -2.5018 28.1524  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
```

(continues on next page)

(continued from previous page)

```
273.7500 21.5500 -2.5018 28.0808  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 21.6500 -2.5018 28.0143  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 21.7500 -2.5018 28.0242  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 21.8500 -2.5018 28.0160  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 21.9500 -2.5018 28.0077  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 22.0500 -2.5018 28.3399  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 22.1500 -2.5018 27.8852  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
273.7500 22.2500 -2.5018 27.8145  3 0.0400  1  GLIDER_TEMPERATURE 19960101  10000
...
```

It is always possible to combine observation sequence files with the program *program obs_sequence_tool*, so it was simply convenient to generate a separate file for each observation platform and type ('GLIDER' and 'TEMPERATURE'), however it is by no means required.

6.239.1 Modules used

Some of these modules use modules . . . **those** modules and namelists are not discussed here. probably should be . . .

```
types_mod
utilities_mod
dart_MITocean_mod
obs_sequence_mod
```

6.239.2 Namelist

This program has a namelist of its own, and some of the underlying modules require namelists. To avoid duplication and, possibly, some inconsistency in the documentation; only a list of the required namelists is provided - with a hyperlink to the full documentation for each namelist.

Namelist	Primary Purpose
utilities_nml	set the termination level and file name for the run-time log
obs_sequence_nml	write binary or ASCII observation sequence files

We adhere to the F90 standard of starting a namelist with an ampersand '&' and terminating with a slash '/'. Consider yourself forewarned that filenames that contain a '/' must be enclosed in quotes to prevent them from prematurely terminating the namelist.

```
namelist /create_ocean_obs_nml/  year, month, day, &
         tot_days, max_num, fname, output_name, lon1, lon2, lat1, lat2
```

This namelist is read in a file called `input.nml`

Contents	Type	Description
year	integer *[default: 1996]*	The first year of interest.
month	integer *[default: 1]*	The first month of interest.
day	integer *[default: 1]*	The first day of interest.
tot_days	integer *[default: 31]*	Stop processing after this many days.
max_num	integer *[default: 800000]*	The maximum number of observations to read/write.
fname	character(len=129) *[default: 'raw_ocean_obs.txt']*	The name of the interim ASCII file of observations.
output_name	character(len=129) *[default: 'raw_ocean_obs_seq.out']*	The output file name.
lon1	real *[default: 0.0]*	The leftmost longitude of interest.
lon2	real *[default: 360.0]*	The rightmost longitude of interest.
lat1	real *[default: -90.0]*	The most southern latitude of interest.
lat2	real *[default: 90.0]*	The most northern latitude of interest.

6.239.3 Files

- input namelist file: `input.nml`
- input data file: as listed by `input.nml&create_ocean_obs_nml:fname`
- output data file: as listed by `input.nml&create_ocean_obs_nml:output_name`

6.239.4 References

- none

6.240 PROGRAM `trans_sv_pv`

`trans_sv_pv` is responsible for converting a DART 'initial conditions' file to a set of model 'snapshot' files and appropriate namelist files: `data.cal` and `data`. This is easier than the reverse process because the DART initial conditions file have a header that contains the valid time for the accompanying state. This same header also has the 'advance-to' time. `trans_sv_pv` uses this information to write out appropriate `&CAL_NML` and `&PARM03` namelists in `data.cal.DART` and `data.DART`, respectively. The rest of the information in `data` is preserved, so it is possible to simply replace `data` with the new `data.DART`.

The input filename is hardwired to that expected by `filter` and the output filenames are able to be renamed into those defined by the `data&PARM05` namelist specifying the filenames to use to cold-start the ocean model. The output filename is comprised of 4 parts: the variable name, the startDate_1 component (YYYYMMDD), the startDate_2 component (HHMMSS), and the extension (.data for the data and .meta for the metadata). The startDate_1 and startDate_2 pieces are identical in format to that used by identically named variables in the `data.cal&CAL_NML` namelist.

6.240.1 Usage

There must be several input files in the current working directory; most of these are required by the model_mod interface. The input filename is hardwired to assim_model_state_ic. Assuming the time tag in the input file is set to 06Z 23 July 1996, this example creates output files named
S.19960723.060000.[data,meta]
T.19960723.060000.[data,meta]
U.19960723.060000.[data,meta]
V.19960723.060000.[data,meta]
Eta.19960723.060000.[data,meta]
data.cal.DART, and
data.DART

mv some_DART_ics_input_file assim_model_state_ic ./trans_sv_pv cp data.cal.DART data.cal cp data.DART data

6.240.2 Modules used

```
types_mod
utilities_mod
model_mod
assim_model_mod
time_manager_mod
```

6.240.3 Namelist

This program has no namelist of its own, but some of the underlying modules require namelists to be read, even if the values are not used. To avoid duplication and, possibly, some inconsistency in the documentation; only a list of the required namelists is provided - with a hyperlink to the full documentation for each namelist.

Namelist	Primary Purpose
utilities_nml	set the termination level and file name for the run-time log
CAL_NML	must be read, values are not used. The data.cal.DART file has an updated namelist to be used for the model advance.
PARM03	must be read, values are not used, The data.DART is an 'identical' version of data with the exception of the PARM03 namelist. The parameters endTime, dumpFreq, and taveFreq reflect the amount of time needed to advance the model. The parameter startTime is set to 0.0, which is required to force the model to read the startup files specified by PARM05
PARM04	ocean model grid parameters, read - never changed.

6.240.4 Files

- input namelist files: `data, data.cal, input.nml`
- output namelist files: `data.cal.DART, data.DART`
- input data file: `assim_model_state_ic`
- output data files: `[S,T,U,V,Eta].YYYYMMDD.HHMMSS.[data,meta]`

6.240.5 References

- none

6.241 mkmf

6.241.1 Introduction

`mkmf` is a tool written in perl version 5 that constructs a makefile from distributed source. `mkmf` typically produces a makefile that can compile a single executable program. But it is extensible to create a makefile for any purpose at all.

Features of mkmf

- It understands dependencies in f90 (`modules` and `use`), the fortran `include` statement, and the cpp `#include` statement in any type of source.
- There are no restrictions on filenames, module names, etc.
- It supports the concept of overlays (where source is maintained in layers of directories with a defined precedence).
- It can keep track of changes to `cpp` flags, and knows when to recompile affected source (i.e, files containing `#ifdefs` that have been changed since the last invocation).
- It will run on any unix platform that has perl version 5 installed.
- It is free, and released under GPL. GFDL users can copy (or, better still, directly invoke) the file `/net/vb/public/bin/mkmf`.

It can be downloaded via GitHub. `mkmf` is pronounced *make-make-file* or *make-m-f* or even *McMuff* (Paul Kushner's suggestion).

6.241.2 Syntax

The calling syntax is:

`mkmf [-a abspath] [-c cppdefs] [-d] [-f] [-m makefile] [-p program] [-t template] [-v] [-w] [-x] [args]`

1. `-a abspath` attaches the `abspath` at the *front* of all *relative* paths to sourcefiles.
2. `cppdefs` is a list of `cpp #defines` to be passed to the source files: affected object files will be selectively removed if there has been a change in this state.
3. `-d` is a debug flag to `mkmf` (much more verbose than `-v`, but probably of use only if you are modifying `mkmf` itself).

4. `-f` is a formatting flag to restrict lines in the makefile to 256 characters. This was introduced in response to a customer who wanted to edit his makefiles using `vi`). Lines longer than that will use continuation lines as needed.
5. `makefile` is the name of the makefile written (default `Makefile`).
6. `template` is a file containing a list of make macros or commands written to the beginning of the makefile.
7. `program` is the name of the final target (default `a.out`)
8. `-v` is a verbosity flag to `mkmf`
9. `-w` generates compile rules which use the `wrapper' commands MPIFC and MPILD instead of FC and LD. These can then be defined as the mpif90 compile scripts to ease changing between an MPI and non-MPI version.
10. `-x` executes the makefile immediately.
11. `args` are a list of directories and files to be searched for targets and dependencies.

6.241.3 Makefile structure

A *sourcefile* is any file with a source file suffix (currently `.F, .F90, .c, .f. .f90`). An *includefile* is any file with an include file suffix (currently `.H, .fh, .h, .inc`). A valid sourcefile can also be an includefile.

Each sourcefile in the list is presumed to produce an object file with the same basename and a `.o` extension in the current working directory. If more than one sourcefile in the list would produce identically-named object files, only the first is used and the rest are discarded. This permits the use of overlays: if `dir3` contained the basic source code, `dir2` contained bugfixes, and `dir1` contained mods for a particular run, `mkmf dir1 dir2 dir3` would create a makefile for correct compilation. Please note that precedence *descends* from left to right. This is the conventional order used by compilers when searching for libraries, includes, etc: left to right along the command line, with the first match invalidating all subsequent ones. See the Examples section for a closer look at precedence rules.

The makefile currently runs `$(FC)` on fortran files and `$(CC)` on C files (unless the `-w` flag is specified). Flags to the compiler can be set in `$(FFLAGS)` or `$(CFLAGS)`. The final loader step executes `$(LD)`. Flags to the loader can be set in `$(LDFLAGS)`. Preprocessor flags are used by `.F`, `.F90` and `.c` files, and can be set in `$(CPPFLAGS)`. These macros have a default meaning on most systems, and can be modified in the template file. The predefined macros can be discovered by running `make -p`.

In addition, the macro `$(CPPDEFS)` is applied to the preprocessor. This can contain the `cpp #defines` which may change from run to run. `cpp` options that do not change between compilations should be placed in `$(CPPFLAGS)`.

If the `-w` flag is given the commands run are `$(MPIFC)` on fortran files, `$(MPICC)` on C files, and `$(MPILD)` for the loader step. The flags retain their same values with or without the `-w` flag. (This is a local addition.)

Includefiles are recursively searched for embedded includes.

For `emacs` users, the make target `TAGS` is always provided. This creates a TAGS file in the current working directory with a cross-reference table linking all the sourcefiles. If you don't know about emacs tags, please consult the emacs help files! It is an incredibly useful feature.

The default action for non-existent files is to `touch` them (i.e create null files of that name) in the current working directory.

All the object files are linked to a single executable. It is therefore desirable that there be a single main program source among the arguments to `mkmf`, otherwise, the loader is likely to complain.

6.241.4 Treatment of [args]

The argument list `args` is treated sequentially from left to right. Arguments can be of three kinds:

- If an argument is a sourcefile, it is added to the list of sourcefiles.
- If an argument is a directory, all the sourcefiles in that directory are added to the list of sourcefiles.
- If an argument is a regular file, it is presumed to contain a list of sourcefiles. Any line not containing a sourcefile is discarded. If the line contains more than one word, the last word on the line should be the sourcefile name, and the rest of the line is a file-specific compilation command. This may be used, for instance, to provide compiler flags specific to a single file in the sourcefile list.

```
a.f90
b.f90
f90 -Oaggress c.f90
```

This will add `a.f90, b.f90` and `c.f90` to the sourcefile list. The first two files will be compiled using the generic command `$(FC) $(FFLAGS)`. But when the make requires `c.f90` to be compiled, it will be compiled with `f90 -Oaggress`.

The current working directory is always the first (and top-precedence) argument, even if `args` is not supplied.

6.241.5 Treatment of [-c cppdefs]

The argument `cppdefs` is treated as follows. `cppdefs` should contain a comprehensive list of the `cpp #defines` to be preprocessed. This list is compared against the current "state", maintained in the file `.cppdefs` in the current working directory. If there are any changes to this state, `mkmf` will remove all object files affected by this change, so that the subsequent `make` will recompile those files. Previous versions of `mkmf` attempted to `touch` the relevant source, an operation that was only possible with the right permissions. The current version works even with read-only source.

The file `.cppdefs` is created if it does not exist. If you wish to edit it by hand (don't!) it merely contains a list of the `cpp` flags separated by blanks, in a single record, with no newline at the end.

`cppdefs` also sets the `make` macro `CPPDEFS`. If this was set in a template file and also in the `-c` flag to `mkmf`, the value in `-c` takes precedence. Typically, you should set only `CPPFLAGS` in the template file, and `CPPDEFS` via `mkmf -c`.

6.241.6 Treatment of includefiles

Include files are often specified without an explicit path, e.g:

```
#include "config.h"
```

`mkmf` first attempts to locate the includefile in the same directory as the source file. If it is not found there, it looks in the directories listed as arguments, maintaining the same left-to-right precedence as described above.

This follows the behaviour of most f90 compilers: includefiles inherit the path to the source, or else follow the order of include directories specified from left to right on the `f90` command line, with the `-I` flags *descending* in precedence from left to right.

If you have includefiles in a directory `dir` other than those listed above, you can specify it yourself by including `-Idir` in `$(FFLAGS)` in your template file. Includepaths in the template file take precedence over those generated by `mkmf`. (I suggest using `FFLAGS` for this rather than `CPPFLAGS` because fortran `includes` can occur even in source requiring no preprocessing).

6.241.7 Examples

The template file for the SGI MIPSpro compiler contains:

```
FC = f90
LD = f90
CPPFLAGS = -macro_expand
FFLAGS = -d8 -64 -i4 -r8 -mips4 -O3
LDFLAGS = -64 -mips4 $(LIBS)
LIST = -listing
```

The meaning of the various flags may be divined by reading the manual. A line defining the `make` macro LIBS, e.g:

```
LIBS = -lmpi
```

may be added anywhere in the template to have it added to the link command line.

Sample template files for different OSs and compilers are available in the directory `/net/vb/public/bin`.

This example illustrates the effective use of `mkmf`'s precedence rules. Let the current working directory contain a file named `path_names` containing the lines:

```
updates/a.f90
updates/b.f90
```

The directory `/home/src/base` contains the files:

```
a.f90
b.f90
c.f90
```

Typing `mkmf path_names /home/src/base` produces the following `Makefile`:

```
# Makefile created by mkmf

.DEFAULT:
        -touch $@
all: a.out
c.o: /home/src/base/c.f90
        $(FC) $(FFLAGS) -c      /home/src/base/c.f90
a.o: updates/a.f90
        $(FC) $(FFLAGS) -c      updates/a.f90
b.o: updates/b.f90
        $(FC) $(FFLAGS) -c      updates/b.f90
./c.f90: /home/src/base/c.f90
        cp /home/src/base/c.f90 .
./a.f90: updates/a.f90
        cp updates/a.f90 .
./b.f90: updates/b.f90
        cp updates/b.f90 .
SRC = /home/src/base/c.f90 updates/a.f90 updates/b.f90
OBJ = c.o a.o b.o
OFF = /home/src/base/c.f90 updates/a.f90 updates/b.f90
clean: neat
        -rm -f .cppdefs $(OBJ) a.out
neat:
        -rm -f $(TMPFILES)
localize: $(OFF)
```

(continues on next page)

(continued from previous page)

```
        cp $(OFF) .
TAGS: $(SRC)
        etags $(SRC)
tags: $(SRC)
        ctags $(SRC)
a.out: $(OBJ)
        $(LD) $(OBJ) -o a.out $(LDFLAGS)
```

Note that when files of the same name recur in the target list, the files in the `updates` directory (specified in `path_names`) are used rather than those in the base source repository `/home/src/base`.

Assume that now you want to test some changes to `c.f90`. You don't want to make changes to the base source repository itself prior to testing; so you make yourself a local copy.

```
$ make ./c.f90
```

You didn't even need to know where `c.f90` originally was.

Now you can make changes to your local copy `./c.f90`. To compile using your changed copy, type:

```
$ mkmf path_names /home/src/base
$ make
```

The new Makefile looks like this:

```
# Makefile created by mkmf

.DEFAULT:
        -touch $@
all: a.out
c.o: c.f90
        $(FC) $(FFLAGS) -c      c.f90
a.o: updates/a.f90
        $(FC) $(FFLAGS) -c      updates/a.f90
b.o: updates/b.f90
        $(FC) $(FFLAGS) -c      updates/b.f90
./a.f90: updates/a.f90
        cp updates/a.f90 .
./b.f90: updates/b.f90
        cp updates/b.f90 .
SRC = c.f90 updates/a.f90 updates/b.f90
OBJ = c.o a.o b.o
OFF = updates/a.f90 updates/b.f90
clean: neat
        -rm -f .cppdefs $(OBJ) a.out
neat:
        -rm -f $(TMPFILES)
localize: $(OFF)
        cp $(OFF) .
TAGS: $(SRC)
        etags $(SRC)
tags: $(SRC)
        ctags $(SRC)
a.out: $(OBJ)
        $(LD) $(OBJ) -o a.out $(LDFLAGS)
```

Note that you are now using your local copy of `c.f90` for the compile, since the files in the current working directory always take precedence. To revert to using the base copy, just remove the local copy and run `mkmf` again.

This illustrates the use of `mkmf -c`:

```
$ mkmf -c "-Dcppflag -Dcppflag2=2 -Dflag3=string ..."
```

will set `CPPDEFS` to this value, and also save this state in the file `.cppdefs`. If the argument to `-c` is changed in a subsequent call:

```
$ mkmf -c "-Dcppflag -Dcppflag2=3 -Dflag3=string ..."
```

`mkmf` will scan the source list for sourcefiles that make references to `cppflag2`, and the corresponding object files will be removed.

6.241.8 Caveats

In F90, the module name must occur on the same source line as the `module` or `use` keyword. That is to say, if your code contained:

```
use &
this_module
```

it would confuse `mkmf`. Similarly, a fortran `include` statement must not be split across lines.

Two `use` statements on the same line is not currently recognized, that is:

```
use module1; use module2
```

is to be avoided.

`mkmf` provides a default action for files listed as dependencies but not found. In this case, `mkmf` will `touch` the file, creating a null file of that name in the current directory. It is the least annoying way to take care of a situation when cpp `#includes` buried within obsolete `ifdefs` ask for files that don't exist:

```
#ifdef obsolete
#include "nonexistent.h"
#endif
```

If the formatting flag `-f` is used, long lines will be broken up at intervals of 256 characters. This can lead to problems if individual paths are longer than 256 characters.

6.242 Copyright

6.243 Changelog

This file documents the most user-visible changes to the DART code. It is not intended to document every change, but instead is intended to inform people what features are now available or have been removed. Detailed changes are always available through the version control framework.

DART now uses Git for version control but has preserved the revision history from when subversion (and CVS before that) was used. The previous revision numbers can be related to git hashes by searching the output of `git log`

```
0[1011] machine:dartGIT % git log > full_git_log.txt
```

A reminder that since many files were moved or renamed, the best way to get the complete log is to use `git log --follow` for information on individual files.

6.243.1 The changes are now listed with the most recent at the top

Oct 29 2020 : radiance support, MPAS, obs converters Tag: v9.9.0
Use RTTOV (Radiative Transfer for TOVS) routines to support radiance assimilation. - Introduction to DART support for RTTOV - WRF, MPAS, and CAM-FV model interfaces now support radiance assimilation. - Added GOES 16-19 ABI converter
NOTE: The `build_templates/mkmf.template` file has been removed from version control. You must now explicitly copy the best example `mkmf.template` into place before compiling. If there is no `mkmf.template` when you try to build, an error message is displayed.
MPAS regional configurations now supported.
Converted CHANGELOG to a markdown document, put newest content at top.
Converted many HTML documents to markdown - renamed `observations/obs_converters/observations.html` to `observations/obs_converters/README.md` for example.
Updated Publications
declare hexadecimal constants according to the Fortran standard.
GSI2DART converter updated - Thanks to **Craig Schwartz** & **Jamie Bresch**.
The WRF-DART tutorial has been rewritten as `models/wrf/tutorial/README.md`
Hydro-DART (AKA wrf-hydro/DART) has been updated to be Manhattan-compliant. - also support masked bucket - added perturbed forcing capability
The support for POP and CESM2 has been implemented and documented.
`obs_diag` now correctly handles the special case when the observation is properly assimilated or evaluated but the posterior forward operator fails. The posterior DART QC in the `obs_diag_output.nc` should be a '2', not a '4'. The prior DART QC value in obs_diag_output.nc can still be a 7 if need be.
`obs_def_tower_mod.f90` was refactored into `obs_def_tower_mod.f90` and `obs_def_land_mod.f90`.
WRF-Chem/DART documentation and datasets have been updated for Manhattan. WRF-Chem contact information.
Fixed bug in `obs_seq_to_netcdf` to correctly append to existing netCDF files.
Support absolute humidity observations - Thanks to **Michael Ying**.
`DEFAULT_obs_kind_mod.F90` has many added quantities.
new observation converters including (but not limited to): - absolute humidity - streamflow observations from the Mexican water agency - streamflow observations from the USGS - total water storage observations from GRACE - radiance observations from GOES
the following forward operator modules are either new or modified: - (M) `observations/forward_operators/DEFAULT_obs_def_mod.F90` - (M) `observations/forward_operators/obs_def_GRACE_mod.f90` - (A) `observations/forward_operators/obs_def_abs_humidity_mod.f90` - (M) `observations/forward_operators/obs_def_altimeter_mod.f90` - (A) `observations/forward_operators/obs_def_land_mod.f90` - (A) `observations/forward_operators/obs_def_mesonet_mod.f90` - (M) `observations/forward_operators/obs_def_oxygen_ion_density_mod.f90` - (M) `observations/forward_operators/obs_def_reanalysis_bufr_mod.f90` - (M) `observations/forward_operators/obs_def_rel_humidity_mod.f90` - (A) `observations/forward_operators/obs_def_rttov_mod.f90` - (A) `observations/forward_operators/obs_def_streamflow_mod.f90` - (M) `observations/forward_operators/obs_def_tower_mod.f90` - (M) `observations/forward_operators/obs_def_upper_atm_mod.f90` - (A) `observations/forward_operators/rttov_sensor_db.csv`
`fill_inflation_restart` now correctly creates inflation values for all variables in the DART state, regardless of the setting of the `no update` metadata.
GITM is now fully Manhattan compliant.
fix bug in madis rawin converter
avoid computing posterior inflation if using the 'relaxation to prior spread' inflation option – Thanks to **Craig Schwartz**.
add additional reporting options to the `obs_assim_count` utility

6.243.2 Nov 20 2019 : FESOM,NOAH-MP model support, better testing Tag: v9.8.0

- first release entirely from GIT
- fixed bug in `fill_inflation_restart` tool which used the prior inflation mean and sd for both prior and posterior inflation files. now correctly uses the posterior mean/sd if requested.
- fixed a typo in the location test script that prevented it from running
- additional functionality in the quad interpolation code, now supports grids which start at 90 (north) and end at -90 (south).
- if possible, send shorter MPI messages. improves performance on some platforms and MPI implementations.
- add explicit call to `initalize_utilities()` where it was missing in a couple of the WRF utility routines.
- added an example of how to use a namelist to the `text_to_obs.f90` observation converter program.
- Removing the clamping messages in `clamp_variable()` of clamped values
- changed argument names using reserved keywords.
  - `state_vector_io_mod:read_state() 'time' to 'model_time'`
  - `random_seq_mod:random_gamma() 'shape' to 'rshape', 'scale' to 'rscale'`
  - `random_seq_mod:random_inverse_gamma() 'shape' to 'rshape', 'scale' to 'rscale'`
  - `obs_def_mod:init_obs_def() 'kind' to 'obkind', 'time' to 'obtime'`
  - `obs_utilities_mod:  'start' to 'varstart', 'count' to 'varcount'`
- The **FESOM** model is now Manhattan-ready. Thanks to **Ali Aydogdu**
- The **noah** model is now Manhattan-ready and may be used with NOAH-MP.
- bugfixed references to the `documentation` directory that was renamed `guide` to comply with GitHub Pages.
- improved `test_dart.csh` functionality.

Apr 30 2019 : cam-fv refactor, posteriors optional, QC 8 Revision: 13138
The CAM Finite Volume (**cam-fv**) `model_mod.f90` has undergone substantial refactoring to improve simplicity and remove code for unsupported CAM variants while also supporting WACCM and WACCM-X. Namelist changes will be required.
cam-fv setup and scripting support added for CESM 2.1, including advanced archiving and compression
fix for WRF's wind direction vectors when using the Polar Stereographic map projection. Thanks to **Kevin Manning** for the fix.
Add filter namelist option to avoid calling the posterior forward operators and to not create those copies in the `obs_seq.final` file.
Use less memory if writing ensemble member values into the `obs_seq.final` file.
added a DART QC of 8 for failed vertical conversions
updated Matlab scripts support QC=8 and no posterior in obs sequence files.
sampling error correction table now has all ensemble sizes between 3 and 200
`closest_member_tool` can be compiled with other MPI targets
`COSMIC_ELECTRON_DENSITY` has been moved from `obs_def_gps_mod.f90` to `obs_def_upper_atm_mod.f90`, which has new quantities for `ION_O_MIXING_RATIO` and `ATOMIC_H_MIXING_RATIO`
`obs_converters/gps/convert_cosmic_ionosphere.f90` has a test dataset
support for NAG compiler
fixed Intel compiler bug in `lorenz_96` comparing long integers to integer loop indices
`get_maxdist()` now a required routine all location modules
Default routines now create a time variable as `time(time)` to allow multiple files to be concatenated along the unlimited dimension more easily. Also conforms to the netCDF convention for coordinate dimensions.
`obs_impact_tool` handles a continuum of values, not just discrete 0 or 1.
`fill_inflation_restart` now produces files with names consistent with filter defaults.
expanded functionality in `xyz_location_mod.f90`
Removed 'slow' sorting routines from `sort_mod.f90`
replacing some repeated native netCDF library calls with routines from the `netcdf_utilities_mod.f90`
Updated dewpoint equation to avoid dividing by zero given a very unlikely scenario (r12832)
More efficient implementation of adaptive inflation
Yongfei Zhang and *Cecilia Bitz* added improvements to the CICE model and observation converters and forward operators. These changes also use the locations of the 'new' glade filesystem. They used CESM tag: cesm2_0_alpha06n
Worked with Yongfei Zhang to remove prototype codes and more completely document observation converters and data sources for cice assimilation.
removed `allow_missing_in_clm` flag from the `&assim_tools_nml` namelist in the CICE work directory. The flag moved to a different namelist and the CICE model doesn't care about it.
increased the maximum number of input files to `obs_diag` from 100 to 10000.
Updated the `developer_tests` to include more cases.
Updated `oned/obs_diag.f90` to support `obs_seq.out` files.
Better error and informational messages in various routines.

6.243.3 Aug 03 2018 : performance fix for distributed mean Revision: 12758

Important performance fix if model does vertical conversion for localization. Results were not wrong but performance was poor if `distribute_mean = .true.` was selected in the `&assim_tools_nml` namelist.

Now distributing the mean runs in close to the non-distributed time and uses much less memory for large models. This only impacts models which do a vertical conversion of either the observation or state vertical coordinate for localization AND which set `&assim_tools_nml :: distribute_mean = .true.` to use less memory.

When using a distributed mean `convert_all_obs_verticals_first = .true.` should be set. If your observations will impact most of the model state, then `convert_all_state_verticals_first = .true.` can also be set.

Jun 18 2018 : CAM/CESM 2.0, DART QC 8, closest_member_tool Revision: 12682
Support for **cam-fv** assimilations in the CESM 2.0 release. See documentation in `models/cam-fv/doc/README_cam-fv` for details.
`obs_diag` and matlab scripts updated to report statistics on DART QC 8, observation failed vertical conversion
Updates to fix minor problems with the new WRF scripts
Added the `inf_sd_max_change` namelist item to all `input.nml` files for the enhanced inflation option
Revival of the `closest_member_tool`, which now runs in parallel on all ensemble members at one time. This tool can be used as a template for any other tools which need to process something for all ensemble members in parallel.
Revival of the `fill_inflation_restart` tool as a Fortran 90 program. Using `ncap2` is still possible, but if the correct version is not installed or available this tool can be used.
Added more functions to the `netcdf_utilities_mod.f90`

6.243.4 May 21 2018 : enhanced inflation option, scripting Revision: 12591

- Enhanced inflation algorithm added. See the `filter_mod.html` for new documentation on this option.
- Updated WRF scripts for the Manhattan release.
- `obs_diag` reports statistics on DART QC 8, observation failed vertical conversion. Matlab scripts also updated to support QC 8.
- New parallel conversion scripts for GPS Radio Occultation observations and NCEP prepbufr conversions.
- Further updates to documentation files to change KIND to QTY or Quantity.
- Documented required changes when moving from the Lanai/Classic release to Manhattan in `documentation/html/Manhattan_diffs_from_Lanai.html`
- Expanded the routines in the `netcdf_utilities_mod.f90`
- Add an ensemble handle parameter to the 6 ensemble manager routines where it was missing.
- The `advance_time` program can read/generate CESM format time strings (YYYY-MM-DD-SSSSS).
- Fixed a bug in the netcdf read routines that under certain circumstances could report an array was using the unlimited dimension incorrectly.
- Removed the option to try to bitwise reproduce Lanai results; due to the number of changes this is no longer possible.
- Minor bug fixes to the (seldom used) perturb routines in the **WRF** and **mpas_atm** `model_mod.f90` files. (used to add gaussian noise to a single model state to generate an ensemble; this is never the recommended method of starting a new experiment but the code remains for testing purposes.)

- Several remaining model-specific `model_mod_check` programs were removed in favor of a single common program source file.
- Keep `filter_mod.dopplerfold.f90` in sync with `filter_mod.f90`, and `assim_tools_mod.pf.f90` in sync with `assim_tools_mod.f90`.
- Removed makefiles for the obsolete `trans_time` program.

Mar 01 2018 : ROMS, MMC, PMO, mpas_atm debug, etc Revision: 12419
Fix a debug message in the **mpas_atm** model which might have caused a buffer overflow crash when formatting a message for a larger ensemble size.
Update the **ROMS** shell scripts to support PBS, SLURM, as well as LSF. Update the ROMS model_mod html documentation.
Update the default **cam-fv** `input.nml` to have more realistic values for the highest observation assimilated, and for where the ramp starts that decreases the increments at the model top. If running with a higher model top than the default check these items carefully.
Fixed variable type for `time` variables we create in diagnostic files
Miscellaneous minor Bug fixes: - Print format wider for fractional levels in `threed_sphere` locations - Fixed a deallocate call at program shutdown time - Fixed an indexing problem computing **cam-fv** U_WIND observations if the observation used HEIGHT as the vertical coordinate (very unusual). - Fixed grid creation bug in a test program used with `model_mod_check`. Now uses correct spacing for grids in the x,y coordinates. - Fixed an allocate problem in a test interpolate routine.
Add surface pressure to the default state list in the **wrf** `work/input.nml`
`developer_tests/test_dart.csh` can run PMO for more models. required updates to the `work/input.nml` in several directories (wrf, cm1, POP, mpas_atm) to match the current namelist.
several `model_mod_check` programs were combined into a single version that allows for selection of individual tests. many of the input.nml `models/xxx/work/input.nml` files have either had a `&model_mod_check_nml` section added or updated to match the updated interface.
the DART QTYs are now available via the state structure in the **wrf** and **clm** `model_mod`s.
support the NAG compiler better. (contact dart@ucar.edu for more help if you want to use this compiler. some hand work is still needed.)
streamlined the debug output from the `state_structure_info()` call to avoid replicating information that was the same for all variables.
minor formatting change to the dart log file output for the list of observation types being assimilated, evaluated, and using precomputed forward operators.
fixed an uninitialized variable in the BGRID model code in a routine that isn't normally used.
Updated the `threed_sphere` location module documentation with some usage notes about issues commonly encountered.
Fixed an incorrect test when printing out a log message describing if the inflation would be variance-adaptive or not.
Change the location of the POP MDT reference file to be relative to the current run directory and not an absolute file location on cheyenne.
Make the ROMS, CM1, and POP model_mod log namelist information to the namelist log file and not the main DART log file.
Updated several html documentation files, including the `template/model_mod.html` which describes the current model_mod required interfaces.
Updated the instructions for the GSI to DART obs converter to suggest some needed compiler flags in certain cases.
Updated the location module test programs.

6.243.5 Dec 01 2017 : ROMS scripting, debugging aids Revision: 12166

- Added an option to the ROMS model scripting to advance the model ensemble members in parallel using a job array.
- Updated the DART_LAB Matlab GUIs to log a history of the settings and results.
- Added a debug option to the filter namelist, `write_obs_every_cycle`, to output the full `obs_seq.final` during each cycle of filter.
  (Very slow - use only when debugging a filter crash.)
- Allow the test grid in `model_mod_check` to cross the prime meridian for testing longitude interpolation in grids that cross the 360/0 line.

Nov 22 2017 :: minor updates for DA challenge files Revision: 12144
added `obs_seq.in.power` to the Lorenz 96 directory
added new obs types to the workshop version of the `input.nml` assimilation list

6.243.6 Nov 21 2017 : 1D obs_diag fix, 1D power forward operator Revision: 12138

- fixed a bad URL reference in tutorial section 18
- fixed a crash with the 1D version of the observation diagnostics program when including identity observations.
- all models with a `workshop_setup.csh` now build the same set of programs. (some/most did not build obs_diag - which is used in the tutorial)
- added a 1D obs-to-a-power forward operator.
- updates to the matlab plotting routines for NetCDF observation formats
- World Ocean Database (WOD) converter supports partial year conversions and 2013 file formats.

Oct 17 2017 : mpas_atm bug fix, various other updates. Revision: 12002
Fixed a bug in the **mpas_atm** `model_mod` that affected surface observations, in particular altimeter obs. also fixed a bug in the vertical conversion if using 'scale height' as the vertical localization type.
Fixed a bug in the **cam-fv** `model_mod` which might have excluded observations with a vertical coordinate of height (meters) which were in fact below the equivalent highest_obs_pressure_Pa namelist setting. also fixed a possible memory leak.
Added two new modules: `options_mod.f90` and `obs_def_utilities_mod.f90` this was required so we didn't have circular dependencies in our modules as we reused common code in more places. We have updated all the `path_names*` files which are in the repository. if you have your own path_names files you may need to add these new modules to your path lists. - `assimilation_code/modules/utilities/options_mod.f90` - `observations/forward_operators/obs_def_utilities_mod.f90`
Removed `QTY_SURFACE_TEMPERATURE` from the default obs quantities list and added `QTY_2M_SPECIFIC_HUMIDITY`. `QTY_2M_TEMPERATURE` exists for atmospheric models, and `QTY_SKIN_TEMPERATURE` and `QTY_SOIL_TEMPERATURE` exist for other models. if you were using `QTY_SURFACE_TEMPERATURE` please replace it with the corresponding other temperature quantity.
Updated and improved the observation converter for ionospheric observations from the COSMIC GPS satellite.
Updated the **cam-fv** scripts for cesm2_0_beta05.
Updated the Matlab diagnostics documentation. 'help DART' or 'doc DART' will give an overview of the available Matlab diagnostics shipped with the dart distribution.
Added the observation type `COSMIC_ELECTRON_DENSITY` to the `obs_def_upper_atm_mod`
`dart_to_clm` and `clm_to_dart` were resurrected to correctly handle conversions for the SWE (snow water equivalent) field.
Updated the channel and column location modules to be compatible with the current required interfaces.
Updated the `model_mod_check.f90` program (most often used when porting DART to a new model). there is now more control over exactly which tests are being run. updated the nml and html documentation files to match the current code and describe the tests in more detail.
Fixed a misleading status message in the `obs_sequence_tool` when all obs are excluded by the min/max lon/lat box namelist items. the incorrect message blamed it on observation height instead of the bounding box.
Added some additional debugging options to the mpi utilities module. if you have problems that appear to be MPI related, contact us for more help in enabling them.
Improved some error messages in `location_io_mod` and `state_structure_mod`

6.243.7 Aug 2 2017 : single filenames, random distributions, bug fixes. Revision: 11864

- added code to support listing input and output filenames directly in the namelist instead of having to go through an indirect text file. most useful for programs that take a single input and output file, but works for all cases.
- bug fix in `location_io_mod.f90` that affected `obs_seq_to_netcdf` (error in adding vertical location types to output file).
- fix to `convert_gpsro_bufr.f90` converter (GPS obs from BUFR files) that failed if r8 defined to be r4.
- added draws from gamma, inverse gamma, and exponential distributions to the random sequence module.
- various updates to the **cam** scripts to work more smoothly with the most recent CIME changes and DART Manhattan updates.
- added `QTY_CWP_PATH` and `QTY_CWP_PATH_ZERO` to the default quantities list for the `obs_def_cwp_mod.f90` forward operator.
- improved some error messages in the diagnostic matlab scripts

July 18 2017 :: bug fixes, documentation updates. Revision: 11830
fixed bug in `obs_impact_tool` when generating the run-time table. specifying a generic quantity resulted in selecting the wrong specific obs types.
fixed a bug that would not allow filter to start from a single ensemble member if `single_file_in = .true.`
updates to HTML documentation especially for types/quantities (replacing kinds)
updates to `input.nml` namelists, code comments, and shell scripts where names changed from `restart` to `state` for input and output files.

6.243.8 July 7th 2017 : cam-fv, mpas_atm scripts, single file i/o. Revision: 11807

- **mpas_atm**: scripts completely revised for the Manhattan release. Many thanks to **Soyoung Ha** and **Ryan Torn** for the contributed code.
- **cam-fv**: scripts and `model_mod.f90` updated for cesm2_0_beta05.

Single File I/O:

- Now we are able to run `single_file_in` and `single_file_out` with MPI.
- `single_file_io_mod.f90` has been removed and its functionality has been moved to `direct_netcdf_mod.f90`.
- `single_file_io_mod.f90` has been removed from all of the `path_names_*` files in the repository. (Remove it from any private `path_names_*` files.)

June 27rd 2017 : CICE 5, model_mod_check, tutorial. Revision: 11770
Updated support for CICE5.
Updated support for `model_mod_check` - now compatible with netCDF input files, input is through [input,output]_state_files namelist variable (variables renamed).
Ensured consistency between low-order namelists and the updated DART tutorial. Updated documentation of many namelists. More to come.
`location_mod`: namelist variable `maintain_original_vert` was deprecated, it is now removed. You must remove it from your existing namelists or DART will error out immediately.
`obs_diag`: namelist variables `rat_cri` and `input_qc_threshold` have been deprecated for years, they have been removed. You must remove them from your existing namelists or obs_diag will error out immediately.

6.243.9 Jun 2nd 2017 : tutorial, DART_LAB, and various updates. Revision: 11696

- bring the DART tutorial pdf slides up to date with the current release.
- include new GUIs with adaptive inflation options in DART_LAB:
  - `oned_model_inf.m`
  - `run_lorenz_96_inf.m`
- added the **lorenz_96_2scale** model - additional kinds of `QTY_SMALL_SCALE_STATE` and `QTY_LARGE_SCALE_STATE` added as required.
- add useful attributes to the variables in the diagnostic files
- updates and minor bug fixes to the matlab diagnostic scripts
- updates to the default input.nmls for models
- updates to the **cam-fv** shell scripts to work with the CESM2.0 framework

- updates to the **cam-fv** model_mod for support of cam-chem variables Added more QUANTITIES/KINDS for chemistry species. Removed support for 'stand-alone' **cam** and **cam-se** (**cam-se** will be a separate 'model').
- major bug fix in the **simple_advection** model_mod: Fixed an error with the layout of the state vector.
- obs_def_radar_mod: Fixed a serious bug in the fall velocity forward operator. If the fall speed field is not in the state the test for a bad istatus from the interpolate() call was looking at the wrong variable and returning ok even if interpolate() had set bad values.
- bug fix in the **wrf** model_mod for fields which have a vertical stagger
- fix to the makefiles for the GSI2DART observation converter
- added additional netcdf and location utility routines
- various fixes to documentation and test code
- renamed QTY_RAW_STATE_VARIABLE to QTY_STATE_VARIABLE (RAW is redundant)
- direct_netcdf_mod: Renamed limit_mem to buffer_state_io. buffer_state_io is now a logical that states if a variable that tells DART it it should read and write variables all at once or variable-by-variable.

May 5th 2017 : major changes to model_mod interfaces. Revision: 11615
A long-awaited overhaul of the model_mod interfaces. All models which are in our subversion repository and are supported in the Manhattan release have been updated to match the new interfaces. If you have model_mods with extensive changes, our recommendation is to diff your changes with the version you checked out and insert those changes into the new version. The changes for this update are unfortunately extensive.
The detailed list of changes:
`model_mod::get_state_meta_data()` is no longer passed an ensemble_handle as the first argument. it should not do vertical coordinate conversion. that will be done as a separate step by `convert_vertical_state()`
`model_mod::vert_convert` is replaced by `convert_vertical_state()` and `convert_vertical_obs()` Any vertical conversion code that was in `get_state_meta_data` should be moved to `convert_vertical_state()` which has access to the state vector index, so the code should move easily.
`model_mod::query_vert_localization_coord` is no longer a required interface `model_mod::get_close_maxdist_init` is not longer a required interface `model_mod::get_close_obs_init` is not longer a required interface
`model_mod::get_close_obs` has a different calling convention and is split into `get_close_obs()` and `get_close_state()`. the close obs routine is passed both the obs types and quantities, and the close state routine is passed both the state quantities and the state index, for ease in vertical conversion if needed.
`model_mod::nc_write_model_vars()` is deprecated for now; it may return in a slightly different form in the future.
`model_mod::nc_write_model_atts()` is now a subroutine with different arguments. it should now only write any global attributes wanted, and possibly some grid information. it should NOT write any of the state variables; those will be written by DART routines.
`model_mod::get_model_size()` needs to return an `i8` (a long integer) for the size.
A new module `default_model_mod` supplies default routines for any required interfaces that don't need to be specialized for this model.
A new module `netcdf_utilities_mod` can do some simple netcdf functions for you and we plan to add many more over the next couple months.
`model_mod::get_model_time_step` has been replaced by `shortest_time_between_assimilations()` since in fact it has always controlled the minimum time filter would request a model advance and never had anything to do with the internal time step of the dynamics of the model.
We have removed `output_state_vector` from the namelist of all model_mods since we no longer output a single 1d vector. all i/o is now in netcdf format.
Models now have more control over when vertical conversion happens - on demand as needed, or all up front before assimilation.
Models that were doing vertical conversion in `get_state_meta_data` should set: ``` &assim_tools_nml convert_all_state_verticals_first = .true. convert_all_obs_verticals_first = .true.
Models which were not should set: convert_all_state_verticals_first = .false. convert_all_obs_verticals_first = .true.
The `location_mod::vert_is_xxx()` routines have become a single `is_vertical(loc, "string")` where string is one of: "PRESSURE", "HEIGHT", "SURFACE", "LEVEL", "UNDEFINED", "SCALE_HEIGHT"
Models doing vertical localization should add a call to `set_vertical_localization_coord()` in their `static_init_model()` routine to tell dart what vertical coordinate system they are expecting to convert to for vert localization
Most `path_names_xxx` files have been updated to add additional modules. compare against what is checked out to see the differences.
Some of the internal changes include pulling common code from the locations modules into a `location_io_mod` which contains common functions for creating and writing 'location' variables for any location type.
`QTY_RAW_STATE_VARIABLE` is redundant and was shortened to `QTY_STATE_VARIABLE`
Many utility programs use the `template/model_mod.f90` because they do not depend on any model-specific functions. this file was also updated to match the new interfaces.
The `obs_impact` facility is enabled in the `assim_tools` namelist. you can use the `obs_impact_tool` to construct a table which prevents one class of observations from impacting another class of state.
Sampling Error Correction now reads the values it needs from a single netcdf file found in `assimilation_code/programs/gen_sampling_err_table/work`. Copy it to the same directory as where filter is running. All ensemble sizes which were previously in `final_full.XX` files are included, and there is a tool to generate and append to the file any other ensemble size required.

6.243.10 April 27th 2017 : diagnostic file changes. Revision: 11545

Two additional Diagnostic Files (forecast and analysis) in Filter which can be set with the namelist option (stages_to_write)

- **input** writes out mean and sd if requested.
  - For low order models, mean and sd are only inserted into restart files with a single time step.
- **forecast**
  - contains the forecast and potentially the mean and sd for the, this is mostly important for lower order models which cycle
- **preassim** before assimilation
  - No Inflation: same as forecast
  - Prior Inf: the inflated ensemble and damped prior inf
  - Post Inf: same as forecast
  - Prior and Post Inf: the inflated ensemble and damped prior inf
- **postassim** after assimilation (before posterior infation)
  - No Inflation: same as analysis
  - Prior Inf: same as analysis
  - Post Inf: assimilated ensemble and damped posterior inflation
  - Prior and Post Inf: assimilated ensemble and damped posterior inflation
- **analysis** after assimilation and before potentially update posterior inflation ensemble and updated prior inf
  - No Inflation: assimilated ensemble
  - Prior Inf: assimilated ensemble and updated prior inf
  - Post Inf: post inflated ensemble and updated posterior inflation
  - Prior and Post Inf: post inflated ensemble and updated prior inf and posterior inflation
- **output**
  - a single time step of the output ensemble and potentially updated prior inf and posterior inflation

Feb 15th 2017 : filter updates. Revision: 11160
The postassim diagnostics file was being incorrectly written after posterior inflation was applied. It is now written immediately after the assimilation update, and then posterior inflation, if enabled, is applied.
Sampling Error Correction now reads data from a single netcdf file for any ensemble size. To add other sizes, a program can generate any ensemble size and append it to this file. The default file is currently in `system_simulation`:
`system_simulation/work/sampling_error_correction_table.nc`
Filter and PMO no longer need the "has_cycling" flag.
Changes to the filter_nml are :
`has_cycling` REMOVED for low order models
Changes to the perfect_model_obs_nml are :
`has_cycling` REMOVED for low order models

6.243.11 Feb 15th 2017 : rma_single_file merge changes. Revision: 11136

Filter and PMO can now run with multiple cycles for low order models. The output for this is only supported with single file output (members, inflation, mean, sd are all in the same file).

Added matlab support for diagnostics format in lower order models.

#### Changes to the filter_nml are :

- `output_restart` RENAMED to `output_members`
- `restart_in_file_name` RENAMED to `input_state_file_list`
- `restart_out_file_name` RENAMED to `output_state_file_list`
- `single_restart_file_in` RENAMED to `single_file_in`
- `single_restart_file_out` RENAMED to `single_file_out`
- `input_state_files` ADDED - not currently working
- `output_state_files` ADDED - not currently working
- `has_cycling` ADDED for low order models

Changes to the perfect_model_obs_nml are :

- `start_from_restart` RENAMED `read_input_state_from_file`
- `output_restart` RENAMED `write_output_state_to_file`
- `restart_in_file_name` RENAMED `input_state_files`
- `restart_out_file_name` RENAMED `output_state_files`
- `single_file_in` ADDED for low order models
- `single_file_out` ADDED for low order models
- `has_cycling` ADDED for low order models

Jan 13th 2017 : rma_fixed_filenames merge changes. Revision: 10902
Specific namelist changes include:
1. Earlier versions of the RMA branch code supported both direct NetCDF reads/writes and the original binary/ascii DART format res
1. Diagnostic and state space data (such as inflation, mean and sd information) that were previously stored in {Prior,Posterior}_Diag.n
1. There is no longer support for observation space inflation (i.e. inf_flavor = 1). Contact us at dart@ucar.edu if you have an interest in
Changes to the filter_nml are :
`restart_in_file_name` has been replaced with `input_restart_file_list`. The namelist must contain one or more file n
`restart_out_file_name` has been replaced with `output_restart_file_list`. Same format as `input_restart_fil`
`inf_in_file_name` REMOVED, now have fixed names of the form `input_{prior,posterior}inf_{mean,sd}.nc`
`inf_out_file_name` REMOVED, now have fixed names of the form `output_{prior,posterior}inf_{mean,sd}.nc`
`inf_diag_filename` REMOVED
`inf_output_restart` REMOVED, inflation restarts will be written out if inflation is turned on
`output_inflation` REMOVED, inflation diagnostic files will be written if inflation is turned on
`stages_to_write` There is more control over what state data to write. Options are at stages : ‘input’, ‘preassim’, postassim’, ‘outp
`write_all_stages_at_end` important for large models - all output file I/O is deferred until the end of filter, but will use more m
`output_restart_mean` renamed output_mean
`output_restart` renamed output_restarts

direct_netcdf_{read,write} REMOVED, always true
restart_list_file renamed input_restart_file_list
single_restart_file_in renamed single_file_in
single_restart_file_out renamed single_file_out
add_domain_extension REMOVED
use_restart_list REMOVED
overwrite_state_input REMOVED, equivalent functionality can be set with single_restart_file_in = single_
Changes to the perfect_model_obs_nml are :
restart_in_filename renamed restart_in_file_names takes a NetCDF file. For multiple domains you can specify a lis
direct_netcdf_{read,write} REMOVED, always true
Changes to the state_space_diag_nml are :
single_file REMOVED, diagnostic files are now controlled in filter_nml with stages_to_write
make_diagnostic_files REMOVED, no longer produce original Prior_Diag.nc and Posterior_Diag.nc
netCDF_large_file_support REMOVED, always true
Changes to the state_vector_io_nml are :
write_binary_restart_files REMOVED
Changes to the ensemble_manager_nml are :
flag_unneeded_transposes – REMOVED
Changes to the integrate_model_nml are :
advance_restart_format – REMOVED, only supporting NetCDF format.
Scripting with CESM
See models/cam-fv/scripts_cesm1_5/assimilate.csh for an example of how to handle the new filename conventions.
(To help find things: input_priorinf_mean output_priorinf_mean) {in,out}put_{prior,post}inf_{mean,sd}.nc ARE i
! stage_name is {input,preassim,postassim,output} ! base_name is {mean,sd,{prior,post}inf_{mean,sd}} from filter/filt
This shows where inflation file names are defined. > grep -I set_file_metadata */*.f90 | grep inf filter/filter_mod.f90: call s
subroutine set_member_file_metadata(file_info, ens_size, my_copy_start) call set_file_metadata(file_info, icopy, stage_name, base_n
subroutine set_stage_file_metadata(file_info, copy_number, stage, base_name, desc, offset) write(string1,'(A,''.nc'')') trim(stage_nam
subroutine set_explicit_file_metadata(file_info, cnum, fnames, desc) file_info%stage_metadata%filenames(cnum,idom) = trim(fnames
function construct_file_names(file_info, ens_size, copy, domain) write(construct_file_names, '(A,''*member*'', I4.4, A,''.nc'')') &
Also see harnesses/filename_harness/files: ENS_MEAN_COPY PriorDiag_mean.nc
ADDITIONAL NOTES :
1. currently the closest_member_tool is broken but plans on being fixed soon. 1. restart_file_tool and most model_to_dart/dart_to_mo

6.243.12 Ancient history

To see previous history, use:

```
$ git log --follow
$ git diff --name-status XXXX YYYY
```

where XXXX and YYYY are commits or branches.

6.244 404 Error

The requested page could not be found.