MITgcm Development HOWTO

Ed Hill III

eh3@mit.edu

Revision History

Revision 0.01 2003-08-07 Revised by: eh3
Initial version.
Revision 0.02 2010-01-21 Revised by: jmc
Update links.
Revision 0.03 2010-04-25 Revised by: jmc
Add subsection "Developer settings" (under CVS Repository).
Revision 0.04 2011-04-24 Revised by: jmc
Update subsection "The verification suite".

This document describes how to develop software for the MITgcm project.

Table of Contents

1. Introduction

The purpose of this document is to help new developers get "up to speed" with MITgcm development.

1.1. New Versions of This Document

You can obtain the latest version of this document online (http://mitgcm.org/public/docs.html) in various formats.

1.2. Feedback and corrections

If you have questions or comments about this document, please feel free to contact the authors (mailto:MITgcm-support@mitgcm.org).

2. Background

2.1. User Manual

Before jumping into development, please familiarize yourself with the MITgcm user manual (http://mitgcm.org/public/docs.html). This document contains volumes of useful information and is included here by reference.

2.2. Prerequisites

To develop for MITgcm project you will need a UNIX or UNIX-like set of build tools including the following:

> CVS client, make or (preferably) GNU make, FORTRAN compiler, C compiler, [ba]sh and [t]csh shells, PERL, LaTeX and LaTeX2HTML

Essentially all of the work described here has been tested on recent versions of Red Hat Linux (eg. 7.3 through 9). Except where noted, all shell commands will be provided using bash syntax.

3. CVS Repository

3.1. Layout

Unlike many open source projects, the MITgcm CVS tree does not follow a simple "src", "docs", "share", and "test" directory layout. Instead, there are multiple higher-level directories that each, to some extent, depend upon the presence of the others. The tree currently resembles:

```
gcmpack/
  CVSROOT               -hidden-
```

```
MITgcm                code
     bin                      empty
     doc                      basic developpment documentation
     eesupp                   execution environment support code (wrapper)
     exe                      empty
     jobs                     runtime shell scripts for
                                various platforms (not maintained)
     lsopt                    line search
     model                    main dynamics (core)
     optim                    line search interface
     pkg                      alternate and optional numerics, etc.
     tools                    scripts to build (and test)
     utils                    pre/post processing tools (matlab, ..)
     verification             standard regression tests + examples
                                    + documented examples (tutorials)
     tutorial_examples        (only in release1 branch)

MITgcm_contrib        contributed code

acesgrid.org          build acesgrid web site
development           experimental stuff
gfd_lab               -?-
manual                source of MITgcm documentation
mitgcm.org            build web site
old_develop           old and early development source
     misc                 -?-
     models               -?-
     packages             -?-
     preprocess           -?-
pdfs                  some pdfs
planetinabottle.org   unfinished web site
www.ecco-group.org    build ecco web site ?
```

3.2. Branches

As shown in the online ViewCVS-generated tree (http://mitgcm.org/viewvc/MITgcm/MITgcm/model/src/forward_step.F?view=graph), the MITgcm codebase is split into branches or "lines" under which development proceeds. The main line of development is referred to as the "MAIN" version of the code.

Periodically, a "Release" branch is formed from the "MAIN" development branch. This is done in order to create a relatively stable reference point for both users and developers. The intent is that once a release branch has been created, only bug-fixes will be added to it. Meanwhile, development (which might "break" or otherwise render invalid the documentation, tutorials, and/or examples contained within a release branch) is allowed to continue along the MAIN line.

3.3. Developer settings

CVS is a convenient tool to keep up-to-date a personal copy of the MITgcm code (see: using CVS (http://mitgcm.org/public/using_cvs.html)). The same tool is used by developers to incorporate any change into the repository. However, this later function requires specific settings, as detailed here after

1. You will need an account (login access) to the server "mitgcm.org" (curently: `mitgcmcvs.mit.edu`) with the proper group setting (e.g., group "gcmctrb" to add/modify code into MITgcm_contrib). This kind of account is granted only upon well motivated request (we recommend to ask your senior MITgcm-collaborator to send such request to marshall-admin at techsquare.com with Cc to Chris Hill for approval).

   The access to the server `mitgcm.org` is through ssh-key authorization which will need to be set properly on both side (on your local machine and on your server account). You need to be able to ssh to `mitgcm.org` (or `ssh MY_USER_NAME@mitgcm.org` in case of different user-name on both sides) to proceed further.

2. You need to register to the mitgcm-cvs (http://mailman.mitgcm.org/mailman/listinfo/mitgcm-cvs) mailing list. This ensures that other developers will receive email notification when you make changes; you will also receive such email when others make changes to the repository.

3. It is highly recommended that you register also to the mitgcm-devel (http://mailman.mitgcm.org/mailman/listinfo/mitgcm-devel) mailing list (expect a short delay for this request to be processed). This list is intended for developer discussions.

4. The standard CVS-anonymous mode (using "cvsanon", as mentionned here (http://mitgcm.org/public/source_code.html)) does not provide check-in ("cvs commit") permission. Instead, you will need to set our CVS environment as follow:

   ```
   $ export CVS_RSH=ssh
   $ export CVSROOT=':ext:MY_USER_NAME@mitgcm.org:/u/gcmpack'
   ```

   The reason for such limitation is that when downloading a directory, e.g.: `myCopy`, from the CVS repository (e.g., `MITgcm_contrib/thisPart`) using the command:

   ```
   $ cvs co -P -d myCopy MITgcm_contrib/thisPart
   ```

   the type of CVS environment which has been used is stored in the file `myCopy/CVS/Root`. This prevent to re-use, for cvs-commit purpose, a cvs local copy (`myCopy`) which was obtained using the CVS anonymous mode.

5. At this stage, you should be able to send your modified source file (e.g., `src_file`) from your local copy directory (`myCopy`) to the CVS repository (`MITgcm_contrib/thisPart`) using the command "cvs commit":

   ```
   $ cd myCopy
   $ cvs -n update        (optional; check if new changes have been made)
   $ cvs diff src_file    (optional; list your changes)
   $ cvs commit src_file
   ```

   It is essential that you provide a short description of the changes you made to `src_file` as you check-in this file (the "cvs commit" command automatically opens your standard editor for this purpose).

   Note: Please ignore the following warnings that the "cvs commit" command produces if you are not part of the "gcmpack" group:

   ```
   cvs commit: failed to create lock directory for `/u/gcmpack/CVSROOT'
   (/u/gcmpack/CVSROOT/#cvs.history.lock): Permission denied
   cvs commit: failed to obtain history lock in repository `/u/gcmpack'
   ```

   These warnings are not affecting the changes you made to the CVS repository.

3.4. Main code development

(formerly named "Tagging" ; this section needs an update)

The intent of tagging is to create "known-good" checkpoints that developers can use as references. Traditionally, MITgcm tagging has maintained the following conventions:

1. Developer checks out code into a local CVS-managed directory, makes various changes/additions, tests these edits, and eventually reaches a point where (s)he is satisfied that the changes form a new "useful" point in the evolution of the code.
2. The developer then runs the testreport (http://mitgcm.org/viewvc/MITgcm/MITgcm/verification/testreport) shell script to see if any problems are introduced. While not intended to be exhaustive, the test cases within the verification directory do provide some indication whether gross errors have been introduced.
3. Having satisfied him- or herself that the changes are ready to be committed to the CVS repository, the developer then:
   a. adds a "checkpointXY_pre" comment (where X is a checkpoint number and Y is a letter) to the tag-index (http://mitgcm.org/viewvc/MITgcm/MITgcm/doc/tag-index) file and checks it into the CVS repository
   b. submits the set of changes to the CVS repository and adds comments to `tag-index` describing what the changes are along with a matching "checkpointXY_post" entry

The result of this tagging procedure is a sequence of development checkpoints with comments which resembles:

```
checkpoint50e_post
o make KPP work with PTRACERS
 - fix gad_calc_rhs to call new routine kpp_transport_ptr, which is
   nearly a copy of kpp_transport_s
 - there is no analogue to SurfaceTendencyS, so I have to use
   gPtr(of the surface layer) instead
o add a new platform SunFire+mpi (SunFire 15000) to genmake
checkpoint50e_pre

checkpoint50d_post
o change kpp output from multiple-record state files to single-record state
  files analogous to write_state.F
o reduce the output frequency of cg3d-related stuff to the monitor frequency,
  analogous to the cg2d-related output.
o fix small problem with in ptracers_write_checkpoint.F: len(suff)=512,
  so that writing to internal file fn (with length 512) fails.
checkpoint50d_pre
```

This information can be used to refer to various stages of the code development. For example, bugs can be traced to individual sets of CVS checkins based upon their first appearance when comparing the results from different checkpoints.

4. Coding for MITgcm

4.1. Build Tools

Many Open Source projects use the "GNU Autotools" to help streamline the build process for various Unix and Unix-like architectures. For a user, the result is the common "configure" (that is, "`./configure && make && make install`") commands. For MITgcm, the process is similar. Typical commands are:

```
$ genmake2 -mods=../code
$ make depend
$ make
```

The following sections describe the individual steps in the build process.

4.1.1. The `genmake2` Utility

(Note: the older `genmake` has been replaced by `genmake2`)

The first step in any MITgcm build is to create a Unix-style `Makefile` which will be parsed by `make` to specify how to compile the MITgcm source files. For more detailed descriptions of what the make tools are and how they are used, please see:

- http://www.gnu.org/software/make/make.html (http://www.gnu.org/software/make/make.html)
- http://www.oreilly.com/catalog/make2/ (http://www.oreilly.com/catalog/make2/)

Genmake can often be invoked successfully with a command line as simple as:

```
$ genmake2 -mods=../code
```

However, some systems (particularly commercial Unixes that lack a more modern "/bin/sh" implementation or that have shells installed in odd locations) may require an explicit shell invocation such as one of the following:

```
$ /usr/bin/sh genmake2 -make=gmake  -mods=../code
$ /opt/gnu/bin/bash genmake2 -ieee -make=/usr/local/bin/gmake -mods=../code
```

The genmake2 code has been written in a Bourne and BASH (v1) compatible syntax so it should work with most "sh" and all recent "bash" implementations.

As the name implies, `genmake2` generates a `Makefile`. It does so by first parsing the information supplied from the following sources

1. a `gemake_local` file in the current directory
2. directly from command-line options
3. an "options file" as specified by the command-line option `-optfile='FILENAME'`
4. a `packages.conf` file (in the current directory or in one of the "MODS" directories, see below) which contains the specific list of packages to compile

then checking certain dependency rules (the package dependencies), and finally writing a `Makefile` based upon the source code that it finds. For convenience within various Unix shells, `genmake2` supports both "long"- and "short"-style options. A complete list of the available options can be obtained from:

```
$ genmake2 -help
```

The most important options for `genmake2` are:

`--optfile=/PATH/FILENAME`

This specifies the "options file" that should be used for a particular build. The options file is a convenient and machine-indepenent way of specifying parameters such as the FORTRAN compiler (`FC=`), FORTRAN compiler optimization flags (`FFLAGS=`), and the locations of various platform- and/or machine-specific tools (eg. `MAKEDEPEND=`). As with `genmake2`, all options files should be written to be compatible with Bourne--shell ("sh" or "BASH v1") syntax. Examples of various options files can be found in `$ROOTDIR/tools/build_options`.

If no "optfile" is specified (either through the command lin or the environment variable), genmake2 will try to make a reasonable guess from the list provided in `$ROOTDIR/tools/build_options`. The method used for making this guess is to first determine the combination of operating system and hardware (eg. "linux_ia32") and then find a working Fortran compiler within the user's path. When these three items have been identified, genmake2 will try to find an optfile that has a matching name.

Everyone is encouraged to submit their options files to the MITgcm project for inclusion (please send to <`MITgcm-support@mitgcm.org`>). We are particularly grateful for options files tested on new or unique platforms!

`-adof=/path/to/file`
`-adoptfile=/path/to/file`

This option specifies the "adjoint" or automatic differentiation options file to be used. The file is analogous to the "optfile" defined above but it specifies information for the AD build process. The default file is located in `$ROOTDIR/tools/adjoint_options/adjoint_default` and it defines the "TAF" and "TAMC" compilers. An alternate version is also available at `$ROOTDIR/tools/adjoint_options/adjoint_staf` that selects the newer "STAF" compiler. As with any compilers, it is helpful to have their directories listed in your $PATH environment variable.

`-mods=DIR`
`-mods='DIR1 [DIR2 ...]'`

This option specifies a list of directories containing "modifications". These directories contain files with names that may (or may not) exist in the main MITgcm source tree but will be overridden by any identically-named sources within the "MODS" directories. The order of precedence for this "name-hiding" is as follows:

- "MODS" directories (in the order given)
- Packages either explicitly specified or provided by default (in the order given)
- Packages included due to package dependencies (in the order that that package dependencies are parsed)
- The "standard dirs" (which may have been specified by the "-standarddirs" option)

`-pgroups=/PATH/FILENAME`

This option specifies the file where package groups are defined. If not set, the package-groups definition will be read from `$ROOTDIR/pkg/pkg_groups`.

It also contains the default list of packages (defined as the group `"default_pkg_list"`) which is used when no specific package list (file: `packages.conf`) is found in current directory or in any "MODS" directory.

`-pdepend=/PATH/FILENAME`

This specifies the dependency file used for packages. If not specified, the default dependency file is `$ROOTDIR/pkg/pkg_depend`. The syntax for this file is parsed on a line-by-line basis where each line containes either a comment ("#") or a simple "PKGNAME1 (+|-)PKGNAME2" pairwise rule where the "+" or "-" symbol specifies a "must be used with" or a "must not be used with" relationship, respectively. If no rule is specified, then it is assumed that the two packages are compatible and will function either with or without each other.

`-make=/path/to/gmake`

Due to the poor handling of soft-links and other bugs common with the `make` versions provided by commercial Unix vendors, GNU `make` (sometimes called `gmake`) should be preferred. This option provides a means for specifying the make program to be used.

A successful run of `genmake2` will produce a `Makefile`, a `PACKAGES_CONFIG.h` file, and various convenience files used for the automatic differentiation process.

In general, it is best to use `genmake2` on a "clean" directory that is free of all source (*.[F,f],*.[F,f]90) and header (*.h,*.inc) files. Generally, this can be accomplished in an "un-clean" directory by running "make Clean" followed by "make makefile".

4.1.2. Using the `Makefile`

Once a `Makefile` has been created using `genmake2`, one can build a "standard" (forward simulator) executable using:

```
$ make Clean
$ make depend
$ make
```

The "make Clean" step will remove any stale source files, include files, and links. It is strongly recommended for "un-clean" directories which may contain the (perhaps partial) results of previous builds. Such "debris" can interfere with the next stage of the build. A more agressive cleaning option, "make CLEAN", can be used to also remove the executable and output files from a previous run.

The "make depend" step will create a large number of symbolic links from the local directory to the source file locations. It also parses these files and creates an extensive list of dependencies within the `Makefile` itself. The links that exist at this stage are mostly "large F" files (*.F and *.F90) that need to be processed by a C preprocessor ("CPP"). Since "make depend" edits the `Makefile`, it is important not to skip this step!

The final "make" invokes the C preprocessor to produce the "little f" files (*.f and *.f90) and then compiles them to object code using the specified FORTRAN compiler and options. An intermediate script is often used during this stage to further process (usually, make simple substitutions) custom definitions such as variable types within the source files. This additional stage is necessary in order to overcome some of the inconsistencies in the sizes of objects (bytes) between different compilers. The result of the build process is an executable with the name `mitgcmuv`.

In addition to the forward simulator described above, the Makefile also has a number of targets that can be used to produce various adjoint and tangent-linear builds for optimization and other parameter-sensitivity problems. The additional targets within the Makefile are:

make adall

This target produces an mitgcmuv_ad executable using the taf or staf adjoint compiler. See the genmake2 "-adof" option for compiler selection.

make ftlall

Similar to make adall above, this produces...

Please report any compilation failures or other build problems to the <MITgcm-support@mitgcm.org> list.

4.2. The Verification Suite

The MITgcm CVS tree (within the $ROOTDIR/verification/ directory) includes many (> 90) examples intended for regression testing. Each one of these test-experiment directories contains "known-good" output files along with all the input (including both code and data files) required for their re-calculation. Also included in $ROOTDIR/verification/ is the shell script testreport to perform regression tests.

4.2.1. Test-experiment Directory Content

Each test-experiment directory (TESTDIR) contains several standard subdirectories and files which testreport recognizes and uses when running a regression test. The directories/files that testreport uses are different for a forward test and an adjoint test (testreport -adm) and some test-experiments are set-up for only one type of regression test whereas others allow both types of tests (forward and adjoint).

Also some test-experiment allows, using the same MITgcm executable, to perform multiple tests using different parameters and input files, with a primary input set-up (input/ or input_ad/) and corresponding results (results/output.txt or results/output_adm.txt) and with one or several secondary inputs (input.OTHER/ or input_ad.OTHER/) and corresponding results (results/output.OTHER.txt or results/output_adm.OTHER.txt).

directory TESTDIR/results/

contains reference standard output used for test comparison. results/output.txt and results/output_adm.txt correspond respectively to primary forward and adjoint test run on the reference platform (currently baudelaire.csail.mit.edu) on one processor (no MPI, single thread) using the reference compiler (curently the GNU fortran compiler gfortran). The presence of these files determines whether or not testreport is testing or skipping this test-experiment. Reference standard output for secondary tests (results/output.OTHER.txt or results/output_adm.OTHER.txt) are also expected here.

The test comparison involves few model variables output, which are, by default and for a forward test, the 2-D solver initial residual (cg2d_init_res) and 3-D state variables (T,S,U,V) monitor output, and, by default and for an adjoint test, the cost-function and gradient-check. However, some test-experiments use some package-specific variable/monitor output according to the file TESTDIR/input[_ad][.OTHER]/tr_checklist specification.

directory `TESTDIR/build/`

> initially empty directory where `testreport` will build the MITgcm executable for forward and adjoint test. It might contains an experiment specific `genmake_local` file (see Section 4.1.1).
>
> Note that the original `code[_ad]/SIZE.h_mpi` is not directly used as "SIZE.h" to build an MPI-executable ; instead, a local copy `build/SIZE.h.mpi` is derived from `code[_ad]/SIZE.h_mpi` by adjusting the number of processors (nPx,nPy) according to `NUMBER_OF_PROCS` (see Section 4.2.2, `testreport -MPI`) ; then it is linked to "SIZE.h" (`ln -s SIZE.h.mpi SIZE.h`) before building the MPI-executable.

directory `TESTDIR/code/`

> contains the test-experiment specific source code used to build the MITgcm executable (`mitgcmuv`) for forward-test (using `genmake2 -mods=../code`).
>
> It can also contain specific source files with the suffix "`_mpi`" to be used in place of the corresponding file (without suffix) for an MPI test (see Section 4.2.2). The presence or absence of `SIZE.h_mpi` determines whether or not an MPI test on this test-experiment is performed or skipped.

directory `TESTDIR/code_ad/`

> contains the test-experiment specific source code used to build the MITgcm executable (`mitgcmuv_ad`) for adjoint-test (using `genmake2 -mods=../code_ad`). It can also contain specific source files with the suffix "`_mpi`" (see above).

directory `TESTDIR/input/`

> contains the input and parameter files used to run the primary forward test of this test-experiment.
>
> It can also contain specific parameter files with the suffix "`.mpi`" to be used in place of the corresponding file (without suffix) for MPI test, or with suffix "`.mth`" to be used for multi-threaded test (see Section 4.2.2). The presence or absence of `eedata.mth` determines whether or not a multi-threaded test on this test-experiment is performed or skipped.
>
> To save disk space and reduce downloading time, multiple copies of the same input file is avoided by using a shell script `prepare_run`. When such a script is found in `TESTDIR/input/` , `testreport` run this script in directory `TESTDIR/run/` after linking all the input file from `TESTDIR/input/` .

directory `TESTDIR/input_ad/`

> contains the input and parameter files used to run the primary adjoint test of this test-experiment. It can also contain specific parameter files with the suffix "`.mpi`" and shell script `prepare_run` as described above.

directory `TESTDIR/input.OTHER/`

> contains the input and parameter files used to run the secondary OTHER forward test of this test-experiment. It can also contain specific parameter files with suffix "`.mpi`" or "`.mth`" and shell script `prepare_run` (see above).
>
> The presence or absence the file `eedata.mth` determines whether or not a secondary multi-threaded test on this test-experiment is performed or skipped.

directory `TESTDIR/input_ad.OTHER/`

> contains the input and parameter files used to run the secondary OTHER adjoint test of this test-experiment. It can also contain specific parameter files with the suffix "`.mpi`" and shell script `prepare_run` (see above).

directory `TESTDIR/run/`

initially empty directory where `testreport` will run the MITgcm executable for primary forward and adjoint test.

Symbolic links (using command "`ln -s`") are made for input and parameter files (from `../input/` or from `../input_ad/`) and for MITgcm executable (from `../build/`) before the run proceeds. The sequence of links (function `linkdata` within shell script `testreport`) for a forward test is:

```
* link+rename or remove links
      to special files with suffix ".mpi" or ".mth" from ../input/
* link files from ../input/
* execute ../input/prepare_run (if it exists)
```

The sequence for an ajoint test is similar, with `../input_ad/` replacing `../input/` .

directory `TESTDIR/tr_run.OTHER/`

directory created by `testreport` to run the MITgcm executable for secondary "OTHER" forward or adjoint test.

The sequence of links for a forward secondary test is:

```
* link+rename or remove links
      to special files with suffix ".mpi" or ".mth" from ../input.OTHER/
* link files from ../input.OTHER/
* execute ../input.OTHER/prepare_run (if it exists)
* link files from ../input/
* execute ../input/prepare_run (if it exists)
```

The sequence for an ajoint test is similar, with `../input_ad.OTHER/` and `../input_ad/` replacing `../input.OTHER/` and `../input/` .

4.2.2. The `testreport` Utility

The shell script `testreport` (in `$ROOTDIR/verification/`), which was written to work with `genmake2`, can be used to build different versions of the MITgcm code, run the various examples, compare the output, and (if specified) email the results of each one of these tests to a central repository.

On some systems, the testreport script can be run with a command line as simple as:

```
$ cd verification
$ ./testreport
```

However, some systems (those lacking or wiht a broken "/bin/sh") may require an explicit shell invocation such as:

```
$ sh ./testreport -t 'exp2 exp4'
$ /some/path/to/bash ./testreport -t 'ideal_2D_oce lab_sea natl_box'
```

The `testreport` script accepts a number of command-line options which can be listed using the `-help` option. The most important ones are:

`-ieee` (default) / `-noieee`

If allowed by the compiler (as defined in the "optfile"), use IEEE arithmetic (`genmake2 -ieee`). This option, along with the gfortran / gcc compiler, is how the standard results are produced.

`-optfile=/PATH/FILENAME`
`-optfile '/PATH/F1 [/PATH/F2 ...]'`

This specifies a list of "options files" that will be passed to `genmake2`. If multiple options files are used (say, to test different compilers or different sets of options for the same compiler), then each options file will be used with each of the test directories.

`-tdir TESTDIR`
`-tdir 'TDIR1 TDIR2 [...]'`

This option specifies the test directory or list of test directories that should be used. Each of these entries should exactly (note: they are case sensitive!) match the names of directories in `$ROOTDIR/verification/`. If this option is omitted, then all directories that are properly formatted (that is, containing an `input` sub-directory and a `results/output.txt` file) will be used.

`-addr EMAIL`
`-addr 'EMAIL1 EMAIL2 [...]'`

Send the results (namely, `output.txt`, `genmake_local`, `genmake_state`, and `Makefile`) to the specified email addresses. The results are gzipped, placed in a tar file, MIME encoded, and sent to the specified address. If no email addresses are specified, no mail is sent.

`-MPI NUMBER_OF_PROCS`
`-mpi`

If the necessary file (`TESTDIR/code/SIZE.h_mpi`) exists, then use it (and all `TESTDIR/code/*_mpi` files) for an MPI--enabled run. The new option (`-MPI` followed by the maximum number of processors) enable to build and run each test-experiment using variable number of MPI processors (multiple of `nPx*nPy` from `TESTDIR/code/SIZE.h_mpi` and not larger than `NUMBER_OF_PROCS`). The short option ("-mpi") can only be used to build and run on 2 MPI processors (equivalent to "`-MPI 2`").

Note that the use of MPI typically requires a special command option (see "-command" below) to invoke the MPI executable. Examples of PBS scripts using testreport with MPI can be found in the tools/example_scripts directory (http://mitgcm.org/viewvc/MITgcm/MITgcm/tools/example_scripts/).

`-command='some command to run'`

For some tests, particularly MPI runs, a specific command might be needed to run the executable. This option allows a more general command (or shell script) to be invoked. Examples of PBS scripts using testreport with MPI can be found in the tools/example_scripts directory (http://mitgcm.org/viewvc/MITgcm/MITgcm/tools/example_scripts/).

For the case where the number of MPI processors varies according to each test-experiment, some key-words within the command-to-run argument will be replaced by their effective value:

`TR_NPROC` will be replaced by the actual number of MPI processors needed to run the current test-experiment.

`TR_MFILE` will be replaced by the name of local-file that testreport creates from the full list of machines which "`testreport -mf MACHINE_FILE`" provides, but truncated to the exact number of machines.

`-mf MACHINE_FILE`

To use with `-MPI NUMBER_OF_PROCS` option, to specify the file containing the full list of `NUMBER_OF_PROCS` machines to use for the MPI runs.

`-mth`

compile (with `genmake2 -omp`) and run with multiple threads (using eedata.mth).

The `testreport` script will create an output directory `tr_NAME_DATE_N/` , with `hostname` as default `NAME`, `DATE` the current date followed by a suffix number "N" to distinguish from previous `testreport` output directories. `testreport` writes progress to the screen (stdout) and reports into the ouput directory as it runs. In particular, one can find, in the ouput directory, the `summary.txt` file that contains a brief comparison of the current output with the "known-good" output. At the end of the testing process, the `tr_out.txt` file is generated in `$ROOTDIR/verification/` as a compact version of `summry.txt` file.

4.2.3. The `do_tst_2+2` Utility

The shell script `do_tst_2+2` (in `$ROOTDIR/tools/`) can be used to check the accuracy of the restart procedure.

4.3. Creating MITgcm Packages

Optional parts of code have been separated from the MITgcmUV core driver code and organised into packages. The packaging structure provides a mechanism for maintaining suites of code, specific to particular classes of problems, in a way that is cleanly separated from the generic fluid dynamical engine.

The MITgcmUV packaging structure is described below using generic package names ${pkg}. A concrete examples of a package is the code for implementing GM/Redi mixing. This code uses the package name

5. Chris's Notes...

```
           MITgcmUV Packages
           =================

  Optional parts of code are separated from
the MITgcmUV core driver code and organised into
packages. The packaging structure provides a mechanism for
maintaining suites of code, specific to particular
classes of problem, in a way that is cleanly
separated from the generic fluid dynamical engine.

 The MITgcmUV packaging structure is describe
below using generic package names ${pkg}.
A concrete examples of a package is the code
for implementing GM/Redi mixing. This code uses
the package name
*   ${PKG} = GMREDI
*   ${pkg} = gmredi
*   ${Pkg} = gmRedi

Package states
==============
```

```
 Packages can be any one of four states, included,
 excluded, enabled, disabled as follows:

 included(excluded) compile time state which
                    includes(excludes) package
                    code and routine calls from
                    compilation/linking etc...

 enabled(disabled)  run-time state which
                    enables(disables) package code
                    execution.

 Every call to a ${pkg}_... routine from outside the package
 should be placed within both a
 #ifdef ALLOW_${PKG} ... block and a
 if ( use${Pkg} ) ... then block.
 Package states are generally not expected to change during
 a model run.

Package structure
=================

o  Each package gets its runtime configuration
   parameters from a file named "data.${pkg}"
   Package runtime config. options are imported
   into a common block held in a header file
   called "${PKG}.h".
   Note: In some packages, the header file "${PKG}.h" is splitted
   into "${PKG}_PARAMS.h" that contains the package parameters and
   ${PKG}_VARS.h" for the field arrays.

o  The core driver part of the model can check
   for runtime enabling or disabling of individual packages
   through logical flags use${Pkg}.
   The information is loaded from a
   global package setup file called "data.pkg".
   The use${Pkg} flags are not used within
   individual packages.

o  Included in "${PKG}.h" is a logical flag
   called ${Pkg}IsOn. The "${PKG}.h" header file can be imported
   by other packages to check dependencies and requirements
   from other packages ( see "Package Boot Sequence" section).
   NOTE: This procedure is not presently implemented,
   ----- neither for kpp nor for gmRedi.

CPP Flags
=========

    1. Within the core driver code flags of the form
       ALLOW_${PKG} are used to include or exclude
       whole packages. The ALLOW_${PKG} flags are included
```

```
   from a PACKAGES_CONFIG.h file that is automatically
   generated by genmake2 (see genmake2 section).
   held in-line in the CPP_OPTIONS.h header file.
   e.g.

   Core model code .....

   #include "PACKAGES_CONFIG.h"
   #include "CPP_OPTIONS.h"
     :
     :
     :

   #ifdef ALLOW_${PKG}
     if ( use${Pkg} ) CALL ${PKG}_DO_SOMETHING(...)
   #endif

2. Within an individual package a header file,
   "${PKG}_OPTIONS.h", is used to set CPP flags
   specific to that package. It also includes
   "PACKAGES_CONFIG.h" and "CPP_OPTIONS.h".
```

```
Package Boot Sequence
=====================

    Calls to package routines within the core code timestepping
    loop can vary. However, all packages follow a required
    "boot" sequence outlined here:

    1. S/R PACKAGES_BOOT()
            :
        CALL OPEN_COPY_DATA_FILE( 'data.pkg', 'PACKAGES_BOOT', ... )

    2. S/R PACKAGES_READPARMS()
            :
        #ifdef ALLOW_${PKG}
          if ( use${Pkg} )
     &       CALL ${PKG}_READPARMS( retCode )
        #endif

    3. S/R PACKAGES_INIT_FIXED()
            :
        #ifdef ALLOW_${PKG}
          if ( use${Pkg} )
     &       CALL ${PKG}_INIT_FIXED( retCode )
        #endif

    4. S/R PACKAGES_CHECK()
            :
        #ifdef ALLOW_${PKG}
          if ( use${Pkg} )
     &       CALL ${PKG}_CHECK( retCode )
        #else
```

```
          if ( use${Pkg} )
     &       CALL PACKAGES_CHECK_ERROR('${PKG}')
        #endif

    5. S/R PACKAGES_INIT_VARIABLES()
             :
        #ifdef ALLOW_${PKG}
          if ( use${Pkg} )
     &       CALL ${PKG}_INIT_VARIA( )
        #endif

Package Output
==============
     6. S/R DO_THE_MODEL_IO

        #ifdef ALLOW_${PKG}
          if ( use${Pkg} )
     &       CALL ${PKG}_OUTPUT( )
        #endif

     7. S/R PACKAGES_WRITE_PICKUP()

        #ifdef ALLOW_${PKG}
          if ( use${Pkg} )
     &       CALL ${PKG}_WRITE_PICKUP( )
        #endif

Description
===========

      - ${PKG}_READPARMS()
    is responsible for reading
    in the package parameters file data.${pkg}, and storing
    the package parameters in "${PKG}.h" (or in "${PKG}_PARAMS.h").
    -> called from INITIALISE_FIXED in PACKAGES_READPARMS

     - ${PKG}_INIT_FIXED()
    is responsible for completing the internal setup of a package.
    -> called from INITIALISE_FIXED in PACKAGES_INIT_FIXED
    note: 1) some pkg use instead:
             CALL ${PKG}_INITIALISE  ( or the old form CALL ${PKG}_INIT )
          2) for simple pkg setup, this part is done inside ${PKG}_READPARMS

     - ${PKG}_CHECK()
    is responsible for validating
    basic package setup and inter-package dependencies.
    ${PKG}_CHECK can import other package parameters it may
    need to check. This is done through header files "${PKG}.h".
    It is assumed that parameters owned by other packages
    will not be reset during ${PKG}_CHECK().
    -> called from INITIALISE_FIXED in PACKAGES_CHECK

     - ${PKG}_INIT_VARIA()
```

```
    is responsible for fill-in all package variables with an initial value.
    Contains eventually a call to ${PKG}_READ_PICKUP that will read
    from a pickup file the package variables required to restart the model.
    This routine is called after the core model state has been completely
    initialised but before the core model timestepping starts.
    -> called from INITIALISE_VARIA in PACKAGES_INIT_VARIABLES
    note: the name ${PKG}_INIT_VARIA is not yet standard and some pkg
     use for e.g. ${PKG}_INI_VARS, ${PKG}_INIT_VARIABLES, or the old
     form ${PKG}_INIT

     - ${PKG}_OUTPUT( )
     is responsible for writing time-average fields to output files
     (but the cumulating step is done within the package main S/R).
     Can also contain other diagnostics (.e.g. CALL ${PKG}_MONITOR)
     and write snap-shot fields that are hold in common blocks. Other
     temporary fields are directly dump to file where they are available.
     NOTE: 1) the S/R old name ${PKG}_DIAGS is used in some packages
              but is beeing replaced by ${PKG}_OUTPUT
              to avoid confusion with pkg/diagnostics functionality.
           2) the output part is not yet in a standard form and might still
              evolve a lot.
    -> called within DO_THE_MODEL_IO

     - ${PKG}_WRITE_PICKUP()
     is responsible for writing a package pickup file when necessary for
     a restart. (found also the old name: ${PKG}_WRITE_CHECKPOINT )
    -> called from FORWARD_STEP and THE_MODEL_MAIN in PACKAGES_WRITE_PICKUP

Summary
=======

- CPP options:
  -----------------------
  * ALLOW_${PKG}          include/exclude package for compilation

- FORTRAN logical:
  -----------------------
  * use${Pkg}             enable package for execution at runtime
                          -> declared in PARAMS.h
  * ${Pkg}IsOn            for package cross-dependency check
                          -> declared in ${PKG}.h
                          N.B.: Not presently used!

- header files
  -----------------------
  * ${PKG}_OPTIONS.h      has further package-specific CPP options
  * ${PKG}.h              package-specific common block variables, fields
   or  ${PKG}_PARAMS.h    package-specific common block parameters
   and ${PKG}_VARS.h      package-specific common block fields

- FORTRAN source files
  -----------------------
  * ${pkg}_readparms.F     reads parameters from file data.${pkg}
```

```
  * ${pkg}_init_fixed.F   complete the package setup
  * ${pkg}_check.F        checks package dependencies and consistencies
  * ${pkg}_init_varia.F   initialises package-related fields
  * ${pkg}_... .F         package source code
  * ${pkg}_output.F       write output to file.
  * ${pkg}_write_pickup.F write a package pickup file to restart the model

  New: Subroutine in one package (pkgA) that only contains code which
       is connected to a 2nd package (pkgB) (e.g.: gmredi_diagnostics_init.F)
       will be named: pkgA_pkgB_something.F

- parameter file
  -----------------------
  * data.${pkg}          parameter file
```

6. Editing the Documentation

6.1. Getting the Docs and Code

The first step towards editing the documentation is to checkout a copy of code, docs, and build scripts from the CVS server using:

```
$ export CVS_RSH=ssh
$ export CVSROOT=':ext:NAME@mitgcm.org:/u/gcmpack'
$ mkdir scratch
$ cvs co -P MITgcm manual mitgcm.org
```

These commands extract the necessary information from the CVS server and create a temporary (called `scratch`) directory for the storage of the HTML and other files that will be created. Please note that you must either create `scratch` as shown or edit the various `Makefile`s and scripts used to create the documentation.

6.2. Editing the Documentation

The documentation is contained in the `manual` directory in a raw LaTeX format. The main document is `manual.tex` and it uses **\input{}**s to include the chapters and subsections.

Since the same LaTeX source is used to produce PostScript, PDF, and HTML output, care should be taken to follow certain conventions. Two of the most important are the usage of the **\filelink{}{}** and **\varlink{}{}** commands. Both of these commands have been defined to simplify the connection between the automatically generated ("code browser") HTML and the HTML version of the manual produced by LaTeX2HTML. They each take two arguments (corresponding to the contents of the two sets of curly braces) which are the text that the author wishes to be "wrapped" within the link, and a specially formatted link thats relative to the `MITgcm` directory within the CVS tree.

The result is a command that resembles either

1. a reference to a variable or subroutine name such as **\varlink{tRef}{tRef}**, or

2. a reference to a file such as **\varlink{tRef}{path-to-the-file_name.F}** where the absolute path to the file is of the form `/foo/MITgcm/path/to/the/file_name.F`

   (please note how the leading "/foo/MITgcm" component of the path is dropped leaving the path *relative* to the head of the code directory and each directory separator "/" is turned into a "-")

6.3. Building the Documentation

Given the directory structure of Section 6.1, the entire documentation for the web site can be built using:

```
$ cd mitgcm.org/devel/buildweb
$ make All
```

Which builds the PDF from the LaTeX source, creates the HTML output from the LaTeX source, parses the FORTRAN code base to produce a hyperlinked HTML version of the source, and then determines the cross-linking between the various HTML components.

If there are no errors, the result of the build process (which can take 30+ minutes on a P4/2.5Ghz) will be contained within a single directory called `scratch/dev_docs`. This is a freshly built version of the entire on-line users manual. If you have the correct permissions, it can be directly copied to the web server area:

```
$ mv scratch/dev_docs /u/u0/httpd/html
```

and the update is complete.