## Contents:

1. **NEMO compilation notes**  
   1.1 NEMO 3.6 (stable) + XIOS 1.0 ........................................ 3  
   1.2 NEMO 3.7/4.0 + XIOS 2.0 ........................................ 9  
   1.3 NEMO 4.0 (beta) + XIOS 2.5 ........................................ 14  
   1.4 Other packages .................................................. 19  

2. **Other NEMO notes**  
   2.1 Adding code to NEMO ................................................ 27  
   2.2 Other NEMO packages ............................................. 27  
   2.3 GYRE: rotated gyre model ........................................ 27  
   2.4 ORCA: global configuration ....................................... 27  
   2.5 UNAGI: custom channel model ................................... 28  
   2.6 pyCDFTOOLS ..................................................... 30  

3. **GEOMETRIC outline**  
   3.1 Advection ....................................................... 34  
   3.2 Source .......................................................... 34  
   3.3 Dissipation ..................................................... 34  
   3.4 Diffusion ....................................................... 34  
   3.5 NEMO implementation ........................................... 35  

4. **Misc. content**  
   4.1 Python / Anaconda notes ........................................ 39  
   4.2 sphinx notes .................................................... 42  
   4.3 git (or bitbucket) commands .................................... 42  

5. **Key updates:**  

Bibliography ......................................................... 45
Homebrew collection of NEMO (mostly post v3.6) related content, existing here primarily to remind myself of code details. Includes compilation notes, analyses codes, and eventually details to some GEOMETRIC details, its NEMO implementation and model building in here too. Appropriate disclaimers in the individual sections themselves.

I can be contacted at julian.mak@physics.ox.ac.uk regarding the modifying / adding to the content here.
Personally I prefer doing small bits of code testing on a small configuration (normally GYRE in NEMO) so I have tried to get NEMO working on a local machine, largely following the instructions from the NEMO forge; please consult that page first as the information on there should be considered the authoritative version.

While it is fairly straightforward on a supported cluster/supercomputer (e.g. first try following NOCL ARCHER guide) it can be a bit temperamental on a local machine to do with compiler compatibilities. The following notes are what I did to get XIOS and NEMO compiling and running, and will display commands with gcc4.9 compilers (which is my default for other reasons). Extra things that need to be modified for other compilers I have tested will be given accordingly (see the top of the individual pages as to which compilers I have tested the notes with).

I added the following to my ~/.bashrc:

```
export CC=/usr/bin/gcc-4.9
export CXX=/usr/bin/g++-4.9
export FC=/usr/bin/gfortran-4.9
export F77=/usr/bin/gfortran-4.9
export CPP=/usr/bin/cpp-4.9
```

which overrides the default gcc5.4 on my computer.

### 1.1 NEMO 3.6 (stable) + XIOS 1.0

Tested with

- gcc4.9, gcc5.4 on a laptop (Ubuntu 16.04)
- gcc4.9 on a modular system (Ubuntu 14.04, Oxford AOPP)
- gcc4.8 on a Mac (El Capitan OSX 10.11)

The assumption here is that the compiler is fixed and the packages (e.g., NetCDF4 and a MPI bindings) are configured to be consistent with the compilers. See here to check whether the binaries exist, where they are, and how they might be installed. All the #CHANGE ME highlighted below needs be modified to point to the appropriate paths or binaries (links for binaries are ok).
The instructions below assumes gcc4.9 compilers but works for gcc4.8 and gcc5.4 compilers too (with one extra flag required in XIOS compilation for the latter). I defined some extra variables on a Linux machine:

```bash
export $BD=/home/julian/testing/gcc4.9-builds # CHANGE ME
export C_INCLUDE_PATH=$BD/install/include:$C_INCLUDE_PATH
export CPLUS_INCLUDE_PATH=$BD/install/include:$CPLUS_INCLUDE_PATH
export LIBRARY_PATH=$BD/install/lib:$LIBRARY_PATH
export LD_LIBRARY_PATH=$BD/install/lib:$LD_LIBRARY_PATH
```

Otherwise I have found the resulting libraries and binaries are not necessarily linked to the right ones (I have a few versions of libraries at different places as a result of the testing recorded here). You shouldn’t need to do the above if the packages are forced to look at the right place, though the above may help.

On a Mac done through anaconda the above was not necessary. My understanding is that setting these variables might not actually do anything unless an option is specifically enabled in Xcode.

### 1.1.1 XIOS 1.0 (svn v703)

To use NEMO you probably do need XIOS to do the I/O. The instructions here follow the one given in the XIOS instructions with any errors that arise. A useful site to search for XIOS related errors may be found on the XIOS user mailing list.

Here XIOS1.0 is used with NEMO3.6 for compatibility reasons. For the purposes here I created a folder called XIOS and used `svn` to get XIOS1.0 (which is going to be XIOS/xios1.0):

```bash
mkdir XIOS
cd XIOS
svn co http://forge.ipsl.jussieu.fr/ioserver/svn/XIOS/branchs/xios-1.0@703
```

To get XIOS to compile, the compilers and packages need to be pointed to first, via modifying files in arch. Since I am using gcc, I did the following just to make a fresh copy:

```bash
cd xios1.0/arch
cp arch-GCC_LINUX.env arch-GCC_local.env
cp arch-GCC_LINUX.fcm arch-GCC_local.fcm
cp arch-GCC_LINUX.path arch-GCC_local.path
```

The *.env file specifies where HDF5 and NetCDF4 binaries live. The *.fcm file specifies which compilers and options to use. The *.path file specifies which paths and options to include. My files look like the following:

```bash
# arch-GCC_local.env
export HDF5_INC_DIR=/usr/local/include # CHANGE ME
export HDF5_LIB_DIR=/usr/local/lib # CHANGE ME
export NETCDF_INC_DIR=/usr/local/include # CHANGE ME
export NETCDF_LIB_DIR=/usr/local/lib # CHANGE ME
```

You could check where the HDF5 and NetCDF4 directories are by doing `which h5copy` and `which nc-config`, which should give you a directory/bin, and it is the directory part you want. If you did install the libraries somewhere else as in other packages, say, then make sure the which commands are pointing to the right place.

```bash
# arch-GCC_local.fcm
```

(continues on next page)
Check the MPI locations by doing `which mpicc` and `mpicc --version` say. If they are the right ones you could just have `mpicc` instead of the full path as given above. MPI bindings are used here to avoid a possible error that may pop up in relation to the build trying to find `mpi.h`. The `gmake` command was swapped out by the `make` command (I don’t have `cmake`).

**Note:** For gcc5.4 and maybe newer versions, doing just the above when compiling leads to a whole load of errors about clashing in C++:

```plaintext
.../include/boost/functional/hash/extensions.hpp:69:33: error: ‘template<class T, class A> std::size_t boost::hash_value’ conflicts with a previous declaration
  std::size_t hash_value(std::list<T, A> const& v)
```

Adding `-D_GLIBCXX_USE_CXX11_ABI=0` to `%BASE_CFLAGS` fixes these.

```plaintext
# arch-GCC_local.path

NETCDF_INCDIR="-I$NETCDF_INC_DIR"
NETCDF_LIBDIR="-Wl,'-allow-multiple-definition' -L$NETCDF_LIB_DIR"
NETCDF_LIB="-lnetcdff -lnetcdf"

MPI_INCDIR=""
MPI_LIBDIR=""
MPI_LIB=""

HDF5_INCDIR="-I$HDF5_INC_DIR"
HDF5_LIBDIR="-L$HDF5_LIB_DIR"
HDF5_LIB="-lhdf5_hl -lhdf5 -lhdf5 -lz"
```

The above has all the OASIS (the atmosphere / ocean coupler) keys removed. I added the `-Wl, '-allow-multiple-definition' key for reasons I don’t remember anymore...
Now it should be ready to compile. Assuming the current directory is `xios1.0/arch`:

```bash
cd ..;
./make_xios --full --prod --arch GCC_local -j2 |& tee compile_log.txt
```

The `-j2` option uses two processors to build. The `tee` command is to keep logs of potential errors (the `|&` is short for `2>&1 |`) for debugging the compiler issues that may arise.

**Note:** If you get

```
/home/julian/testing/nemo-6800/xios-703/xios-1.0/inc/netcdf.hpp:20:26: fatal error:
  → netcdf_par.h: No such file or directory
#include <netcdf_par.h>
^
```

then it is probably because NetCDF4 was not built as parallel. There is actually a copy of the file in `.extern/src_netcdf4/netcdf_par.h`, and it could be pointed to by looking into `bld.cfg`:

```
bld::tool::cflags %CFLAGS %CBASE_INC -I${PWD}/extern/src_netcdf -I${PWD}/extern/
  → boost/include -I${PWD}/extern/rapidxml/include -I${PWD}/extern/blitz/include
```

where `src_netcdf` should be changed to `src_netcdf4`.

**Note:** If you get something like

```
libhdf5.a(H5PL.o): undefined reference to symbol 'dlclose@GLIBC_2.2.5'
```

then this suggests that the HDF5 library that is being called is built as a static and/or not shareable library. In this case adding the `-ldl` flag to `HDF5_LIB` in `arch-GCC_local.path` should work. Or if you want to you can recompile HDF5 as a shareable library; see other packages on how you might go about doing this.

It should work and takes around 5 mins to compile for me. The main end result is are binaries in `xios1.0/bin/` which NEMO will call.

**Note:** Do `ldd bin/xios_server.exe` (or wherever `xios_server.exe` lives) to make sure the libraries linked to it are the intended libraries. XIOS may still work if the NetCDF versions are ok, but if not, go back and define `LD_LIBRARY_PATH` and other variables accordingly; see above.

`xios_server.exe` is one of the other binaries built from compiling but is not required for small runs on a laptop. For its use on a cluster see for example the instructions on the NOCL ARCHER guide.

### 1.1.2 NEMO 3.6 (svn v6800)

Check out a version of NEMO. I have another folder separate to the XIOS folders to contain the NEMO codes and binaries:
mkdir NEMO

```
cd NEMO
svn co http://forge.ipsl.jussieu.fr/nemo/svn/NEMO/trunk@6800 nemo3.6-6800
```

This checks out version 6800 (NEMO 3.6) and dumps it into a folder called nemo3.6-6800 (change the target path to whatever you like). A similar procedure to specify compilers and where XIOS lives needs to be done for NEMO. Again, because I of the compilers I am using:

```
cd nemo3.6-6800/NEMOGCM/ARCH
cp OLD/gfortran_linux.fcm ./gfortran_local.fcm
```

None of the fcm files associated with gfortran actually worked for me out of the box so here is my build of it (click HERE for a detailed log of how I got to the following):

```
# gfortran_local.fcm

# generic gfortran compiler options for linux
# NCDF_INC netcdf include file
# NCDF_LIB netcdf library
# FC Fortran compiler command
# FCFLAGS Fortran compiler flags
# FFLAGS Fortran 77 compiler flags
# LD linker
# LDFLAGS linker flags, e.g. -L<lib dir> if you have libraries in a
# FPPFLAGS pre-processing flags
# AR assembler
# ARFLAGS assembler flags
# MK make
# USER_INC additional include files for the compiler, e.g. -I<include dir>
# USER_LIB additional libraries to pass to the linker, e.g. -l<library>

%NCDF_HOME /usr/local       # CHANGE ME
%XIOS_HOME /home/julian/testing/gcc4.9-builds/XIOS/xios-1.0 # CHANGE ME
%CPP      cpp-4.9            # CHANGE ME
%CPPFLAGS -P -traditional
%XIOS_INC -I%XIOS_HOME/inc
%XIOS_LIB -L%XIOS_HOME/lib -lxios
%NCDF_INC -I%NCDF_HOME/include
%NCDF_LIB -L%NCDF_HOME/lib -lnetcdf -lnetcdff -lstdc++
%FC        mpif90            # CHANGE ME
%FCFLAGS   -fdefault-real-8 -O3 -funroll-all-loops -fcray-pointer -cpp -ffree-line-length-none
%FFLAGS    %FCFLAGS
%LD        %FC
%LDFLAGS   -P -C -traditional
%AR         ar
%ARFLAGS    -rs
%MK         make
%USER_INC   %XIOS_INC %NCDF_INC
%USER_LIB   %XIOS_LIB %NCDF_LIB
```

The main changes are (again, see here for an attempt at the reasoning and a log of errors that motivates the changes):

1.1. NEMO 3.6 (stable) + XIOS 1.0
J Mak NEMO notes

- added `%NCDF_HOME` to point to where NetCDF lives
- added `%XIOS_*` keys to point to where XIOS lives
- added `%CPP` and flags, consistent with using gcc4.9
- added the `-lnetcdff` and `-lstdc++` flags to NetCDF flags
- using mpif90 which is a MPI binding of gfortran-4.9
- added -cpp and -ffree-line-length-none to Fortran flags
- swapped out `gmake` with `make`

Note: It might be worthwhile doing the following first:

```
cd ../CONFIG/
./makenemo -j0 -r GYRE -n GYRE_testing -m gfortran_local
```

Then, edit add `key_nosignedzero` to the end of `/GYRE_testing/cpp_GYRE_testing.fcm` (see note at the bottom of the page). `-j0` does all the folder creation and copying but doesn’t do the compile step.

To compile a configuration (using the GYRE config):

```
cd ../CONFIG/
./makenemo -j2 -r GYRE -n GYRE_testing -m gfortran_local |& tee compile_log.txt
```

This uses two processors, with GYRE as a reference, builds a new folder called `GYRE_testing`, with the specified architecture file, and outputs a log.

Note: The -r GYRE flag here only needs to be done once to create an extra folder and add GYRE_testing to `cfg.txt`. The subsequent compilations should then read, e.g., `./makenemo -n GYRE_testing -m gfortran_local`.

Check that it does run with the following:

```
cd GYRE_testing/EXP00
mpiexec -n 1 ./opa
```

This may be `mpirun` instead of `mpiexec`, and `-n 1` just runs it as a single core process. Change `nn_itend = 4320` in `nn_itend = 120` to only run it for 10 days (rdt = 7200 which is 2 hours). With all the defaults as is, there should be some `GYRE_5d_*_.nc` data in the folder. You can read this with `ncview` (see the `ncview` page or, if you have `sudo` access, you can install it through `sudo apt-get install ncview`), bearing in mind that this is actually a rotated gyre configuration (see the following NEMO forge page or search for `gyre` in the NEMO book).

Note: My run actually crashed immediately. Looking into `ocean.output` and searching for `ERROR` shows that `key_nosignedzero` needed to be added to `/GYRE_testing/cpp_GYRE_testing.fcm`. Rebuilding with the key then works fine.

Note: If your installation compiles but does not run with the following error

```
dyld: Library not loaded: @rpath/libnetcdff.6.dylib
Referenced from: /paths/./nemo
```

(continues on next page)
then it is not finding the right libraries. These could be fixed by adding the \texttt{-Wl,-rpath,/fill me in/lib} flag to the relevant flags bit in the \texttt{*.fcm} (or possibly in XIOS the \texttt{path} and/or \texttt{env}) files (in this case it is NetCDF as it calls the \texttt{libnetcdf.f} library) specifying exactly where the libraries live. This can happen for example on a Mac or if the libraries are installed not at the usual place.

\textbf{Note:} One infuriating problem I had specifically with a Mac (though it might be a \texttt{gcc4.8} issue) is that the run does not get beyond the initialisation stage. Going into \texttt{ocean.output} and searching for \texttt{ERROR} shows that it could complain about a misspelled namelist item (in my case it was in the \texttt{namberg} namelist). If you go into \texttt{output.namelist.dyn} and look for the offending namelist is that it might be reading in nonsense. This may happen if the comment character \texttt{!} is right next to a variable, e.g.

\begin{verbatim}
ln_icebergs = .true.!this is a comment
\end{verbatim}

Fix this by adding a white space, i.e.

\begin{verbatim}
ln_icebergs = .true. !this is a comment
\end{verbatim}

which should fix it...

\section*{1.2 NEMO 3.7/4.0 + XIOS 2.0}

Tested with

\begin{itemize}
\item \texttt{gcc4.9, gcc5.4} on a laptop (Ubuntu 16.04)
\item \texttt{gcc4.9} on a modular system (Ubuntu 14.04, Oxford AOPP)
\item \texttt{gcc4.8} on a Mac (El Capitan OSX 10.11)
\end{itemize}

This is the version I first implemented GEOMETRIC in, which is a development version I guess (?) that eventually led to NEMO 4.0. The code structure largely follows NEMO 3.6 but the commands are slightly different.

If you get errors that are not documented here, see if the \texttt{XIOS1.0 NEMO3.6} page contains the relevant errors.

The assumption here is that the compiler is fixed and the packages (e.g., NetCDF4 and a MPI bindings) are configured to be consistent with the compilers. See \texttt{here} to check whether the binaries exist, where they are, and how they might be installed.

The instructions below assumes \texttt{gcc4.9} compilers but works for \texttt{gcc4.8} and \texttt{gcc5.4} compilers too (with one extra flag required in XIOS compilation for the latter). I defined some extra variables on a Linux machine:

\begin{verbatim}
export $BD=/home/julian/testing/gcc4.9-builds # CHANGE ME
export C_INCLUDE_PATH=$BD/install/include:$C_INCLUDE_PATH
export CPLUS_INCLUDE_PATH=$BD/install/include:$CPLUS_INCLUDE_PATH
export LIBRARY_PATH=$BD/install/lib:$LIBRARY_PATH
export LD_LIBRARY_PATH=$BD/install/lib:$LD_LIBRARY_PATH
\end{verbatim}

Otherwise I have found the resulting libraries and binaries are not necessarily linked to the right ones (I have a few versions of libraries at different places as a result of the testing recorded here). You shouldn’t need to do the above if the packages are forced to look at the right place, though the above may help.

\subsection*{1.2. NEMO 3.7/4.0 + XIOS 2.0}

9
On a Mac done through anaconda the above was not necessary. My understanding is that setting these variables might not actually do anything unless an option is specifically enabled in Xcode.

### 1.2.1 XIOS 2.0 (svn v1322)

Do the following:

```
mkdir XIOS
cd XIOS
svn co http://forge.ipsl.jussieu.fr/ioserver/svn/XIOS/trunk@1322 xios-2.0
```

**Note:** Turns out I initially took a version out from the trunk. Doing it from the branch as in `svn co http://forge.ipsl.jussieu.fr/ioserver/svn/XIOS/branches/xios-2.0 xios-2.0` seems to also work with instructions below.

To get XIOS to compile, the compilers and packages need to be pointed to first, via modifying files in arch. Since I am using gcc, I did the following just to make a fresh copy:

```
cd xios2.0/arch
cp arch-GCC_LINUX.env arch-GCC_local.env
cp arch-GCC_LINUX.fcm arch-GCC_local.fcm
cp arch-GCC_LINUX.path arch-GCC_local.path
```

The *.env file specifies where HDF5 and NetCDF4 binaries live. The *.fcm file specifies which compilers and options to use. The *.path file specifies which paths and options to include. My files look like the following:

```
# arch-GCC_local.env
export HDF5_INC_DIR=/usr/local/include # CHANGE ME
export HDF5_LIB_DIR=/usr/local/lib # CHANGE ME
export NETCDF_INC_DIR=/usr/local/include # CHANGE ME
export NETCDF_LIB_DIR=/usr/local/lib # CHANGE ME
```

You could check where the HDF5 and NetCDF4 directories are by doing `which h5copy` and `which nc-config`, which should give you a directory/bin, and it is the directory part you want. If you did install the libraries somewhere else as in other packages, say, then make sure the which commands are pointing to the right place.

```
# arch-GCC_local.fcm
```

(continues on next page)
%PROD_FFLAGS  -O3
%DEV_FFLAGS  -g -O2
%DEBUG_FFLAGS  -g
%BASE_INC  -D__NONE__
%BASE_LD  -lstdc++
%CPP  cpp-4.9 # CHANGE ME
%FPP  cpp-4.9 -P # CHANGE ME
%MAKE  make

Check the MPI locations by doing \texttt{which mpicc} and \texttt{mpicc --version} say. If they are the right ones you could just have \texttt{mpicc} instead of the full path as given above. MPI bindings are used here to avoid a possible error that may pop up in relation to the build trying to find \texttt{mpi.h}. The \texttt{gmake} command was swapped out by the \texttt{make} command (I don’t have \texttt{cmake}).

**Note:** For \texttt{gcc5.4} and maybe newer versions, doing just the above when compiling leads to a whole load of errors about clashing in C++:

```
.../include/boost/functional/hash/extensions.hpp:69:33: error: ‘template<class T,…
    class A> std::size_t boost::hash_value’ conflicts with a previous declaration
    std::size_t hash_value(std::list<T, A> const& v)
```

Adding \texttt{-D_GLIBCXX_USE_CXX11_ABI=0} to \texttt{%BASE_CFLAGS} fixes these.

```
# arch-GCC_local.path
NETCDF_INCDIR="-I$NETCDF_INC_DIR"
NETCDF_LIBDIR="-Wl,’--allow-multiple-definition’ -L$NETCDF_LIB_DIR"
NETCDF_LIB="-lncdf -lnetcdff"
MPI_INCDIR=""
MPI_LIBDIR=""
MPI_LIB=""
HDF5_INCDIR="-I$HDF5_INC_DIR"
HDF5_LIBDIR="-L$HDF5_LIB_DIR"
HDF5_LIB="-lhdf5_hl -lhdf5 -lhdf5 -lz"
```

The above has all the OASIS (the atmosphere / ocean coupler) keys removed. I added the \texttt{-Wl, ’--allow-multiple-definition’} key for reasons I don’t remember anymore...

I went into \texttt{bld.cfg}, found the line

```
bld::tool::cflags %CFLAGS %CBASE_INC -I$(PWD)/extern/src_netcdf -I$(PWD)/
    extern/boost/include -I$(PWD)/extern/rapidxml/include -I$(PWD)/extern/
    blitz/include
```

and changed \texttt{src_netcdf} to \texttt{src_netcdf4} (see \texttt{XIOS1.0 stuff} for the reason).

Now it should be ready to compile. Assuming the current directory is \texttt{xios2.0/arch}:

```
cd ..
./make_xios --full --prod --arch GCC_local -j2 |& tee compile_log.txt
```
The \texttt{-j2} option uses two processors to build. The \texttt{tee} command is to keep logs of potential errors (the \texttt{\&} is short for \texttt{2>&1}) for debugging the compiler issues that may arise. It should work and takes around 5 mins to compile for me. The main end result is are binaries in \texttt{xios2.0/bin/} which NEMO will call.

### 1.2.2 NEMO 3.7/4.0 (svn v8666)

Check out a version of NEMO. I have another folder separate to the XIOS folders to contain the NEMO codes and binaries:

```bash
mkdir NEMO
cd NEMO
svn co http://forge.ipsl.jussieu.fr/nemo/svn/NEMO/trunk@8666 nemo3.7-8666
```

This checks out version 8666 (NEMO 3.7/4.0) and dumps it into a folder called \texttt{nemo3.7-8666} (change the target path to whatever you like). A similar procedure to specify compilers and where XIOS lives needs to be done for NEMO. Again, because I of the compilers I am using:

```bash
cd nemo3.7-8666/NEMOGCM/ARCH
cp OLD/gfortran_linux.fcm ./gfortran_local.fcm
```

None of the fcm files associated with gfortran actually worked for me out of the box so here is my build of it (click \texttt{HERE} for a detailed log of how I got to the following):

```bash
# gfortran_local.fcm
# generic gfortran compiler options for linux
# NCDF_INC netcdf include file
# NCDF_LIB netcdf library
# FC Fortran compiler command
# FCFLAGS Fortran compiler flags
# FFLAGS Fortran 77 compiler flags
# LD linker
# LDFLAGS linker flags, e.g. -L<lib dir> if you have libraries in a
# FPFLAGS pre-processing flags
# AR assembler
# ARFLAGS assembler flags
# MK make
# USER_INC additional include files for the compiler, e.g. -I<include dir>
# USER_LIB additional libraries to pass to the linker, e.g. -l<library>

%NCDF_HOME /usr/local # CHANGE ME
%XIOS_HOME /home/julian/testing/gcc4.9-builds/XIOS/xios-2.0 # CHANGE ME
%CPP cpp-4.9 # CHANGE ME
%CPPFLAGS -P -traditional
%XIOS_INC -I%XIOS_HOME/inc
%XIOS_LIB -L%XIOS_HOME/lib -lxios
%NCDF_INC -I%NCDF_HOME/include
%NCDF_LIB -L%NCDF_HOME/lib -lnetcdf -lncdf -lstdc++
%FC mpif90 # CHANGE ME
%FCFLAGS -fdefault-real-8 -O3 -funroll-all-loops -fcray-pointer -cpp -ffree-line-length-none
%FLAGS %FCFLAGS
```

(continues on next page)
The main changes are (again, see here for an attempt at the reasoning and a log of errors that motivates the changes):

- added `%NCDF_HOME` to point to where NetCDF lives
- added `%XIOS_*` keys to point to where XIOS lives
- added `%CPP` and flags, consistent with using gcc4.9
- added the `-lnetcdff` and `-lstdc++` flags to NetCDF flags
- using mpif90 which is a MPI binding of gfortran-4.9
- added `-cpp` and `-ffree-line-length-none` to Fortran flags
- swapped out `gmake` with `make`

Then, I did (see NEMO 3.6 for the reason):

```bash
cd ../CONFIG/
./makenemo -j0 -r GYRE -n GYRE_testing -m gfortran_local
```

Edit `/GYRE_testing/cpp_GYRE_testing.fcm` and replaced `key_top` with `key_nosignedzero` (does not compile TOP for speed speeds, and make sure zeros are not signed). Then

```bash
./makenemo -j2 -n GYRE_testing -m gfortran_local |& tee compile_log.txt
```

which should compile and take a few minutes. Check that it does run with the following:

```bash
cd GYRE_testing/EXP00
mpiexec -n 1 ./opa
```

This may be `mpirun` instead of `mpiexec`, and `-n 1` just runs it as a single core process. Change `nn_itend = 4320` in `nn_itend = 120` to only run it for 10 days (rdt = 7200 which is 2 hours). With all the defaults as is, there should be some `GYRE_5d_*.nc` data in the folder. You can read this with `ncview` (see the `ncview` page or, if you have `sudo` access, you can install it through `sudo apt-get install ncview`), bearing in mind that this is actually a rotated gyre configuration (see the following NEMO forge page or search for `gyre` in the NEMO book).

**Note:** If your installation compiles but does not run with the following error

```
dyld: Library not loaded: @rpath/libnetcdff.6.dylib
Referenced from: /paths/./nemo
Reason: no suitable image found. Did find:
/usr/local/lib/libnetcdff.6.dylib: stat() failed with errno=13
```

then it is not finding the right libraries. These could be fixed by adding the `-Wl,-rpath,./fill me in/lib` flag to the relevant flags bit in the `$fcm` (or possibly in XIOS the `path` and/or `env`) files (in this case it is NetCDF as it calls the `libnetcdff.6` library) specifying exactly where the libraries live. This can happen for example on a Mac or if the libraries are installed not at the usual place.
Note: One infuriating problem I had specifically with a Mac (though it might be a gcc4.8 issue) is that the run does not get beyond the initialisation stage. Going into ocean.output and searching for ERROR shows that it could complain about a misspelled namelist item (in my case it was in the namberg namelist). If you go into output.namelist.dyn and look for the offending namelist is that it might be reading in nonsense. This may happen if the comment character ! is right next to a variable, e.g.

\begin{verbatim}
ln_icebergs = .true.!this is a comment
\end{verbatim}

Fix this by adding a white space, i.e.

\begin{verbatim}
ln_icebergs = .true. !this is a comment
\end{verbatim}

which should fix it...

1.3 NEMO 4.0 (beta) + XIOS 2.5

Tested with

\begin{itemize}
  \item gcc4.9, gcc5.4 on a laptop (Ubuntu 16.04)
  \item gcc4.9 on a modular system (Ubuntu 14.04, Oxford AOPP)
  \item gcc4.8 on a Mac (El Capitan OSX 10.11)
\end{itemize}

The code structure in NEMO 4.0 and the use of some commands are slightly different (at least in v9925) and will be documented below (please see the official NEMO announcement for details). If you get errors that are not documented here, see if the XIOS1.0 NEMO3.6 page contains the relevant errors.

The assumption here is that the compiler is fixed and the packages (e.g., NetCDF4 and a MPI bindings) are configured to be consistent with the compilers. See here to check whether the binaries exist, where they are, and how they might be installed.

The instructions below assumes gcc4.9 compilers but works for gcc4.8 and gcc5.4 compilers too (with one extra flag required in XIOS compilation for the latter). I defined some extra variables on a Linux machine:

\begin{verbatim}
export $BD=/home/julian/testing/gcc4.9-builds # CHANGE ME
export C_INCLUDE_PATH=$BD/install/include:$C_INCLUDE_PATH
export CPLUS_INCLUDE_PATH=$BD/install/include:$CPLUS_INCLUDE_PATH
export LIBRARY_PATH=$BD/install/lib:$LIBRARY_PATH
export LD_LIBRARY_PATH=$BD/install/lib:$LD_LIBRARY_PATH
\end{verbatim}

Otherwise I have found the resulting libraries and binaries are not necessarily linked to the right ones (I have a few versions of libraries at different places as a result of the testing recorded here). You shouldn’t need to do the above if the packages are forced to look at the right place, though the above may help.

On a Mac done through anaconda the above was not necessary. My understanding is that setting these variables might not actually do anything unless an option is specifically enabled in Xcode.

1.3.1 XIOS 2.5 (svn v1566)
Note: Looks like you could use XIOS 2.0 with NEMO 4.0, so if the following doesn’t work for you, try compiling \textit{XIOS 2.0} instead.

Do the following:

```
mkdir XIOS
cd XIOS
```

To get XIOS to compile, the compilers and packages need to be pointed to first, via modifying files in \texttt{arch}. Since I am using \texttt{gcc}, I did the following just to make a fresh copy:

```
cd xios2.5/arch
cp arch-GCC LINUX.env arch-GCC local.env
cp arch-GCC LINUX.fcm arch-GCC local.fcm
cp arch-GCC LINUX.path arch-GCC local.path
```

The \texttt{*.env} file specifies where HDF5 and NetCDF4 binaries live. The \texttt{*.fcm} file specifies which compilers and options to use. The \texttt{*.path} file specifies which paths and options to include. My files look like the following:

```
# arch-GCC_local.env
export HDF5 INC DIR=/usr/local/include # CHANGE ME
export HDF5 LIB DIR=/usr/local/lib # CHANGE ME
export NETCDF INC DIR=/usr/local/include # CHANGE ME
export NETCDF LIB DIR=/usr/local/lib # CHANGE ME
```

You could check where the HDF5 and NetCDF4 directories are by doing \texttt{which h5copy} and \texttt{which nc-config}, which should give you a directory/bin, and it is the directory part you want. If you did install the libraries somewhere else as in \textit{other packages}, say, then make sure the which commands are pointing to the right place.

```
# arch-GCC_local.fcm
```

(continues on next page)
Check the MPI locations by doing `which mpicc` and `mpicc --version` say. If they are the right ones you could just have `mpicc` instead of the full path as given above. MPI bindings are used here to avoid a possible error that may pop up in relation to the build trying to find `mpi.h`. The `gmake` command was swapped out by the `make` command (I don’t have `cmake`).

**Note:** For `gcc5.4` and maybe newer versions, doing just the above when compiling leads to a whole load of errors about clashing in C++:

```cpp
../include/boost/functional/hash/extensions.hpp:69:33: error: `template<class T,˚
    std::size_t boost::hash_value` conflicts with a previous declaration
    std::size_t hash_value(std::list<T, A> const & v)

```

Adding `-D_GLIBCXX_USE_CXX11_ABI=0` to `%BASE_CFLAGS` fixes these.

A difference I’ve found between XIOS 2.5 and other XIOS versions is that doing just the above might lead to an error like the following:

```cpp
This file requires compiler and library support for the ISO C++ 2011 standard. This support is currently experimental, and must be enabled with the -std=c++11 or -std=gnu++11 compiler options.
```

Adding `-std=c++11` to `%BASE_CFLAGS` seems to fix this.

You might also get the following:

```cpp
SUBROUTINE cxios_set_interpolate_domain_read_write_convention(interpolate_domain_hdl,˚
    read_write_convention, read_write_conventi
        Error: Unexpected junk in formal argument list at (1)
```

The Fortran lines are too long, so fix this by adding `-ffree-line-length-none` to `%BASE_FFLAGS`.

```bash
# arch-GCC_local.path
NETCDF_INCDIR="-I$NETCDF_INC_DIR"
NETCDF_LIBDIR="-Wl,'--allow-multiple-definition' -L$NETCDF_LIB_DIR"
NETCDF_LIB="-lnetcdff -lnetcdf"

MPI_INCDIR=""
MPI_LIBDIR=""
MPI_LIB=""

HDF5_INCDIR="-I$HDF5_INC_DIR"
HDF5_LIBDIR="-L$HDF5_LIB_DIR"
HDF5_LIB="-lhdf5_hl -lhdf5 -lhdf5 -lz"
```

The above has all the OASIS (the atmosphere / ocean coupler) keys removed. I added the `-Wl, '--allow-multiple-definition'` key for reasons I don’t remember anymore...

I went into `bld.cfg`, found the line
and changed `src_netcdf` to `src_netcdf4` (see *XIOS1.0 stuff* for the reason).

Now it should be ready to compile. Assuming the current directory is `xios2.5/arch`:

```bash
cd ../
./make_xios --full --prod --arch GCC_local -j2 |& tee compile_log.txt
```

The `-j2` option uses two processors to build. The `tee` command is to keep logs of potential errors (the `|&` is short for `2>&1 |` ) for debugging the compiler issues that may arise. It should work and takes around 5 mins to compile for me. The main end result is are binaries in `xios2.5/bin/` which NEMO will call.

### 1.3.2 NEMO 4.0 (svn v9925)

There is a restructuring of folders (see the *official announcement* for details) so the commands below will reflect this.

Check out a version of NEMO. I have another folder separate to the XIOS folders to contain the NEMO codes and binaries:

```bash
mkdir NEMO
cd NEMO
svn co http://forge.ipsl.jussieu.fr/nemo/svn/NEMO/trunk@9925 nemo4.0-9925
```

This checks out version 9925 (NEMO 4.0 beta) and dumps it into a folder called `nemo4.0-9925` (change the target path to whatever you like). A similar procedure to specify compilers and where XIOS lives needs to be done for NEMO. Again, because of the compilers I am using:

```bash
cd nemo4.0-9925/arch
cp arch-linux_gfortran.fcm ./gfortran_local.fcm
```

None of the fcm files associated with gfortran actually worked for me out of the box so here is my build of it (click [HERE](#) for a detailed log of how I got to the following):

```bash
# gfortran_local.fcm

# generic gfortran compiler options for linux
# NCDF_INC netcdf include file
# NCDF_LIB netcdf library
# FC Fortran compiler command
# FCFLAGS Fortran compiler flags
# FFLAGS Fortran 77 compiler flags
# LD linker
# LDFLAGS linker flags, e.g. `-L<lib dir>` if you have libraries in a
# FPPFLAGS pre-processing flags
# AR assembler
# ARFLAGS assembler flags
# MK make
# USER_INC additional include files for the compiler, e.g. `-I<include dir>`
# USER_LIB additional libraries to pass to the linker, e.g. `-l<library>`

%NCDF_HOME /usr/local # CHANGE ME
%XIOS_HOME /home/julian/testing/gcc4.9-builds/XIOS/xios-2.5 # CHANGE ME
```

(continues on next page)
The main changes are (again, see here for an attempt at the reasoning and a log of errors that motivates the changes):

- added %NCDF_HOME to point to where NetCDF lives
- added %XIOS_* keys to point to where XIOS lives
- added %CPP and flags, consistent with using gcc4.9
- added the -lnetcdff and -lstdc++ flags to NetCDF flags
- using mpif90 which is a MPI binding of gfortran-4.9
- added -cpp and -ffree-line-length-none to Fortran flags
- swapped out gmake with make

Go into the configuration folder by

```
cd ../cfgs
```

One of the things I noticed is that makenemo now seems to work slightly differently (at least with this version). Normally you can do makenemo -r GYRE -n GYRE_testing -j0 -m gcc_fortran_local, which copies a configuration but does not compile it, so you can edit the cpp flags before compiling (and note that it adds an entry into works_cfgs.txt). However now it seems you have to specify a -r flag or a -d flag (which specifies what NEMO modules the configuration should have), whereas before just a -n flag would work by itself.

You could just compile as usual with makenemo (see NEMO 3.6 for syntax). The slightly untidy way to circumvent errors that I know will come up was to do the following:

1. Open refs_cfg.txt, copy the GYRE_PISCES OCE TOP line and paste it at the bottom, but then change the configuration name (GYRE_PISCES to GYRE тестing in my case), save and close it;

2. Then do

```
mkdir GYRE_testing
rsync -arv GYRE_PISCES/* GYRE_testing/
```
3. I opened /GYRE_testing/cpp_GYRE_testing.fcm and replaced key_top with key_nosignedzero (does not compile TOP for speed speeds, and make sure zeros are not signed), save it;

4. Compile with (because makenemo is now one level up)

```
../makenemo -j2 -r GYRE_testing -m gfortran_local & tee compile_log.txt
```

(note the -r rather than -n flag here).

**Warning:** See if this feature of makenemo has been modified in the trunk?

Note the executable opa is now called nemo (so make sure you change those submission scripts on the relevant clusters if you use NEMO on them). Check that it does run with the following:

```
cd GYRE_testing/EXP00
mpiexec -n 1 ./nemo
```

Note that what used to be solver.stat is now called run.stat, and there is an extra run.stat.nc for whatever reason. The ocean.output file is still the same.

**Note:** If your installation compiles but does not run with the following error

```
dyld: Library not loaded: @rpath/libnetcdff.6.dylib
Referenced from: /paths/./nemo
Reason: no suitable image found. Did find:
/usr/local/lib/libnetcdff.6.dylib: stat() failed with errno=13
```

then it is not finding the right libraries. These could be fixed by adding the -Wl,-rpath,/fill me in/lib flag to the relevant flags bit in the *.fcm (or possibly in XIOS the path and/or env files) (in this case it is NetCDF as it calls the libnetcdff.6 library) specifying exactly where the libraries live. This can happen for example on a Mac or if the libraries are installed not at the usual place.

**Note:** One infuriating problem I had specifically with a Mac (though it might be a gcc4.8 issue) is that the run does not get beyond the initialisation stage. Going into ocean.output and searching for E R R O R shows that it could complain about a misspelled namelist item (in my case it was in the namberg namelist). If you go into output.namelist.dyn and look for the offending namelist is that it might be reading in nonsense. This may happen if the comment character ! is right next to a variable, e.g.

```
ln_icebergs = .true.!this is a comment
```

Fix this by adding a white space, i.e.

```
ln_icebergs = .true. !this is a comment
```

which should fix it...

## 1.4 Other packages

Tested with
• gcc4.9, gcc5.4 on a laptop (Ubuntu 16.04)
• gcc4.9 on a modular system (Ubuntu 14.04, Oxford AOPP)
• gcc4.8 on a Mac (El Capitan OSX 10.11)

The following packages are needed for NEMO and XIOS and they may need to be installed or configured accordingly. I don’t have a windows machine handy (and I don’t really want to try it there either) so for that I would recommend doing the following through virtualbox or something analogous (which might be another way to do it on a Mac); I am guessing cygwin and the new Windows 10 terminals might be a possibility.

Note: I would suggest trying the following in reverse order of effort required:

1. Get someone who knows what they are doing to do it for you! Compiling the following from scratch is not the most interesting activity and is actually quite fiddly (especially the HDF5 and NetCDF4 stuff)...if you don’t have access to people who can do that, then try

2. Doing it through anaconda. There you are somewhat restricted to a certain set of compilers (gcc 4.8) but anaconda sorts out the dependencies for you. The only thing then you need to do is to force XIOS and NEMO to use the libraries within the anaconda installation. Failing that...

3. Do it from scratch. I’m sorry and good luck; see below for some notes to possibly ease your pain.

As of 24 Oct 2018, the following remains on the agenda:

• intel compilers
• reproducing sample compatibility errors

1.4.1 Anaconda

Anaconda is a framework mostly for downloading Python packages, with the added advantage that it resolves the package dependencies for you (cf. apt, yum on a Linux machine or port on a Mac if you have MacPorts). See the official conda manual or some of my own notes on some things to do with installing and managing conda. I used the full anaconda with Python 3.6 but you could use miniconda or with other pythons probably.

First I created an environment so all the changes only apply in that environment:

```bash
conda create -n nemo python=3.6
```

Accept to install the basic packages for the environment. Then activate the nemo environment with

```bash
>> julian@psyduck:~/
source activate nemo
>> (nemo) julian@psyduck:~/
```

Now if you have compilers you want to use already then you can skip the compiler installation. On the Mac I was dealing with there was no gcc or a Fortran compiler and I had problems with clang, so I did the following to get a set of gcc compilers:

```bash
conda install gcc
conda install gfortran_osx-64
```

The second line you should change to gfortran_linux-64 if on a Linux machine. The command will add some compiler flags that is unset when exiting from the environment. Check that the compilers are the now default compilers by doing gcc --version (which should probably give 4.8) and which gcc (which should point to the anaconda folder). If not, do something like echo CC and export CC=/folder/bin to force it to point to the right folder (also do it for FC and CXX, and maybe put it in the $PATH variable; see below).
Note: One thing I found to be an issue is that while gfortran can compile a sample program through

```
program hello
    print *, "hi mum"
end program
```

Executing through ./hi could throw a library complaint:

```
dyld: Library not loaded: @rpath/libgfortran.3.dylib
Referenced from:
  /usr/local/lib/libnetcdff.3.dylib: stat() failed with errno=13
```

So the problem here is that the computer is looking for the library at the wrong place. To force the computer to look at the right place, try

```
export FCFLAGS=-Wl,-rpath,${CONDA_PREFIX}/lib
```

where `${CONDA_PREFIX}` should have been defined by anaconda.

If you already have the MPI capabilities bound to the compilers you will use then you can skip the following. To make life easier it is advisable to install either MPICH (Linux?) or OpenMPI (Mac?). You could try this by

```
conda install -c conda-forge mpich
conda install -c conda-forge openmpi
```

and check whether which mpicc and in particular which mpif90, which should be pointed to the gcc compilers. I had a similar problem with gfortran not being bound properly, which could be fixed with setting FCFLAGS, or to compile it from scratch (see below for the way to do it for MPICH, which also works for OpenMPI with suitable changes in the hyperlink address; do a search for this in Google).

To get NetCDF4 and its dependencies I did

```
conda install netcd4
conda install -c conda-forge netcdf-fortran
```

Do which nc-config and nc-config --all to see which paths are being pointed to. Again, you may need to add the FCFLAGS detailed above to make sure it is pointing to the right libraries. Take note of the path where the libraries and header files live and put those into the XIOS and NEMO files and that should be it!

### 1.4.2 Compiling it yourself

(Good luck!)

The following has been tried on a Linux machine. I had some problems on a Mac with Clang that I don’t know how to fix without sudo access but it is probably fixable; I have not tried installing things with port through MacPorts partly because it requires Xcode to be installed.

If you want a script to do all of the following on a Linux machine in one go, then please scroll right to the bottom of this page. The way I went about it was to first choose a set of compilers and use the same set of compilers to install the dependencies, primarily to avoid errors relating to compatibility of packages. For example, gcc4.9 was downloaded through sudo apt-get install gcc4.9, or loaded through a network computer through something like a module load command. You may have to look it up on the internet if you don’t have either of these.

The order I did them in are:

1.4. Other packages

---

---
1. mpich (to bind the set of compilers to a MPI form; I chose mpich but it should work on OpenMP too)
2. zlib (1.2.11, for HDF5)
3. hdf5 (1.8.19, for NetCDF)
4. netcdf (4.4.1.1) and netcdf-fortran (4.4.4), for XIOS

Within a folder called gcc4.9-builds, I added an extra extra_variables file containing the following:

```bash
export BD=/home/julian/testing/gcc4.9-builds  # CHANGE ME
export CC=/usr/bin/gcc-4.9
export CXX=/usr/bin/g++-4.9
export FC=/usr/bin/gfortran-4.9
export F77=/usr/bin/gfortran-4.9
export CPP=/usr/bin/cpp-4.9
export C_INCLUDE_PATH=$BD/install/include:$C_INCLUDE_PATH
export CPLUS_INCLUDE_PATH=$BD/install/include:$CPLUS_INCLUDE_PATH
export LIBRARY_PATH=$BD/install/lib:$LIBRARY_PATH
export LD_LIBRARY_PATH=$BD/install/lib:$LD_LIBRARY_PATH
export PATH=$BD/install/bin:$PATH
```

Set this by doing `source extra_variables`, and upon closing the terminal the variables will be flushed. Some of these may want to be added to ~/.bashrc for convenience. The instructions below attempts to build shared rather than static libraries, and somewhat depends LD_LIBRARY_PATH variable being set (with the added bonus that the ldd command provides an extra check whether the correct libraries are being called). Suggestions on how to build the packages without setting LD_LIBRARY_PATH or build static packages are given below (using LD_LIBRARY_PATH can be dangerous, see e.g., here).

**Note:** Do for example `$CC --version` or `echo $CC` to see what the variables are set to. If you don’t want to set the compiler variables then you need to do e.g.

```bash
CC=/usr/bin/gcc-4.9 FC= something ./configure something
```

where the path points to where the compiler binary lives. This then only sets the variable temporarily for the particular command.

Some or all of these may be skipped depending on which ones packages you have already installed and/or configured. The following installs all the libraries and binaries to the folder specified in $BD; you have sudo access you could always just install it to /usr/local. The sub-directories in the folder are:

- source, where all the compressed files are going to live;
- build, where all the source file folders are going to live
- install, where all the compiled libraries, binaries and header files are going to live.

source and build can be deleted later.

**Note:** The binaries built here will not register by default unless it is added to the $PATH variable. If you are going to add to the $PATH variable, the one that gets registered first gets priority, i.e.

```bash
echo $PATH
> /home/julian/testing/gcc4.9-builds/install/bin:/usr/local/bin
```
means any binaries in /home/julian/testing/gcc4.9-builds/install/bin gets used first. Do this by adding to
~/.bashrc the following:

```bash
export PATH=/usr/local/bin:$PATH
```

If you don’t do this then it just means when you call the binaries you have to provide an explicit call, e.g., /home/julian/testing/gcc4.9/build/bin/mpif90. Do for example which mpif90 to check what the mpif90 is linked to; if you did add to $PATH then the which command above should point to the right binary.

### 1.4.3 MPICH

Check if there are any MPI capabilities and which compilers they are bound to:

```bash
mpicc --version
which mpicc
```

If you have these already they may not need to be installed. If they need to be installed separately for whatever reason, then you could do the following. I took the source files from the MPICH website itself and chose v3.0.4 here. Being in the $BD folder, I did:

```bash
cd $BD/source/
wget http://www.mpich.org/static/downloads/3.0.4/mpich-3.0.4.tar.gz
cd $BD/build/
tar -xvzf $BD/source/mpich-3.0.4.tar.gz
cd mpich-3.0.4
./configure prefix=$BD/install/
makes -j 2
make check install
```

Within install/ there should now be some folders that can be pointed to for the binaries, libraries and header files to include for later installations.

**Note:** The ./configure prefix= step requires an absolute (not relative) path; change this to change the installation folder.

### 1.4.4 zlib and HDF5

Check whether HDF5 exists first (may still need to be installed again for compatibility reasons). h5copy is the command that should exist if HDF5 is installed:

```bash
which h5copy
h5copy --version
```

If you still want to install both zlib and HDF5, then do the following (following the instructions on the Unidata UCAR website). The raw files are taken from the HDF5 website using HDF5 v1.8.19. Again, with $BD as defined:

```bash
cd $BD/source/
wget http://www.zlib.net/zlib-1.2.11.tar.gz
cd $BD/build/
tar -xvzf $BD/source/zlib-1.2.11.tar.gz
cd zlib-1.2.11
```

(continues on next page)

1.4. Other packages
CFLAGS=-fPIC ./configure --prefix=$BD/install/
make -j 2
make check install
cd $BD/source/
wget https://support.hdfgroup.org/ftp/HDF5/releases/hdf5-1.8/hdf5-1.8.19/src/hdf5-1.8.19.tar.gz
cd $BD/build/
tar -xvzf $BD/source/hdf5-1.8.19.tar.gz
cd hdf5-1.8.19
#CPPFLAGS=-I$BD/install/include LDFLAGS=-L$BD/install/lib \
CFLAGS=-fPIC ./configure --enable-shared --enable-fortran --enable-cxx \
# --with-zlib=$BD
--prefix=$BD/install/ 
make -j 2
make check install
cd $BD

Note: If LD_LIBRARY_PATH is set then accordingly then zlib should be detected by the HDF5 install. If not, consider including the commented out CPPFLAGS and LDFLAGS or the --with-zlib line (or both).

HDF5 checking and installation can take a while. If it’s more that 30 mins however it probably has crashed.

If a shared build option was on, then you can do ldd h5copy (or wherever h5copy is installed at if the directory has not been added to $PATH) to check that libhdf5 does point to where you think it should point to. If it isn’t, then try the first point in this note.

If an error shows up saying recompile with -fPIC, then trying doing a static build. Replace --enable-shared with --disable-shared and do the first point in this note, possibly adding LIBS="-lz -lhdf5 etc.; see here for a guide. I would be tempted to keep the CFLAGS=-fPIC so shared builds of NetCDF4 can still be made.

1.4.5 NetCDF4

Check whether NetCDF4 exists first (may still need to be installed again for compatibility reasons). nc-config is the command that should exist if NetCDF4 is installed, and shows where it is installed and what compilers were used to build it.

nc-config all

If you still want to install it, then do the following (following the instructions on the Unidata UCAR website). The raw files are taken from the the NetCDF4 website, using netcdf v4.4.1.1 and netcdf-fortran v4.4.4:

cd $BD/source/
wget ftp://ftp.unidata.ucar.edu/pub/netcdf/netcdf-4.4.1.1.tar.gz
cd $BD/build/
tar -xvzf $BD/source/netcdf-4.4.1.1.tar.gz
cd netcdf-4.4.1.1
#CPPFLAGS=-I$BD/install/include LDFLAGS=-L$BD/install/lib \
./configure --enable-netcdf4 --enable-shared --prefix=$BD/install/ 
make -j 2
make check install

(continues on next page)
Note: NetCDF4 checking and installation can take a while. If it's more that 30 mins however it probably has crashed.

If a shared build option was on, then you can do `ldd ncdump` (or wherever `ncdump` was installed if the directory has not been added to `$PATH`) and check that `libnetcdf`, `libhdf5` and `libz` really does point to where you think it should point to. If not, consider doing something similar to the HDF5 note above.

If an error shows up saying `recompile with -fPIC`, then trying doing a static build (I had this problem on one of the computers where the Fortran part is static). See HDF5 note above.

I had a problem with not having the m4 package, which I just installed as the installation commands above, with the binaries found from `wget ftp://ftp.gnu.org/gnu/m4/m4-1.4.10.tar.gz`. This is not in the script below.

This should be it! Try `./install/bin/nc-config --all` and/or `./install/bin/nf-config --all` to see where everything is configured. The things in `build/` and `source/` may now be deleted.

1.4.6 Combined shell script

A script that does all of the above in one go may be found in the following commands (use at your own risk):

```
mkdir gcc4.9-builds/  # CHANGE ME
cd gcc4.9-builds/    # CHANGE ME
wget https://raw.githubusercontent.com/julianmak/NEMO-related/master/docs/compile_dependencies.sh
chmod +x compile_dependencies.sh
```

Before you execute the shell script with `./compile_dependencies.sh`, make sure the compilers are pointed to appropriately. You can do this in `~/.bashrc` (see first code block on this page) or within the shell script itself (it is commented out at the moment). If some packages already exist and you don’t want them installed, comment the appropriate lines.
Other NEMO notes

Here you will find some of my notes relating to NEMO.
If you are interested in building a model, see how I went about doing it in the model building subsection.

2.1 Adding code to NEMO

2.2 Other NEMO packages

2.3 GYRE: rotated gyre model

2.3.1 Brief overview and sample outputs

2.3.2 How to get the model running

GYRE is hard-coded into NEMO so nothing needs to be provided to run it out of the box.

2.3.3 Custom analysis scripts

2.4 ORCA: global configuration

2.4.1 Brief description
Fig. 1: Absolute speed of surface currents between ORCA at a nominal horizontal resolution of 1/12 ° and 1 ° (there is a refinement towards the equators). The ORCA 1/12 ° data was obtained from the NOC Jasmin archives.

### 2.4.2 How to get the model running

### 2.4.3 Custom analysis scripts

### 2.5 UNAGI: custom channel model

#### 2.5.1 Brief overview and sample outputs

UNAGI (naming based on EEL which was original due to Marina Levy) is a re-entrant $\beta$-plane channel with temperature as the thermodynamic variable that’s largely based on Dave Munday’s MITgcm channel model reported in [MJM15] as an idealised model to the Antarctic Circumpolar Current. The model at present takes in some to-be specified bathymetry, wind stress profile and initial state (which may be customised accordingly within the gen_UNAGI_fields.ipynb).

With the choice of SST restoring over the surface layer, to maintain a sensible thermocline the vertical tracer diffusivity is enhanced in a sponge region to the north (see [MJM15]). See e.g. [AMF11] for alternative model formulations. Some model set up choices:

- relatively long re-entrant zonal channel, no topography except ridge in the middle of the channel extending up to half the depth of the domain
- fixed sinusoidal wind stress with some peak wind stress value $\tau_0$
- SST restoring (a relatively hard restoring, the $\text{rn}_dqdt$ value in $\text{namelist}_cfg$ has been amplified by a factor of 2)
- linearly varying temperature profile at the surface with $e$-folding depth of 1000 metres
- linear friction
- linear EOS with only temperature as the variable
- sponge region to the north where vertical diffusivity is amplified by a factor of 250 from the background value of $10^{-5} \text{m}^2 \text{s}^{-1}$

The diagram below shows an output from the 10km resolution model with biharmonic tracer diffusion and no eddy parameterisation, showing a rich eddying field.
2.5.2 How to get the model running

[TO BE ADDED]

2.5.3 Custom analysis scripts

[TO BE ADDED]

2.5.4 Notes 1: building a custom model

The following approach is strictly for NEMO models beyond v3.6, where one can build a customised model through providing a domcfg.nc, which is the main goal of the following text. The details are given below are what I did for the idealised channel model UNAGI.

The biggest obstacle in generating the appropriate domcfg.nc file for me was in transferring the code that modifies the vertical spacing variables e3t/u/v/w to have a partial cell description. I first tried to brute force it by writing from scratch a file that provides all the relevant variables needed in the domcfg.nc; see for example the input required in ORCA2. I gave up after a while and fell back to using the NEMO native [MI96] grid and the TOOLS/DOMAINcfg package, as follows:

1. in an external folder (e.g., ~/Python/NEMO/UNAGI), create the bathymetry data through a program of your choice (I did it in Python [LINK TO ADD]) and output it as a netCDF file (e.g. bathy_meter.nc)
2. link/copy it as bathy_meter.nc (the tool requires that specific naming) into the TOOLS/DOMAINcfg that comes with NEMO
3. modify the namelist_cfg file accordingly for the horizontal and vertical grid spacing parameters (see here for usage and compiling notes)
4. a domcfg.nc should result (if not, see ocean.output for messages), copy it back into the working folder in step 1
5. open domcfg.nc and use those variables to create the state.nc and forcing.nc file again in the program of your choice (this is mostly to keep consistency)
6. copy the domcfg.nc, state.nc and forcing.nc (I prefixed them with something, e.g. UNAGI_R010_domcfg.nc) and modify the namelist cfg accordingly, e.g.

```fortran
!-----------------------------------------------------------------------
&namrun          ! parameters of the run
!-----------------------------------------------------------------------
```

(continues on next page)
That is more or less it. Once you can build the domain variables the model will at least run and the rest is more to do with experimental design.

### 2.5.5 Notes 2: hacking NEMO to get UNAGI

That two main things that needed hacking into NEMO for UNAGI are the vertical tracer diffusion (in the sponge region to the north) and possible combination with the GEOMETRIC parameterisation. [TBC, 15 Apr 2019]

### 2.6 pyCDFTOOLS

For various reasons (mostly personal preference and forcing myself to write in Python) I made a translation of sorts of CDFTOOLS in Python. pyCDFTOOLS I think is:

- slightly more flexible, e.g., no need to recompile if variable name changes between files
- saves on the creation and reading of files
- everything done within Python, rather than Fortran and MATLAB say
- marginally more up-to-date, e.g. dealings with TEOS-10 equation of state

On the other hand, it is

- not as complete, because I only translated ones that I needed (see here for list)...
• not as fast (though things that I could not vectorise I used JIT to speed up the looping)
• not NEMO code compliant (CDFTOOLS is designed to conform to NEMO code conventions)

An additional criticism I have is that I wrote pyCDFTOOLS more like Fortran/MATLAB and not making full use of the Python functionalities (e.g., Panda and so forth). I have some idea how I might get it to work but watch this space...

The routine naming conventions of the programs are basically the same as CDFTOOLS (see MEOM page). All codes with the prefix cdf are based on CDFTOOLS; all errors are entirely mine (any things I did change are commented in the code).
Grab it with:

```
git clone https://github.com/julianmak/NEMO-related
```

Some slightly more configuration/model specific Python scripts and notebooks are in other folders (e.g., GYRE and ORCA). I tend to just do

```
cd GYRE
rsync -arv ../pyCDFTOOLS .
```

which then means the scripts and notebooks within the folder have access to the module, and it separates out a version that I do testing on.

CDFTOOLS itself depends on the following packages (the things I think that come as standard are omitted):

• numba (for JIT to speed up loops)
•umpy (for tools)
• netCDF4 (for reading)
• scipy (for the occasional times when a MATLAB file is read)

The configuration specific programs depend additionally on Matplotlib and a whole load of other ones for the ORCA configuration; see the relevant pages. I installed most of the things through Anaconda; see the Python page here for my notes on these.

Use these scripts at your own risk and feel free to modify them (rights etc. as stated in the license and in line with the CDFTOOLS one). For comparison purposes you may also want to grab CDFTOOLS to compare results (see the CDFTOOLS page):

```
git clone https://github.com/meom-group/CDFTOOLS
```

**Note:** The programs I have uploaded I was satisfied enough with the tests I have done, but don’t just take my word for it :-)
GEOMETRIC (Geometry and Energetics of Ocean Mesoscale Eddies and Their Rectified Impact on Climate) is an approach to representing the unresolved turbulent eddies in ocean climate models, first derived in [MMB12]. David Marshall’s page has an excellent outline and summary of GEOMETRIC, so this page will focus on outlining the details relating to the NEMO implementation.

The implementation of GEOMETRIC was done by providing a new module ldfeke.f90 and adding appropriate calls and variables to ldftra.f90, step.f90 step_oce.f90 and nemogcm.f90. This was initially done in SVN version 8666, which is somewhere between the 3.6 stable and 4.0 beta, by myself and Gurvan Madec back in November 2017. The current implementation of GEOMETRIC is what may be considered GM-based [GM90] and follows the prescription described in [MMMM18]. The GEOMETRIC scaling gives \( \kappa_{gm} = \alpha E(N/M^2) \) (see below for symbol definitions). While \( \alpha \) is prescribed and \( M \) and \( N \) are given by the coarse resolution ocean model, information relating to \( E \) is provided by a parameterised eddy energy budget. The recipe for GEOMETRIC then is as follows:

1. time-step the parameterised eddy energy budget to get \( E \) with info provided by the GCM
2. calculate the new \( \kappa_{gm} \)
3. use the existing GM routines with new \( \kappa_{gm} \) and time-step the GCM. Cycle as appropriate.

The current NEMO implementation considers an eddy energy field that varies in longitude, latitude and time (and so \( \kappa_{gm} \) inherits this spatio-temporal dependence), given by

\[
\frac{d}{dt} \int E \, dz + \nabla \cdot \left( (\bar{u} - c e_1) \int E \, dz \right) = \int \kappa_{gm} \frac{M^4}{N^2} \, dz - \lambda \int E \, dz + \nu E \nabla^2 \int E \, dz,
\]

(respecively, the time-evolution, advection, source, dissipation and diffusion of eddy energy), with \( \kappa_{gm} \) calculated as

\[
\kappa_{gm} = \alpha \frac{\int E \, dz}{\int \Gamma(M^2/N) \, dz} \Gamma(z).
\]

The symbols are as follows:
symbol | definition | units
--- | --- | ---
$\alpha$ | eddy efficiency parameter non-dimensional, $|\alpha| \leq 1$ | $-$
$E$ | total eddy energy | $m^2 s^{-2}$
$\bar{M}, \bar{N}$ | mean horizontal and vertical buoyancy gradient | $s^{-1}$
$\bar{u}$ | depth-mean flow | $m^2 s^{-1}$
c | long Rossby phase speed of 1st baroclinic mode | $m^2 s^{-1}$
$\kappa_{gm}$ | Gent–McWilliams coefficient | $m^2 s^{-1}$
$\lambda$ | linear damping rate of eddy energy | $s^{-1}$
$\nu_E$ | Laplacian diffusion of eddy energy | $m^2 s^{-1}$

### 3.1 Advection

The advection of eddy energy is given in flux form and has a contribution from the depth-mean flow as well as a contribution associated with the westward propagation of eddies at the long Rossby phase speed (motivated by e.g. [CSS11] and [KM14]). The advection is by the barotropic mean flow already computed in NEMO, with a first order upwind scheme. The Rossby wave contribution requires computing for the eigenvalue associated with the first baroclinic mode and uses two subroutines (eke_rossby and eke_thomas) the WKB expression given in [CdeSzoekeS+98] (their equation 2.2). See [here](#) for usage and implementation details.

**Note:** As of Feb 2019 the removal of the routines to solve the tri-diagonal eigenvalue problem means the nn_wav_cal variable in namelist_cfg has been removed.

### 3.2 Source

The source of mesoscale eddy energy here is only from the slumping of neutral surfaces through the eddy induced velocity as parameterised by the GM scheme (note that it is positive-definite). These are straight-forwardly computed as is (rather than using the quasi-Stokes streamfunction) using the already limited slopes computed in NEMO. See [here](#) for implementation details.

### 3.3 Dissipation

The damping of eddy energy is linearly damped and the coefficient is specified in namelist_cfg as a time-scale in days (which is subsequently converted in ldf_eke_init). There is an option to read in an externally prepared NetCDF file geom_diss_2D.nc that varies in longitude and latitude. See [here](#) for usage details, and [here](#) for a sample Python Notebook to generate the file.

### 3.4 Diffusion

The diffusion of eddy energy is through a Laplacian (cf. [EG08]), easily done through copy and pasting code that are in other NEMO modules. The GEOMETRIC scheme is actually stable (most likely because of the upwinding scheme). The diffusion may be switched off by setting rn_eke_lap = 0. in namelist_cfg which will bypass the relevant loop in ldf_eke.
3.5 NEMO implementation

For details relating to the NEMO implementation (e.g. variable names, numerical treatment, namelist_cfg file), please see the following pages.

3.5.1 NEMO implementation details

advection

source

dissipation

3.5.2 NEMO files

Below are sample files or file generators you may need for using GEOMETRIC in NEMO.

Note: As of NEMO 4.0 at least v9925 onwards namtra_ldfeiv needs to be changed to namtra_eiv. Otherwise these could be copy and pasted in. Look into namelist_ref to copy the other entries in if desired.

namelist_cfg

!----------------------------------------------------------------------------------
&namtra_ldfeiv ! eddy induced velocity param.
!----------------------------------------------------------------------------------

\(\text{ln_ldfeiv} = \text{true.} \) ! use eddy induced velocity parameterization
\(\text{ln_ldfeiv_dia} = \text{false.} \) ! diagnose eiv stream function and velocities
\(\text{rn_aeiv_0} = 1000. \) ! eddy induced velocity coefficient [m2/s]
\(\text{nn_aei_ijk_t} = 32 \) ! space/time variation of the eiv coefficient
! ! =-20 (=30) read in eddy_induced_velocity_
\(\rightarrow\) 2D.nc (\(...\)_3D.nc) file
! ! = 0 constant
! ! = 10 \(F(k) = ldf_{cld}\)
! ! = 20 \(F(i,j) = ldf_{c2d}\)
! ! = 21 \(F(i,j,t) = Treguier \text{ et al. JPO 1997}\)
\(\rightarrow\) formulation
! ! = 30 \(F(i,j,k) = ldf_{c2d} + ldf_{cld}\)
! ! = 32 \(F(i,j,t) = \text{GEOMETRIC parameterization}\)
\(\rightarrow\) (=> fill namldf_eke)
\(\text{ln_ekedis} = \text{true.} \) ! switch on update of GEOMETRIC eddy energy equation
\(\rightarrow\) (=> fill namldf_eke)
\(\because\) forced to be true if \(\text{nn_aei_ijk_t} = 32\)

!----------------------------------------------------------------------------------
&namldf_eke ! GEOMETRIC param. (total EKE equation) (\(\text{nn_aei_ijk_t} = 32\))
!----------------------------------------------------------------------------------

\(\text{rn_ekedis} = 180. \) ! dissipation time scale of EKE [days]
\(\text{nn_ekedis} = -20 \) ! dissipation option
! ! = 0 constant in space
! ! = -20 read in geom_diss_2D.nc file

(continues on next page)
J Mak NEMO notes

(continued from previous page)

\[
\begin{array}{l}
\text{rn_geom} = 0.1 \quad ! \text{geometric parameterization master coefficient (}>0 \text{ \& } <1) \\
\text{rn_ekinit} = 1.e-2 \quad ! \text{initial total EKE value} \\
\text{rn_ekin} = 1.e+0 \quad ! \text{background value of total EKE} \\
\text{rn_ross_min} = 4.e+3 \quad ! \text{tapering of aeiv based on min Rossby radius [m]} \\
\text{rn_eklap} = 2000. \quad ! \text{Laplacian diffusion coefficient of EKE} \\
\end{array}
\]

! this is in all options below, so set it to zero, and nothing is done

\[
\begin{array}{l}
\text{rn_aeiv_min} = 1.e+1 \quad ! \text{minimum bound of eiv coefficient} \\
\text{rn_aeiv_max} = 1.5e+4 \quad ! \text{maximum bound of eiv coefficient} \\
\text{rn_SFmin} = 1.0 \quad ! \text{minimum bound of Structure Function} \\
\text{rn_SFmax} = 1.0 \quad ! \text{maximum bound of Structure Function} \\
\text{nn_ekopt} = 1 \quad ! \text{options for terms to include in EKE budget} \\
\end{array}
\]

! = 0 PE->EKE conversion, dissipation only
! = 1 as 0 but with advection
! = 2 as 1 but with additional KE->EKE

--conversion

! for testing purposes:
! = 88 only advection by depth-averaged flow
! = 99 only Laplacian diffusion

\[
\ln_{adv \text{ wav}} = .true. \quad ! \text{include advection at long Rossby speed}
\]

Note: As of Feb 2019 the removal of the routines to solve the tri-diagonal eigenvalue problem means the nn_wav_cal variable in namelist_cfg has been removed.

field_def_nemo-opa.xml

These following needs to be added into field_def_nemo-opa.xml if any of the GEOMETRIC routines in ldfeke is used, so XIOS does not crash the runs. Call these in file_def_nemo.xml as appropriate (see a sample below).

(Note: you may or may not find the bn2 variable (the vertical buoyancy frequency diagnostic) in the T grid for instead of the W grid. The file below has bn2 moved to the W grid group.)

<!-- T grid -->

<FieldGroup id="grid_T" grid_ref="grid_T_2D" >

<!-- GEOMETRIC fields (requires nn_aei_ijk_t = 32) -->

<Field id="eke" long_name="total EKE (EKE+EPE)" unit="m3/s2" />
<Field id="trd_ek advancement ubt" long_name="ubt advective trend of EKE (LHS)" unit="m3/s3" />
<Field id="trd_ek advancement wav" long_name="wav advective trend of EKE (LHS)" unit="m3/s3" />
<Field id="trd_ek advancement lap" long_name="diffusive trend of EKE (RHS)" unit="m3/s3" />
<Field id="trd_ek advancement peS" long_name="PE to EKE source trend (RHS)" unit="m3/s3" />
<Field id="trd_ek advancement keS" long_name="KE to EKE source trend (RHS)" unit="m3/s3" />

(continues on next page)
(continued from previous page)

```xml
  <field id="trd_eke_dis" long_name="dissipation trend of EKE (RHS)" unit="m3/s3" />
</field_group>

<!-- W grid -->

<field_group id="grid_W" grid_ref="grid_W_3D">
  <!-- GEOMETRIC fields (requires nn_aei_ijk_t = 32) -->
  <field id="aeivGeom" long_name="3D w-EIV coefficient from GEOMETRIC param." unit="m2/s" />
  <field id="rossby_rad" long_name="internal Rossby deformation radius" unit="m" grid_ref="grid_W_2D" />
  <field id="bn2" long_name="squared Brunt-Vaisala frequency" unit="s-1" />
  <field id="c1_vert" long_name="1st baroclinic mode phase speed" unit="m/s" grid_ref="grid_W_2D" />
  <field id="c_rose" long_name="long Rossby phase speed" unit="m/s" grid_ref="grid_W_2D" />
</field_group>

file_def_nemo.xml

3.5. NEMO implementation
This manual of sorts is generated using Sphinx in reStructuredText, uploaded to GitHub and generated using ReadTheDocs. The syntax for the relevant rst files I mostly took from the MITgcm ReadTheDocs manual. Included here [TO DO, 04 Jul 2018] are some notes and terminal commands I used to get the underlying python things (which acts as backend for sphinx and the sample notebooks) working.

4.1 Python / Anaconda notes

At some point I encountered some problem with plotting data in MATLAB (to do with the tripolar grid meaning the co-ordinate files were not monotonic so MATLAB hated it), and I went over to Python because the Cartopy and Iris packages lets me do data projection and plotting in different projects fairly easily. Here are some notes for Python and Anaconda which may be useful (the latter might be useful for getting the libraries that NEMO and XIOS need).

4.1.1 Anaconda

Most of these are taken from the official conda manual. The installation for conda (or the lighter version miniconda) is somewhat dependent on the OS and the instructions are here. You end up downloading a bash file that you run in the terminal, and from there you can accept and change some of the settings accordingly. No administrator rights should be required, though it does mean the installed packages may not be shareable. The installation will ask if you want to add to your $PATH variable, which I accepted (it means the some of the anaconda based binaries take precedence over the system ones).

One conda is installed, I would recommend creating an environment so that if damage is to occur, it is only within the environment which may be deleted easily without touching other things. The creation, entering and leaving of the environment is done by:

```
>> julian@psyduck:~/$ conda create -n nemo python=3.6
... 
>> julian@psyduck:~/$
>> julian@psyduck:~/$ source activate nemo
>> (nemo) julian@psyduck:~/$
```

(continues on next page)
The first command creates and environment called nemo that uses python 3.6, and the other commands are self explanatory. An environment may be removed by issuing the command

```
conda remove --name nemo --all
```

Packages are installed through (make sure you are in an environment first)

```
conda install netcdf
conda install -c conda-forge netcdf-fortran
```

Some packages need to be searched for in the forge.

Note that while the environment is active some commands take precedence over others, and a bit of care is needed to make sure the ones you intend to call really are the ones that are called (e.g. my mercurial command hg seems to be overwritten on my machine when I am in my environment). Check with things like which python for example which shows which binary the command python is actually calling.

### 4.1.2 Python

I mostly develop code in a notebook because I am too heavily influenced by MATLAB. Notebooks (in particular with Jupyter) lets you write code within cells that you run and see outputs then and there which is what I am used to. Later on I do write code in a text editor when I have more specific things I want need to do.

I normally do the following to get what I need. Within the environment:

```
conda install scipy
conda install numpy
conda install matplotlib
conda install jupyter
conda install -c conda-forge cartopy
conda install -c conda-forge iris
```

I normally install NetCDF as well. Numpy and scipy gives the number crunching stuff I normally need. Matplotlib gives most of the plotting capabilities. Cartopy and iris are the map and projection packages, and jupyter is the notebook stuff. To trigger the notebook, I normally do from a terminal

```
jupyter notebook 2>/dev/null &
```

just to suppress the terminal outputs. The notebook opens in a browser and you do coding in there (I think there is another software that lets you open and edit notebooks somewhere else though I’ve never used it); it’s basically ipython but in a browser. Note that just closing the tabs does not necessarily close the notebook; you need to do files>>close and halt. Also, just because the relevant pages are closed in the browser does not mean the notebook server is shutdown either; you need to click logout on the top right corner (assuming you are not using a custom theme which suppresses that). To kill it in the terminal, either find the job through jobs and use kill %n or do

```
jupyter notebook list
```

(continues on next page)
4.1.3 Some Python banana skins

The big banana skin with Python to watch out for is that indexing starts at 0 (rather than 1 in MATLAB), and index slicing normally omits the last entry, e.g.

```python
x_vec = [1, 2, 3, 4, 5, 6]
x_vec[0:-1]  # [1, 2, 3, 4, 5]
x_vec[1:4]  # [2, 3, 4]
x_vec[0::]   # [1, 2, 3, 4, 5, 6]
x_vec[-1]   # 6
x_vec[-2]   # 5
```

Contrast this to MATLAB which would be

```matlab
x_vec = [1, 2, 3, 4, 5, 6]
x_vec(0:end-1)  # 1, 2, 3, 4, 5
x_vec(2:4)  # 2, 3, 4
x_vec(:)    # 1, 2, 3, 4, 5, 6
x_vec(end)  # 6
x_vec(end - 1)  # 5
```

Another banana skin with python is that data is not necessarily copied when defining new variables. For example:

```python
x_vec = [1, 2, 3, 4, 5, 6]
y_vec = x_vec
y_vec[0] = 2
y_vec  # [2, 2, 3, 4, 5, 6]
x_vec  # [2, 2, 3, 4, 5, 6]
```

This is especially dangerous if you, like me, do the following in MATLAB:

```matlab
x_vec = zeros(6)
y_vec = x_vec
z_vec = x_vec
```

If you really mean to do a copy, do the following:
from copy import deepcopy
x_vec = [1, 2, 3, 4, 5, 6]
y_vec = x_vec
z_vec = deepcopy(x_vec)
y_vec[0] = 2
y_vec
>> [2, 2, 3, 4, 5, 6]
x_vec
>> [2, 2, 3, 4, 5, 6]
z_vec
>> [1, 2, 3, 4, 5, 6]

Python is really slow with loops, so the more vectorising commands you can use, the better! If you have routines that you have to use loops in (e.g. transformation of data from Cartesian co-ordinates to density co-ordinates through binning into density bins), then consider using cython (write code in C but call it through Python), f2py (same but for Fortran), or numba/JIT (compile and run loops, usually on the order of 200 speed up; restricted to fairly low level commands).

4.2 sphinx notes

(working notes)

- To get bibtex working on ReadTheDocs a requirements.txt may be needed.

Having the following in there got the sphinx-bibtex extension working for me.

```bash
sphinx>>=1.7.0b1
sphinxcontrib-bibtex
```

4.3 git (or bitbucket) commands

testing
CHAPTER 5

Key updates:

- 13 Feb 2019 – added placeholder to *model building* in NEMO
- 04 Jul 2018 – test added a README.rst
- 08 Apr 2018 – initial commit of a “pyCDFTOOLS” with sample Jupyter notebooks
- 16 Mar 2018 – opening of repository as a move to get me to actually do something…
Bibliography


