Installing, compiling, and running MITgcm and its adjoint in the ECCO configuration

for sensitivity studies of oceanic Quantities of Interest (QoIs)

Dafydd Stephenson

June 2022

Contents

1 Introduction

(Basic concepts of an adjoint model, examples of uses, segue into ECCO with an explanation of what it is. The idea of the ECCO configuration vs. the ECCO run, etc.)

2 Reproducing, analysing, and perturbing the ECCO state estimate (without its adjoint)

As any adjoint ocean model is necessarily adjoined to a "regular", forward ocean model (in this case, MITgcm), we will begin by installing and running this model before advancing to its adjoint. As with any model, MITgcm can be run in a number of "configurations": combinations of settings, packages, and coding choices which are geared towards running the model for a specific purpose. Perhaps the most obvious choice made in a configuration is its geometry: higher-resolution configurations are geared towards studying the nature and impact of finer-scale processes, but are typically more demanding to run. Lower-resolution configurations are cheaper and so can be used to run the model on very long timescales or on nonspecialised computers, for example for teaching purposes. The ECCO configuration is designed for adjoint-based state estimation, and therefore has many features which lend themselves to ocean adjoint sensitivity studies.

While a configuration such as ECCO contains specific choices which are "baked in" when the model is compiled (and so are present every time the model is run), there are still further choices to be made at runtime. The default selection of settings when the configuration is downloaded are those which cause the model to reproduce the ECCO state estimate: the model begins running in 1992 and its output reflects a best guess of historical ocean dynamics in the time since. We begin by installing the configuration and conducting this reproduction run in Section 2.1, before then changing some settings such that the model run is the same, but there is more flexibility for the user in Section 2.2. With the model running successfully and producing the desired output, Section 2.3 describes some undocumented ways to process this output. Section 2.4 then describes ways in which the standard ECCO reproduction run can be perturbed, a common technique in sensitivity studies of forward models. In Section 2.5, we lastly consider other runs which can be conducted using the ECCO configuration, including the alternative "flux-forced" version of the reproduction run and a much more general "normal-year" run which allows the study of neutral ocean dynamics without the historical influence of specific events since 1992.

2.1 Installing and running with default settings

This guide assumes the reader is using either NCAR's Cheyenne or NASA's Pleiades supercomputer, though the majority of these instructions can be applied to any high-performance computing (HPC) system. The default settings we will begin with reproduce the ECCO state estimate using 96 CPUs, with 100 output variables at monthly mean frequency. This run takes approximately 24 hours, and so will not be able to finish on Cheyenne with its 12 hour time limit, an issue which is addressed in Sections 2.2.2 and 2.2.3.

2.1.1 Installing MITgcm and the ECCO configuration

In your home space (`cd ~`) get the most recent version of MITgcm which works with ECCO, using

```
git clone https://github.com/MITgcm/MITgcm.git -b checkpoint66g
```

this creates a directory `MITgcm` which contains the source code for the model, including a range of different basic configurations in the subdirectory `MITgcm/verification`. The ECCO configuration is maintained separately by the ECCO Group and can be found elsewhere on GitHub. We'll make a separate subdirectory and get it from there:

```
mkdir -p MITgcm/ECCOV4/release4
cd MITgcm/ECCOV4/release4
git clone https://github.com/ECCO-GROUP/ECCO-v4-Configurations.git
```

```
mv ECCO-v4-Configurations/ECCOv4\ Release\ 4/* .
rm -rf ECCO-v4-Configurations
```

2.1.2 Compiling the model

We now have all the source code necessary to compile MITgcm in the standard ECCOv4r4 configuration. The last ingredient is a text file containing a set of compilation instructions or "build options", which are specific to the computer system in use. A selection of examples are included in `MITgcm/tools/build_options`, but below are two tested examples for Cheyenne and Pleiades, including specific instructions for compiling on those systems. If the below instructions are followed correctly, the executable file `mitgcmuv` will appear at `MITgcm/ECCOV4/release4/build`.

NCAR-Cheyenne: Save the following as a text file, e.g. `MITgcm/tools/build_options/cheyenne`:

```
FC=mpif90
CC=mpicc
F90C=mpif90

DEFINES='-DALLOW_USE_MPI -DALWAYS_USE_MPI -DWORDLENGTH=4'
CPP='/lib/cpp -traditional -P'
EXTENDED_SRC_FLAG='-132'
OMPFLAG='-openmp'
CFLAGS='-fPIC'
LDADD='-shared-intel'

LIBS="-L${MPI_ROOT}/lib"
INCLUDES="-I${MPI_ROOT}/include"

NOOPTFLAGS='-O0 -fPIC'

FFLAGS="-fPIC -convert big_endian -assume byterecl -align -march=corei7 -axAVX"
FDEBUG='-W0 -WB'
FFLAGS="$FDEBUG $FFLAGS"

FOPTIM='-O2'
FOPTIM="$FOPTIM -ip -fp-model precise -traceback -ftz"
```

Before compilation, load the following modules:

```
module load intel/17.0.1 mpt/2.15f netcdf/4.4.1.1
```

it may be worth saving this arrangement as the default set of modules using `module save default`.

We lastly make a directory in which to build the configuration:

```
cd ~/MITgcm/ECCOV4/release4
mkdir build
cd build
```

The actual compilation can now take place using:

```
../../../tools/genmake2 -mods=../code -optfile=../../../tools/build_options/cheyenne -mpi
make depend
make all
```

NASA-Pleiades : Unlike with Cheyenne, an existing option file:

```
MITgcm/tools/build_options/linux_amd64_ifort+mpi_ice_nas
```

exists in the MITgcm source code which can be used to compile on Pleiades, which (at time of writing) works with the following modules loaded:

```
module load comp-intel/2016.2.181 mpi-hpe/mpt.2.23 hdf4/4.2.12 hdf5/1.8.18_mpt netcdf
    /4.4.1.1_mpt
```

We lastly make a directory in which to build the configuration:

```
cd ~/MITgcm/ECCOV4/release4
mkdir build
cd build
```

The actual compilation can now take place using:

```
../../../tools/genmake2 -mods=../code -optfile=../../../tools/build_options/
    linux_amd64_ifort+mpi_ice_nas -mpi
make depend
make all
```

2.1.3 Obtaining files required for the ECCO reproduction run

If the above steps were carried out successfully, the standard ECCO configuration of MITgcm should have compiled as an executable file. The choices particular to each run are specified at execution time using a series of "namelist" files, which tell the model, among other things, where to look for files specifying initial and boundary conditions (surface forcing). To reproduce the ECCO state estimate we require both the namelists and these external condition files.

These files are quite large ($\sim$ 150GB) and thus probably exceed the storage limit for the home space on any given computer system, so should be saved somewhere separately to the source code. The files can be re-downloaded at any time, so don't need to be backed up and are safe on a scratch space. On Pleiades this is the `/nobackup/` space. The Cheyenne space `/glade/scratch/` is subject to routine deletion and so, as the files are needed long term, the space `/glade/work/` is more appropriate.

In the chosen directory (e.g. `/glade/work/YOUR_USERNAME/ECCOv4r4_input/`), three sets of files should be downloaded from the Physical Oceanography Distributed Active Archive Center (podaac). An account is required, which can be created here (link active 2022-06-27, search online for "NASA EarthData Login" otherwise). Note that your password to log into the podaac portal is not the same as the password which is required to retrieve the required files using the terminal. This password can be found by logging into the web interface before clicking "back to WebDAV credentials". Once you have a username and this password, use the following lines of code to obtain the input files:

```
wget -r --no-parent --user YOUR_PODAAC_USERNAME --ask-password \
https://ecco.jpl.nasa.gov/drive/files/Version4/Release4/input_forcing
mv ecco.jpl.nasa.gov/drive/files/Version4/Release4/input_forcing/ .
rm -r ecco.jpl.nasa.gov/
```

```
wget -r --no-parent --user YOUR_PODAAC_USERNAME --ask-password \
https://ecco.jpl.nasa.gov/drive/files/Version4/Release4/input_init
mv ecco.jpl.nasa.gov/drive/files/Version4/Release4/input_init/ .
rm -r ecco.jpl.nasa.gov/
```

2.1.4 Running the model to reproduce the ECCO state estimate

At this stage the model has been successfully compiled and all the necessary files to reproduce the ECCO state estimate have been downloaded. It now remains to make a directory in which to run the model and assemble links to the compiled executable and the necessary run files in this directory. On Cheyenne this will be something like

```
mkdir /glade/scratch/YOUR_USERNAME/ECCOv4r4_reproduction
cd /glade/scratch/YOUR_USERNAME/ECCOv4r4_reproduction

inputdir="/glade/work/YOUR_USERNAME/ECCOv4r4_input"
srcdir="~/MITgcm/ECCOV4/release4"
ln -s ${inputdir}/input_init/xx_*.*.??ta . ;\
ln -s ${inputdir}/input_init/tile00?.mitgrid . ; \
ln -s ${inputdir}/input_init/error_weight/ctrl_weight/*.* . ; \
ln -s ${inputdir}/input_init/total_*.bin . ; \
ln -s ${inputdir}/input_init/smooth* . ; \
ln -s ${inputdir}/input_init/runoff-2d-Fekete-1deg-mon-V4-SMOOTH.bin . ; \
ln -s ${inputdir}/input_init/pickup*.0000000001.??ta . ; \
ln -s ${inputdir}/input_init/geothermalFlux.bin . ; \
ln -s ${inputdir}/input_init/fenty_biharmonic_visc_v11.bin . ; \
ln -s ${inputdir}/input_init/bathy_eccollc_90x50_min2pts.bin . ; \
ln -s ${inputdir}/input_forcing/ . ; \
cp -p ${inputdir}/input_init/tools/mkdir_subdir_diags.py . ; \
cp -p ${srcdir}/namelist/data{,.autodiff,.cal,.cost,.ctrl,\
.ctrl.restart,.diagnostics,.exch2,.exf,.ggl90,.gmredi,\.optim,.pkg,\
.salt_plume,.sbo,.seaice,.smooth} . ; \
cp -p ${srcdir}/namelist/eedata .
```

where `inputdir` is set to wherever you downloaded the reproduction run files in the Section 2.1.3 and `srcdir` is where we installed the configuration. We also need the executable we created in Section 2.1.2:

```
ln -s ${srcdir}/build/mitgcmuv .
```

The directory should now contain a range of files, predominantly symbolic links. Essential to our reproduction run are:

- files beginning `data`, which are plaintext namelists files giving the user control over various aspects of the run. We will not modify any of these settings yet.
- files beginning `pickup` are restart files, which give the model all of the information about the state of the ocean at the beginning of the run, before ECCO's "control" adjustments are made
- files beginning `xx_` are ECCO's initial control adjustments. These apply a small perturbation to the ocean state in the `pickup` files in order to push the modelled ocean onto the trajectory which is closest to the historically obvserved ocean.
- files in `input_forcing` are forcing files which act to keep the model on this trajectory during the run

We have a few small adjustments to make to the default set-up before running the model. By default, the ECCOv4r4 configuration is set up to not only reproduce the ECCO state estimate, but to *improve* it via an iterative assimilation procedure. As we are only interested in the reproduction run, our use-case is substantially simpler.

```
sed -i -e "s#eccov4r4_#input_forcing/eccov4r4_#g" data.exf; \
sed -i -e "s#useProfiles=.TRUE.#useProfiles=.FALSE.#g" data.pkg; \
```

```
printf "&ECCO_COST_NML /\n&ECCO_GENCOST_NML /" > data.ecco ; \
python2 mkdir_subdir_diags.py
```

Line-by-line:

- Line 1 adjusts the namelist controlling settings related to external forcing (`data.exf`). By default, the model looks in the run directory for the forcing files, but there are hundreds of these, so we have instead added a symbolic link to the directory `input_forcing` in which they are kept, and are now updating this namelist to tell it to look inside this directory instead.

- Line 2 adjusts the namelist controlling which packages are used (`data.pkg`). The "profiles" package allows assimilation of profiling measurements made by CTDs, Argo floats, gliders, and so on. As we are not interested in assimilation, we turn this package off.

- Line 3 creates an effectively empty `data.ecco` namelist file. Usually this namelist would tell the model where to look for the observations to be assimilated into the run (see the default `data.ecco` in `input_init/NAMELIST`), but, again, we are not interested in the assimilation procedure, so tell the model to do nothing instead.

- Line 4 runs a python script which creates a location (the directory `diags` and its subdirectories) for the model output to be saved. The default settings request monthly average output of 100 variables. We will adjust which variables are saved, how often, and where, in Section 2.2.1.

Lastly, to run the model, we require a job submission script. These are again plain text files which specific to the computer system being used:

NCAR-Cheyenne Create a file `ECCOv4r4_reproduction.pbs` containing the following lines:

```
#PBS -S /bin/bash
#PBS -A PROJECT_CODE
#PBS -l select=6:ncpus=16:mpiprocs=16
#PBS -l walltime=12:00:00
#PBS -q regular
#PBS -j oe
#PBS -o ECCOv4r4_reproduction.out
#PBS -m bea

module purge
module load intel/17.0.1 mpt/2.15f netcdf/4.4.1.1
export TMPDIR="/glade/scratch/YOUR_USERNAME/temp"
mkdir -p "${TMPDIR}"

mpiexec_mpt dplace -s 1 ./mitgcmuv
```

The various options at the beginning are documented in the manual page for `qsub` . `TMPDIR` is created in scratch space to prevent the home space from filling up during the run.

The job can be submitted using `qsub ECCOv4r4_reproduction.pbs`

NASA-Pleiades Create a file `ECCOv4r4_reproduction.pbs` containing the following lines:

```
#PBS -S /bin/bash
```

```
#PBS -l select=4:ncpus=24:mpiprocs=24:model=has
#PBS -q long
#PBS -l walltime=24:00:00
#PBS -j oe
#PBS -o ECCOv4r4_reproduction.out
#PBS -m bea
#PBS -M YOUR@EMAIL.ADDRESS

module load comp-intel/2016.2.181 mpi-sgi/mpt.2.14r19 hdf4/4.2.12 hdf5/1.8.18_mpt netcdf
    /4.4.1.1_mpt

mpiexec_mpt -np 96 ./mitgcmuv
```

The job can be submitted using `qsub ECCOv4r4_reproduction.pbs`

If the run is successful, the run directory will populate with a number of new files. We will ultimately be interested in those which appear in the `diags` folder, which are the model output, as well as the files `STDOUT.0000` - `STDOUT.0095`, which is the standard output of the model for each of the 96 processors on which we ran the model. While the model is still running, these provide a live update as to what is happening which can be followed using `tail -f STDOUT.0000`. The companion `STDERR` files contain any standard error messages, while `ECCOv4r4_reproduction.out` contains any output returned to the job scheduler. This is typically not updated until the job is finished, however.

So far we have purely followed the default off-the-shelf settings for reproducing the ECCO state estimate, with no control over, for example which variables are returned by the model, or how often. Before analysing the output, it would be useful to make sure the output we are analysing is useful to us, and so the next section describes how to run the model again with more customised runtime choices.

2.2 Running with customised settings

2.2.1 Changing the output variables and frequency

The first change we'll make to the default run settings is the `data.diagnostics` namelist, which controls which variables are returned by the model and how often. The default namelist contains 100 entries, all monthly averages, but typically we are only interested in a handful of ocean diagnostics and may require higher frequency output. The entries in the ECCOv4r4 default namelist specify the output frequency (`frequency`), the variable of interest (`fields`) and the location to save the output to (`filename`). For instance, the 70th entry is monthly-averaged ocean temperature:

```
#---
frequency(70) = 2635200.0,
fields(1,70) = 'THETA',
filename(70) = 'diags/THETA_mon_mean/THETA_mon_mean',
#---
```

A positive value (in seconds) for `frequency` returns the average value during that period; a negative value returns a snapshot at the end of the period. `THETA` is the model name given to potential temperature. A general list of the available fields can be found in the MITgcm manual here, or a list specific to our configuration in the file `available_diagnostics.log` in the run directory.

As an example, the following `data.diagnostics` returns only the five primitive ocean variables as daily snapshots, and all in the same directory. We will look at how to read and process this output in Section 2.3.

```
 &diagnostics_list
#
    dumpatlast = .TRUE.,
#---
frequency(1) = -86400.0,
fields(1,1) = 'THETA',
filename(1) = 'diags/THETA'
#---
frequency(2) = -86400.0,
fields(1,2) = 'SALT',
filename(2) = 'diags/SALT'
#---
frequency(3) = -86400.0,
fields(1,3) = 'UVEL'
filename(3) = 'diags/UVEL'
#---
frequency(4) = -86400.0,
fields(1,4) = 'VVEL'
filename(4) = 'diags/VVEL'
#---
frequency(5) = -86400.0,
fields(1,5) = 'WVEL'
filename(5) = 'diags/WVEL'
#---
  /
```

2.2.2 Running with 192CPUs

Running with twice as many processors allows the run to progress approximately twice as quickly. If running on the Cheyenne HPC system, it is almost certain that the reproduction run, requiring around 24 hours, did not finish before encountering the 12 hour time limit placed on jobs. It is possible to get around this problem by restarting the run as a new job half way through (Section 2.2.3), but increasing the number of processors is much more time-efficient. This requires us to make some modifications to the source code of the configuration and recompile, however.

Two modifications are needed to tell the model to run on 192 processors instead of 96: in the directory `~/MITgcm/ECCOV4/release4/code`, the file `SIZE.h` specifies how the grid is distributed among the 96 processors. Make a copy of this file (something like `SIZE.h_96cpus` and replace it with the 192 CPU version found here. The model can then by recompiled as in Section 2.1.2.

Before running, the namelist file `data.exch2` should also be modified. ECCO is run on a "cubed sphere" (Section 2.3) which, like a typical cube, has "faces". These faces must exchange information with each other and this namelist describes where this happens, which is different for a 192 CPU configuration. Replace `data.exch2` in the run directory with the version found here.

2.2.3 Interrupting and restarting the ECCO reproduction run

If it is necessary to run the model with the default 96 CPUs, we can execute the ECCO reproduction run as two (or more) jobs. As with many models, MITgcm has a restart feature, which periodically (every year by default) saves all the information about the ocean state necessary into "pickup" files, to continue the run from that point.

To utilise this feature we'll make a simple adjustment to the default settings. We'll run for 13 years instead of the full 26 (from January 1992 to January 2005), then restart the run in January 2005. To do this, we'll adjust the main namelist file, `data`, changing the line `nTimeSteps=227903` (26 years) to read `nTimeSteps=114048` (13 years + 7 days padding).

As we'll run two separate jobs, we'll copy the file `ECCOv4r4_reproduction.pbs` which we created in Section 2.1.4 to `ECCOv4r4_reproduction_part1.pbs` and submit it as before. When the run is complete, the directory should contain the files

```
pickup.0000113964.data
pickup.0000113964.meta
pickup_ggl90.0000113964.data
pickup_ggl90.0000113964.meta
pickup_seaice.0000113964.data
pickup_seaice.0000113964.meta
```

which will be used to restart the run.

We now need to tell the model to start from this time step (113964) instead of step 1. Modify the main namelist file such that instead of `nIter0=1` reads `nIter0=113964`. We will also change the number of time steps to `nTimeSteps=113940`.

Lastly, the ECCO state estimate works by adjusting the state of the ocean when the run begins in 1992, but if we don't turn this feature off, it will apply these same adjustments in 2005. By replacing (be sure to make sure you have a backup copy first!) the file `data.ctrl` with the file `data.ctrl.restart`, we prevent these adjustments from being made.

Lastly, copy the original submission script to `ECCOv4r4_reproduction_part2.pbs` and submit the job as before.

2.3 Analysing the output

The raw model output is in binary format, in file pairs `X.data`, `X.meta` known as MDS. We will predominantly process the output using tools in the `ecco_v4_py` package, with installation instructions described here. Familiarity with the `ECCOv4-py` package is assumed, and a thorough tutorial is available here. One additional file which will prove essential is `ECCO-GRID.nc` which contains information about the LLC90 grid, and can be found here.

Additionally, the python package `xmitgcm` will be of use. The package can be installed through anaconda using `conda install -c conda-forge xmitgcm` or using other methods described here.

This section will describe useful ways to process the model output, some of which are not described in the `ecco_v4_py` tutorial.

2.3.1 Converting standard model output to netCDF

The simplest and most robust way to convert the output is using `xmitgcm`'s `open_mdsdataset` function, which returns an `xarray` dataset that can be saved as a netCDF file using the `to_netcdf` operation. However, I have found this approach to be extremely slow for large amounts of output and personally prefer a combination of `ECCOv4-py`'s `read_llc_to_tiles` and `llc_tiles_to_xda` functions. A test while writing this document found the `xmitgcm` approach to take around 30 minutes for around 200 monthly outputs of a single variable, vs. about 30 seconds using the latter approach. Based on this approach, a python script is included in Appendix A.1 which accepts either a list of variable names or reads the `data.diagnostics` namelist to ascertain what model output has been produced, then saves a netCDF file of the output in the same location. This python script can be called at the end of job script `ECCOv4r4_reproduction.pbs` so that the model output is converted immediately after the run ends.

Examples Only convert potential temperature and salinity outputs to netCDF:

```
python mds_to_netcdf.py THETA SALT
```

Or just convert all variables in `data.diagnostics` to netCDF:

```
python mds_to_netcdf.py
```

2.3.2 The LLC90 grid and interpolation onto a regular lat-lon grid

On the ECCO data portal, data are available on the native LLC90 grid but also interpolated onto a regular $0.5° \times 0.5°$ grid. Running the model only returns the native grid output, however. Again, `ecco_v4_py` has a routine which can address this: `resample_to_latlon`.

Note that `resample_to_latlon` only operates on 2D data, and so for time-varying data on multiple vertical levels, it is necessary to call the function multiple times in a loop, which can be extremely time consuming. In the below example, the monthly temperature output that we converted to netCDF in Section 2.3.1 is opened, interpolated onto a regular grid, and saved to a different netCDF file:

```
import xarray as xr
import xmitgcm as xm
import ecco_v4_py as ecco
################################################################################
DS=xr.open_dataset('diags/THETA_mon_mean/THETA_mon_mean.nc')
varname='THETA'
VAR=DS[varname]
################################################################################
GDS=xr.open_dataset('~/ECCO_GRID.nc')
# from https://ecco.jpl.nasa.gov/drive/files/Version4/Release4/nctiles_grid
llcX=GDS.XC
llcY=GDS.YC
llcZ=GDS.Z
################################################################################

# Initialise empty array
Nt=len(DS.time)
VARi=np.zeros((Nt,50,360,720))
# Loop over time index and depth index
for t in np.arange(Nt):
    print(t)
    for k in range(50):
        XCi,YCi,XGi,YGi,VARi[t,k,:]=ecco.resample_to_latlon(llcX,llcY,VAR[t,k,:],\
            -90,90,0.5,-180,180,0.5,mapping_method='nearest_neighbor',\
                                                    fill_value=np.nan)

# Create xarray Dataset with same attributes as the MDS output
DSi=xr.Dataset(\
        data_vars={varname:(['time','depth','latitude','longitude'],VARi)},\
        coords={'latitude':YCi[:,0],'depth':llcZ.values,'longitude':XCi[0,:],\
               'time':DS.time},\
        attrs=DS.attrs)
DSi[varname]=DSi[varname].assign_attrs(VAR.attrs)

#Save to netCDF

DSi.to_netcdf('diags/THETA_mon_mean/THETA_mon_mean_interpolated.nc')
```

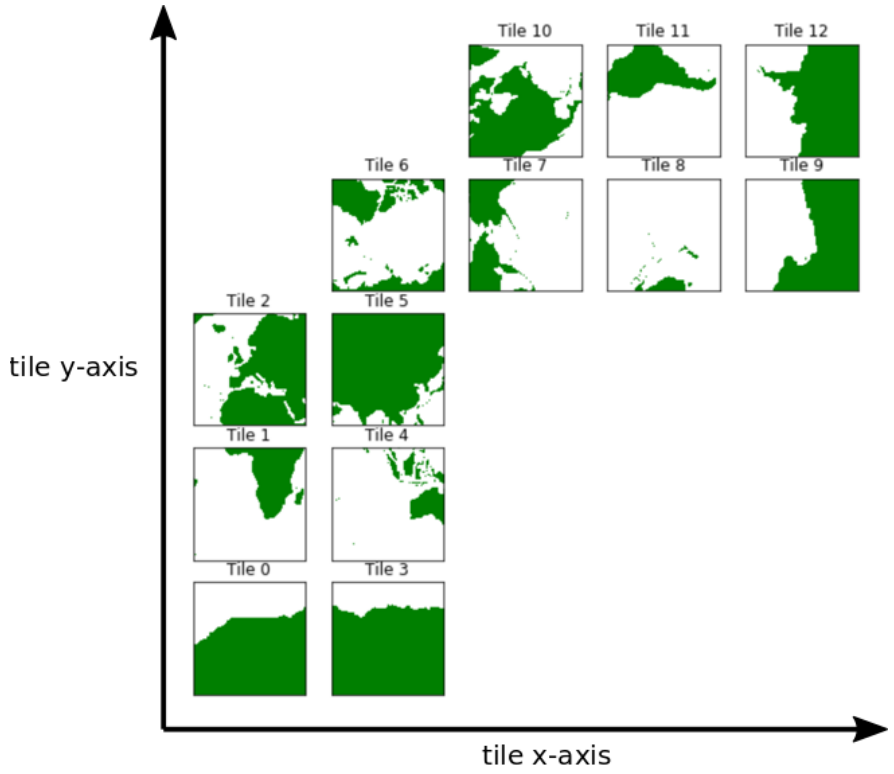

Figure 1: Orientation of the 13 tiles of the LLC90 grid

2.3.3 Re-orienting velocity (and other directional) fields

A particularly unintuitive feature of the LLC90 grid is that, depending on which of the 13 "tiles" is being considered (Figure 1), the North-South direction may align with either the x-axis (tiles 0-5), the y-axis (tiles 7-12) or both (tile 6, the Arctic "cap"). This means that the x-oriented component of the velocity ("u") is not strictly the same as the zonal velocity. An `ecco_v4_py` routine again exists to handle this, `vector_calc.UEVNfromUXVY`, taking model-native fields with this idiosyncrasy and returning more intuitive zonal and meridional fields recast onto the grid's "C"-points.

As an example, we'll open the monthly mean velocity fields `UVEL` and `VVEL` that we converted to netCDF in Section 2.3.1, reorient them using `ecco.vector_calc.UEVNfromUXVY`, then show the input and output fields in a before-after comparison:

```
import xarray as xr
import ecco_v4_py as ecco
import matplotlib.pyplot as plt
UVEL=xr.open_dataset('diags/UVEL_mon_mean/UVEL_mon_mean.nc')
VVEL=xr.open_dataset('diags/VVEL_mon_mean/VVEL_mon_mean.nc')
GDS=xr.open_dataset('~/ECCO_GRID.nc')

zon_vel,mer_vel=ecco.vector_calc.UEVNfromUXVY(UVEL.UVEL,VVEL.VVEL,GDS)
```

Comparing the input and output (Figure 2), we see that simply plotting the native model output onto a standard map results in undesirable behaviour for fields such as velocity, but that this can be "corrected" using appropriate routines. Code to plot Figure (2):

```
fig,ax=plt.subplots(2,2)
ecco.plot_proj_to_latlon_grid(GDS.XG,GDS.YC,UVEL.UVEL[0,0,:],subplot_grid=[2,2,1],cmin
    =-1,cmax=1,cmap='seismic');
plt.gca().set_title('x-oriented velocity')
```

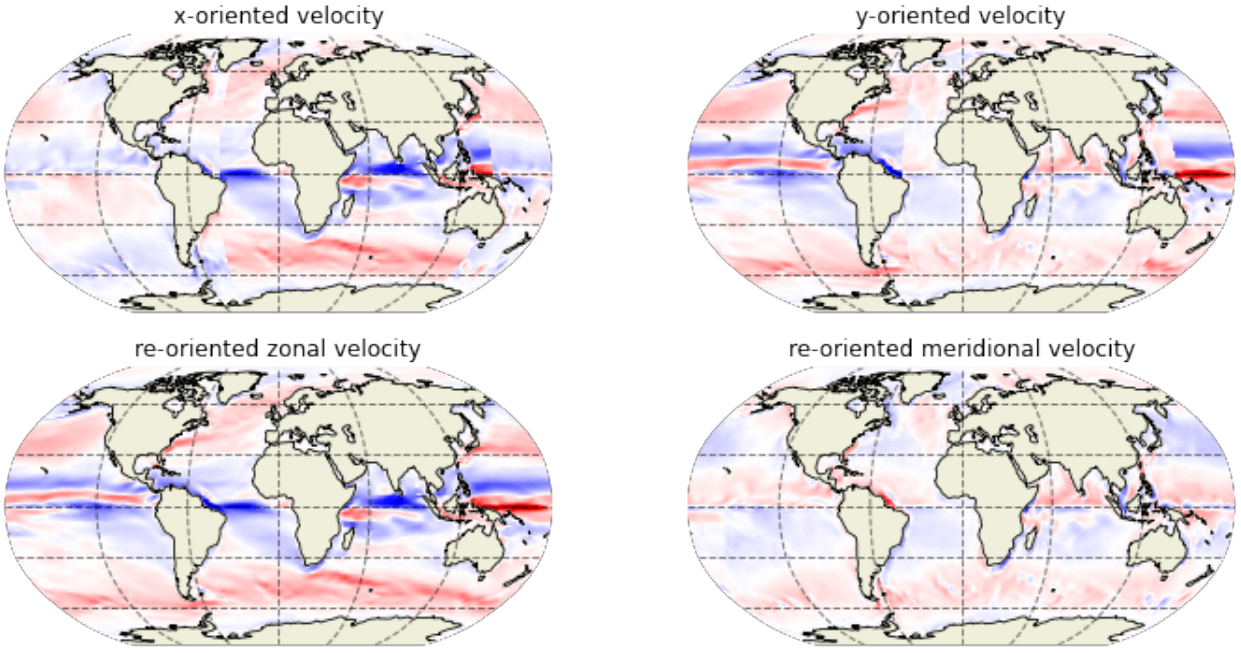

Figure 2: Monthly mean velocity fields for January 1992 before (top row, native LLC90) and after (bottom row) reorientation.

```
ecco.plot_proj_to_latlon_grid(GDS.XC,GDS.YG,VVEL.VVEL[0,0,:],subplot_grid=[2,2,2],cmin
    =-1,cmax=1,cmap='seismic');
plt.gca().set_title('y-oriented velocity')
ecco.plot_proj_to_latlon_grid(GDS.XC,GDS.YC,zon_vel[0,0,:],subplot_grid=[2,2,3],cmin=-1,
    cmax=1,cmap='seismic');
plt.gca().set_title('re-oriented zonal velocity')
ecco.plot_proj_to_latlon_grid(GDS.XC,GDS.YC,mer_vel[0,0,:],subplot_grid=[2,2,4],cmin=-1,
    cmax=1,cmap='seismic');
plt.gca().set_title('re-oriented meridional velocity')
fig.set_size_inches(12,6)
```

2.3.4 Subsetting output for cheaper diagnostics

In many cases, we are only interested in a specific region of the ocean, and, typically, only points not on land. For memory-intensive diagnostics such as EOF analysis, it is therefore much cheaper to throw away any locations we are not interested in, perform the diagnostics on a heavily reduced dataset, and then, if desired, restore the results back to full size (for plotting, etc.).

Example As an example, we'll look at an EOF decomposition of surface pressure, with an interest in extracting the typical pattern associated with the North Atlantic Oscillation. We'll use the `EXFpress` variable which is by default produced by the ECCO reproduction run, assuming it was converted to netCDF format following section 2.3.1. We'll begin by using the full, global output, before reducing our focus to the Atlantic.

We first open the output and calculate the anomaly from climatology using `xarray`'s `groupby` functionality. This approach is quite slow relative to manually calculating and extracting the climatology, but has the advantage of simpler code:

```
import numpy as np
```

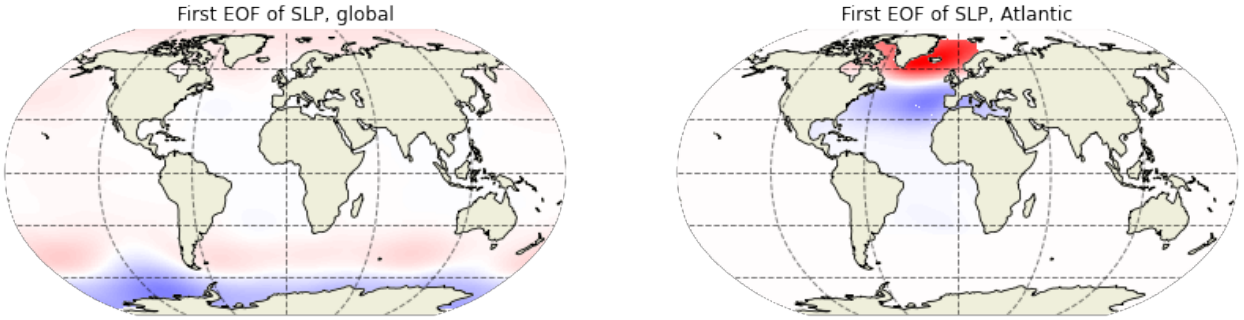

Figure 3: Leading EOFs of the `EXFpress` variable. Left: global; Right: Atlantic only. The Right panel shows the North Atlantic Oscillation pattern, as desired, and resulted in a far lower demand on computational resources given that unnecessary gridpoints were discarded for the calculation.

```
import xarray as xr
PDS=xr.open_dataset('diags/EXFpress_mon_mean/EXFpress_mon_mean.nc')
Pgb=PDS.EXFpress.groupby('time.month')
P_anom=(Pgb-Pgb.mean('time')).drop_vars('month')
```

We then reshape the anomaly values into a space x time data matrix, and calculate its singular value decomposition: the returned matrix "`Vh`" consists of the EOFs, the first of which is shown in the left panel of Figure 3. Note that it is not advisable to replicate this step in the global case due to the extreme demands on memory (250GB) and long computation time.

```
X=P_anom.values.reshape(len(P_anom.time),-1)
U,s,Vh=np.linalg.svd(X)
```

We now repeat this procedure, but restrict the analysis to at-sea gridpoints in the Atlantic Ocean north of 35°S. The total number of gridpoints reduces from 105300 to 10469, and the calculation is over one hundred times faster, ultimately producing a more useful Atlantic-focused result which resembles the North Atlantic Oscillation pattern.

Firstly, we create a list of the gridpoints we are interested in, using the `ECCO-GRID.nc` file:

```
import numpy as np
import xarray as xr
import ecco_v4_py as ecco
GDS=xr.open_dataset('~/ECCO-GRID.nc')

# Use the ecco_v4_py routine to get a binary mask of surface 'C' points in the Atlantic
AmskC=ecco.get_basin_mask('atlExt',GDS['maskC'].isel(k=0),less_output=True)

# Set points South of 35S to 0
AmskC=AmskC.where(GDS['YC'].values>-35,0)

# Get a 1D list of non-zero indices
Ci_atl=np.argwhere(AmskC.values.flatten()>0).squeeze()
```

Next, we repeat the above approach of singular value decomposition, but only include the 10469 gridpoints in our list:

```
X=P_anom.values.reshape(len(P_anom.time),-1)[:,Ci_atl]
U,s,Vh=np.linalg.svd(X)
```

When testing the code to write this documentation, the inclusion of `[:,Ci_atl]` reduced the computation time from 8.5 minutes to 4 seconds. Furthermore, by restricting the analysis to the Atlantic sector, the result is not influenced by regions which are not important to us, leading to the first EOF resembling the NAO pattern in which we are interested.

Note that in order to plot Figure 3, it was first necessary to restore the result back to the full grid:

```
#Make a 1D array of zeros with enough points to cover the full 13x90x90 surface grid
EOF1_SLP_atl=np.zeros(13*90*90)

#Fill that array at the points of interest using the first singular vector / EOF
EOF1_SLP_atl[Ci_atl]=Vh[0,:]

# Reshape the 1D array back to the usual grid shape
EOF1_SLP_atl=EOF1_SLP_atl.reshape(13,90,90)

# If desired, plot using ecco_v4_py's plot_proj_to_latlon_grid
ecco.plot_proj_to_latlon_grid(GDS.XC,GDS.YC,EOF1_SLP_atl,cmin=-0.05,cmax=0.05,cmap='
    seismic');
```

2.4 Perturbing the ECCO state estimate

The primary feature of the ocean adjoint model, to which we are building, is to describe sensitivity of the ocean system to change. In the context of a non-adjointed ocean model, or to verify the results of an ocean adjoint model, sensitivity analysis in a "forward" model typically involves applying a small change to an ocean model, and then comparing model runs with and without this perturbation. As we will see later, the advantage of an ocean adjoint model is that it allows (with some limitations) every possible perturbation to be considered at once. In a regular "forward" ocean model, each perturbation has to be applied separately, and only a sample of the possible perturbations can be obtained. Nevertheless, this can be a powerful way to understand ocean dynamics. In this section we'll look at ways to perturb the initial and boundary conditions (i.e., forcing) of the ECCO state estimate.

2.4.1 Perturbing initial conditions

The ocean state from which MITgcm restarts is preserved exactly in the `pickup` files. This state may be perturbed using the `ctrl` package. As described previously, this is how ECCO pushes the ocean state onto a path that more closely matches ocean observations, before maintaining that path using specially constructed surface forcing. The initial control adjustments we applied to the ECCO reproduction run are contained in the files:

```
xx_diffkr.0000000129.data
xx_kapgm.0000000129.data
xx_kapredi.0000000129.data

xx_etan.0000000129.data
xx_salt.0000000129.data
xx_theta.0000000129.data
xx_uvel.0000000129.data
xx_vvel.0000000129.data
```

The first three adjust diffusion parameterisations, while the rest respectively handle the primitive ocean variables: sea surface height, salinity, temperature, x-oriented velocity and y-oriented velocity. It should be noted that these adjustments are *unitless* and that the model applies a smoothing and rescaling during the run. Thus we cannot easily apply a dimensional perturbation (e.g. "perturb temperature by 1K everywhere in the Irminger Sea").

Instead, we can tell the model to use alternative versions of the control adjustments which have the smoothing

and rescaling already applied. As with the other ECCO input files from Section 2.1.3, these are available from podaac:

```
inputdir="/glade/work/YOUR_USERNAME/ECCOv4r4_input"
cd ${inputdir};
wget -r --no-parent --user YOURUSERNAME --ask-password \
https://ecco.jpl.nasa.gov/drive/files/Version4/Release4/other/adjustments;
mv ecco.jpl.nasa.gov/drive/files/Version4/Release4/other/adjustments/xx_*.dim.*.data .;
```

Now make a new directory in which to run ECCOv4r4 (as in Section 2.1.4) with a name such as `ECCOv4r4_perturbed`, noting that alongside the files listed above should be dimensional counterparts, e.g. `xx_theta.dim.0000000129.data`. We need to now adjust the `ctrl` package namelist file, `data.ctrl`, to tell it to use these files instead and not apply any smoothing or rescaling to them. Make a backup copy of the original `data.ctrl`, then

1. Edit `data.ctrl` so that any lines containing `xx_<VAR>` are modified to point to the dimensionalised versions, `xx_<VAR>.dim`

2. Edit `data.ctrl` so that any "preprocessing" lines (`xx_genarr?d_preproc`) are changed from `WC01` to `noscaling`.

Without any further changes, the model should now run exactly as before. What we desire instead is to edit these initial control adjustments so that a perturbation is applied.

Example As an example, we shall apply a freshening anomaly of 0.5 psu (roughly equivalent to the Great Salinity Anomaly of the 1980s;REF) to the Labrador Sea. There is no basin mask in the `ecco_v4_py` package for the Labrador Sea, so we'll define it as the portion of the Atlantic basin mask North of 52.5° N (approximately Cape St. Lewis) and West of 45° W (approximately Cape Farewell):

```
import xarray as xr
import ecco_v4_py as ecco
GDS=xr.open_dataset('~/ECCO-GRID.nc')

# Get binary mask of the surface Atlantic
labsea_mask=ecco.get_basin_mask('atl',GDS.hFacC,less_output=True).isel(k=0)

# Throw away points outside our chosen Labrador Sea region
labsea_mask=labsea_mask-labsea_mask.where((GDS.XC.values>-45) | (GDS.YC.values<52.5),0);
```

This mask is 1 in our chosen region and 0 elsewhere. In our new run directory, we now load in the dimensional salinity adjustment and subtract 0.5 psu in the region (Figure 4).

```
xx_salt_dim=ecco.llc_tiles_to_xda(\
            ecco.read_llc_to_tiles('.','xx_salt.dim.0000000129.data',nk=50)\
           ,var_type='c',dim4='depth')
xx_salt_mod=xx_salt_dim.copy()
xx_salt_mod[0,:,:,:]=xx_salt_dim[0,:,:,:]-(0.5*labsea_mask)
```

To write this to MDS files which can be read by the model, we follow the example in the `xmitgcm` documentation. Note that in the second step we rename the "tile" dimension (preferred by `ecco_v4_py`) to the less accurate "face" (preferred by `xmitgcm`):

```
import xmitgcm as xm
extra_metadata = xm.utils.get_extra_metadata(domain='llc', nx=90)

# Create a dictionary of 5 entries, one for each cubed sphere face
```

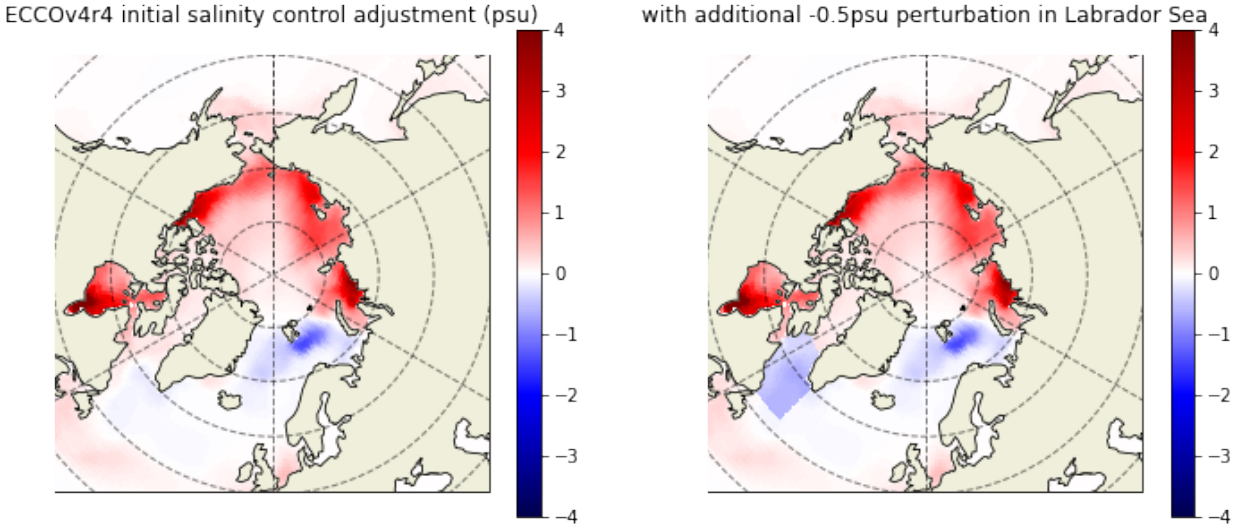

Figure 4: Initial ECCOv4r4 dimensional salinity adjustments, surface layer. Left: standard ECCOv4r4; Right: ECCOv4r4 with a 0.5psu freshening anomaly applied in the Labrador Sea

```
facets = xm.utils.rebuild_llc_facets(xx_salt_mod.rename({'tile':'face'}), extra_metadata)

# Flatten the information into one long array
compact = xm.utils.llc_facets_3d_spatial_to_compact(facets, 'k', extra_metadata)

# Write the flattened array to a binary/MDS file
xm.utils.write_to_binary(compact, 'xx_salt_mod.dim.0000000129.data')
```

Note that the above procedure is not the same if we are writing a 2D file (such as `xx_etan.0000000129.data`). In particular, `llc_facets_3d_spatial_to_compact` becomes `llc_facets_2d_to_compact`, and is called without the `'k'` argument. Lastly, we replace `xx_salt.dim.0000000129.data` in the `data.ctrl` namelist with `xx_salt_mod.dim.0000000129.data`, and can re-run the model as before.

2.4.2 Modifying forcing files

Modifying the surface control adjustments (i.e. forcing) is a more involved procedure than the initial control adjustments, for a few reasons. Firstly, unlike the initial control adjustments, they are not applied separately. Whereas the `ctrl` package allows adjustments (in the `xx_` files) to be made relative to the ocean state from which the model is restarted (in the `pickup` files), the unadjusted forcing is not kept separate from the forcing adjustments: they are all written into one file. Adding further perturbations to the existing adjustments is therefore nontrivial. Secondly, while the initial control adjustments are instantaneous, the forcing is time-varying, using a real calendar. We therefore have to account for leap years, as well as the fact that the `xmitgcm` routines in the example of Section 2.4 cannot write time-varying MDS files to be written by the model. We will have to use a custom routine to save our modified forcing files.

Example: Replacing the ECCO wind stress forcing with its climatology In this example we'll calculate a climatology from the ECCO wind stress forcing files and then write a new set of wind stress forcing files consisting only of this climatology. This is a relatively straightforward yet versatile example: in more complicated cases we may wish to modify the forcing anomalies, for instance by removing an EOF pattern. Calculating the climatology is a necessary first step to any modifications involving these anomalies.

Firstly, we initialise import necessary modules and initialise a zero array which we will fill with the forcing climatology, as well as other variables which will allow us to look up how many forcing entries are present in

a given year (4 per day corresponding to 1460 in most years, 1464 in leap years, and 1459 in the special case of 2017 where the final forcing entry is missing):

```
import numpy as np
import xmitgcm as xm
import ecco_v4_py as ecco

varname='ustr' # The forcing variable of choice
indir ='input_forcing' # Where to read the ECCOv4r4 forcing
outdir='input_forcing_modified' # Where to write the modified forcing

# Initialise empty arrays for forcing climatology
forc_array_clim=np.zeros((366*4,13,90,90))

# Number of years/leap years to average over
nyears=len(range(1992,2018))
leaps=[1992,1996,2000,2004,2008,2012,2016]
nleaps=len(leaps)

# Number of forcing entries for each year
nentries = dict.fromkeys(leaps, 1464)
nentries.update(dict.fromkeys([y for y in range(1992,2017) if (y not in leaps)], 1460))
nentries[2017]=1459
```

Next we'll build the climatology by reading in the forcing files year by year and adding them to the running average:

```
# Loop over years, importing data and adding to the average
for YR in range(1992,2018):

    # Filename corresponding to forcing (e.g. eccov4r4_ustr_1992)
    fname=('eccov4r4_'+varname+'_'+str(YR).zfill(4))

    # Read the forcing to an xarray DataArray
    forc_array=ecco.llc_tiles_to_xda(\
               ecco.read_llc_to_tiles(indir,fname=fname,nl=nentries[YR])[:,0,:,:,:],\
                                var_type='c',dim4='time')

    # Update climatology where entries for this year exist:
    if nentries[YR]>=1459:
        forc_array_clim[:1459,:]=forc_array_clim[:1459,]+(forc_array[:1459,]/nyears)

    # For all years except 2017 (final forcing entry missing in 2017)
    if nentries[YR]==1460:
        forc_array_clim[1459,:]=forc_array_clim[1459,]+(forc_array[1459,]/(nyears-1))

    # For all leap years
    if nentries[YR]==1464:
        forc_array_clim[1460:1464,]=forc_array_clim[1460:1464,]+\
                                        (forc_array[1460:1464,]/nleaps)
```

Notice that we specify `var_type='c'` (for 'center') when reading the forcing, even though we are loading the "u" direction wind stress, which we might expect to be of type 'w' (for 'west') and therefore subject to the issues related to the velocity gridpoints described in Section 2.3.3. However, in the eternal interest of making things as confusing as possible, the standard ECCOv4r4 *forcing* files are on the A-grid, i.e. all variables are read from the same location, in the centre of the grid cell, before being mapped to the staggered grid on which the model output is returned. This is specified by two namelist options in `data.exf`:

```
readStressOnAgrid = .TRUE.,
rotateStressOnAgrid = .TRUE.,
```

This point is not essential to this example, but is worth keeping in mind when working with ECCOv4r4 input rather than output.

We can now write modified forcing files containing only this climatology. Note that in order to do this we need to define a function based on the `xmitgcm` function `xmitgcm.utils.llc_facets_2d_to_compact` (which only works with spatially two-dimensional variables that do not vary in time) allowing us to write time-varying surface data:

```
# Define a function which converts time-varying spatially 2D data to compact form
def llc_facets_3d_temporal_to_compact(facets,extra_metadata):
    for kfacet in range(len(facets)):
        if facets['facet' + str(kfacet)] is not None:
            if 'time' in facets['facet' + str(kfacet)].dims:
                tlen=len(facets['facet' + str(kfacet)].time)
                tmp = np.reshape(facets['facet' + str(kfacet)].values, (tlen,-1))
        if kfacet==0: flatdata=tmp.copy()
        else: flatdata = np.hstack([flatdata, tmp])

    flatdata=flatdata.reshape(-1)

    return flatdata

# Load metadata necessary for xmitgcm writing routines
extra_metadata=xm.utils.get_extra_metadata(domain='llc',nx=90)

# Loop over years again, this time writing to modified forcing files
for YR in range(1992,2018):
    facets=xm.utils.rebuild_llc_facets(var_clim[:nentries[YR],:],extra_metadata) #
        Rearrange the data for xmitgcm
    compact=llc_facets_3d_temporal_to_compact(facets,extra_metadata=extra_metadata)
    xm.utils.write_to_binary(compact,\
    output_dir+'eccov4r4_'+variable_name+'_clim_'+str(YR).zfill(4))
```

Lastly, we can repeat this for the other wind stress variables `vstr`, and, optionally `wspeed`, before modifying the forcing namelist `data.exf` to tell it to read our modified forcing files.

For example, the line

```
ustressfile = 'input_forcing/eccov4r4_ustr',
```

becomes

```
ustressfile = 'input_forcing_modified/eccov4r4_ustr_clim',
```

and so on.

2.5 Running with different forcing sets

Before moving on to the use of the adjoint model, we note that, for the purposes of adjoint sensitvity analysis, the standard ECCOv4r4 configuration possesses some traits which can lead to ambiguities in the interpretation of adjoint sensitivities.

Firstly, the forcing variables in ECCOv4r4 are interconnected atmospheric variables (such as air temperature and humidity) which are passed through bulk formulae to be converted into quantities such as heat flux

which directly impact the ocean. Thus, while an ocean quantity may be sensitive to heat flux, unambiguously attributing this to a change in atmospheric variables is nontrivial. We will address this in Section 2.5.1.

Secondly, ECCOv4r4 is, by nature, a nonstationary state estimate. In a year of anomalously deep convection, for example, the ocean may be particularly sensitive to changes in the convection region, and this may not reflect the "typical" behaviour of the ocean in other years. So we may well use the ECCO state to describe, for example, the sensitivity of ocean heat content *in 2015* to earlier changes, this does not strictly translate to the sensitivity of heat content *in general* to earlier changes. We will address this in Section 2.5.2.

2.5.1 Flux-forced ECCO

In the example of Section 2.4.2, where the wind stress forcing was replaced with its climatology, it was briefly noted that the user could choose whether to replace just the zonal and meridional wind stress fields, or also the wind speed. Removing the wind speed in this case would directly affect the turbulent heat flux, which is linearly proportional to wind speed. As described in Section 2.5 above, this sort of behaviour leads to complications when attempting to unambiguously attribute ocean changes to surface changes. To avoid this issue, the *flux-forced* version of ECCOv4r4 was created. This leads to the same state estimate to the one we have worked with up to this point, but with a different set of forcing variables (and settings).

Note that when it does come to using the adjoint model, we are still able to obtain sensitivities to any forcing variable, from either version of ECCOv4r4. This configuration merely changes the way the forward model is run, which may be useful if, e.g., we want to use information determined from adjoint sensitivities to modify the forcing in the forward run, as in Section 2.4.2.

Each of the forcing variables in flux-forced ECCOv4r4 directly impact one of either the rate of change of ocean temperature, salinity, or velocity, without impacting the others. These variables are

Temperature:

- `hflux`: Net (latent + sensible + longwave + shortwave) heat flux (Wm^{-2}), positive out of the ocean
- `swflux`: Net shortwave radiation (Wm^{-2}), positive out of the ocean. This is a component of the net heat flux, but needs to also be read separately as it penetrates below the ocean surface. Notably, this component predominantly varies on seasonal timescales and exhibits very little other variability.

Salinity:

- `sflux`: Net freshwater flux ($\mathrm{m}^3\mathrm{m}^{-}2\mathrm{s}^{-1}$, or just ms^{-1}). Positive out of the ocean.
- `saltFlux`: Rate of salt removal from the ocean by melting and formation of sea ice ($\mathrm{psukgm}^{-2}\mathrm{s}^{-1}$). Positive out of the ocean.
- `saltPlumeFlux`: Net flux of salt into the ocean by melting and formation of sea ice ($\mathrm{psukgm}^{-2}\mathrm{s}^{-1}$). Positive into the ocean. As with shortwave radiation, this extends below the surface and is kept separate.

Velocity:

- `ustress` x-oriented surface wind stress (Nm^{-2}). **Note** that, unlike standard ECCO, wind stress in flux-forced ECCO is read from the staggered grid, and is subject to the velocity-variable orientation issues described in 2.3.3.
- `vstress` y-oriented surface wind stress (Nm^{-2}).

Pressure:

- `apressure`: Atmospheric pressure (Nm^{-2}).

Installing flux-forced ECCOv4r4 When we installed the default ECCOv4r4 in Section 2.1, we also obtained the necessary source code and namelist files for flux-forced ECCO, which should be located in the directory `~/MITgcm/ECCOV4/release4/flux-forced`. We create a new directory in which to build this configuration, and copy this source code into that directory

```
mkdir /glade/u/home/YOUR_USERNAME/MITgcm/ECCOV4/release4_flux_forced;
cd /glade/u/home/YOUR_USERNAME/MITgcm/ECCOV4/release4_flux_forced;
cp -rp /glade/u/home/YOUR_USERNAME/MITgcm/ECCOV4/release4/flux-forced/* .;
mkdir build
```

We can then compile the model exactly as in Section 2.1.2. Note that the instructions here are for compiling to run the model on the default of 96 CPUs. The modifications necessary to run on 192 CPUs (as is preferable) are identical to those described in Section 2.2.2.

Although the initial control variables are the same as in ECCOv4r4, the forcing files are, of course, different, and need to be downloaded as in Section 2.1.3. In the same directory which holds the directories `input_init` and `input_forcing`, obtain the flux-forced files as follows:

```
wget -r --no-parent --user YOUR_PODAAC_USERNAME --ask-password \
https://ecco.jpl.nasa.gov/drive/files/Version4/Release4/other
mv ecco.jpl.nasa.gov/drive/files/Version4/Release4/other/ .
rm -r ecco.jpl.nasa.gov/
```

Note that these files require an additional 210GB of storage, and so it may be advantageous to delete the original `input_forcing` directory **if standard ECCOv4r4 is no longer required**.

Running flux-forced ECCOv4r4 Creating a run directory is largely the same as in Section 2.1.4, but we now get the namelist files from the directory where we installed the flux-forced configuration, and the forcing files from the location we just downloaded them to. Adapting the lines from Section 2.1.4:

```
mkdir /glade/scratch/YOUR_USERNAME/ECCOv4r4_reproduction_FF;
cd /glade/scratch/YOUR_USERNAME/ECCOv4r4_reproduction_FF;

# Directory where you saved flux-forced forcing files:
inputdir="/glade/work/YOUR_USERNAME/ECCOv4r4_input";

# Directory where you installed the flux-forced configuration:
srcdir_ff="/glade/u/home/YOUR_USERNAME/MITgcm/ECCOV4/release4_flux_forced";

ln -s ${inputdir}/input_init/xx_*.*.??ta . ;\
ln -s ${inputdir}/input_init/tile00?.mitgrid . ; \
ln -s ${inputdir}/input_init/error_weight/ctrl_weight/*.* . ; \
ln -s ${inputdir}/input_init/total_*.bin . ; \
ln -s ${inputdir}/input_init/smooth* . ; \
ln -s ${inputdir}/input_init/runoff-2d-Fekete-1deg-mon-V4-SMOOTH.bin . ; \
ln -s ${inputdir}/input_init/pickup*.0000000001.??ta . ; \
ln -s ${inputdir}/input_init/geothermalFlux.bin . ; \
ln -s ${inputdir}/input_init/fenty_biharmonic_visc_v11.bin . ; \
ln -s ${inputdir}/input_init/bathy_eccollc_90x50_min2pts.bin . ; \
cp -p ${inputdir}/input_init/tools/mkdir_subdir_diags.py . ; \

cp -p ${srcdir_ff}/namelist/data{,.autodiff,.cal,.cost,\
.exch2,.exf,.ggl90,.gmredi,.optim,.pkg,\
.salt_plume,.sbo,.seaice,.smooth} . ; \
cp -p ${srcdir_ff}/namelist/eedata . ; \
ln -s ${inputdir}/other/flux-forced/forcing/ . ; \
ln -s ${srcdir_ff}/build/mitgcmuv .
```

Note that we **did not copy** the `data.diagnostics`, `data.ctrl`, or `data.ctrl.restart` namelist files. The format of the flux-forced `data.diagnostics` is incompatible with the python script `mkdir_subdir_diags.py` and contains many output variables we might not be interested in, so we'll make a custom one as in Section 2.2.1. Any initial control adjustments specific to the flux-forced version are incorporated into the forcing files, and so, while the flux-forced version of `data.ctrl` differs from the original version, it is only in reference to files which we do not need for the reproduction run. We'll copy the old versions instead:

```
srcdir="/glade/u/home/YOUR_USERNAME/MITgcm/ECCOV4/release4"
cp -p ${srcdir}/namelist/data.{ctrl,ctrl.restart} .
```

We also run the following lines, adapted from (and described in) Section 2.1.4:

```
sed -i -e "s/#//" data.pkg; \
sed -i -e "s#'oce#'forcing/oce#g" data.exf; \
sed -i -e "s#'TFLUX#'forcing/TFLUX#g" data.exf; \
sed -i -e "s#'sIce#'forcing/sIce#g" data.exf; \
printf "&ECCO_COST_NML /\n&ECCO_GENCOST_NML /" > data.ecco ; \
python2 mkdir_subdir_diags.py
```

The only new line here is the first, which uncomments line 13 of `data.pkg`, which for some reason turned off the `diagnostics` package, preventing model output.

We also make a custom `data.diagnostics` file. The example below simply returns the monthly averages of the primitive ocean variables, but can be customised as desired by the user:

```
&diagnostics_list
#
    dumpatlast = .TRUE.,
#---
frequency(1) = 2635200.0,
fields(1,1) = 'THETA',
filename(1) = 'diags/THETA_mon_mean/THETA_mon_mean'
#---
frequency(2) = 2635200.0,
fields(1,2) = 'SALT',
filename(2) = 'diags/SALT_mon_mean/SALT_mon_mean'
#---
frequency(3) = 2635200.0,
fields(1,3) = 'UVEL'
filename(3) = 'diags/UVEL_mon_mean/UVEL_mon_mean'
#---
frequency(4) = 2635200.0,
fields(1,4) = 'VVEL'
filename(4) = 'diags/VVEL_mon_mean/VVEL_mon_mean'
#---
frequency(5) = 2635200.0,
fields(1,5) = 'WVEL_mon_mean'
filename(5) = 'diags/WVEL_mon_mean/WVEL_mon_mean'
#---
/
&DIAG_STATIS_PARMS
/
```

We can submit the job as before.

2.5.2 CORE Normal Year Forcing

As described above, in the context of using adjoint modelling to describe generalised ocean dynamics, the ECCO state estimate possesses the potential issue that its ocean state is necessarily changing with time, and thus the dynamics in any given year do not strictly represent those of the ocean "in general". The "Normal Year" forcing set (REF) was designed with this idea of "general" ocean dynamics in mind: instead of the surface forcing evolving to reflect historical atmospheric fluxes, it repeats annually, reflecting "typical" atmospheric forcing, including typical weather patterns and neutral indices of modes of major variability such as ENSO. This makes it particularly well-suited to long integrations and adjoint sensitivity studies. Here we will describe the installation of a modified ECCOv4 configuration which uses this forcing set, based on that of (REF).

Installing normal-year-forced ECCO We do not need to compile a wholly separate configuration of ECCO to run with CORE Normal Year forcing; it can be run using the standard (i.e. not flux-forced) executable which we compiled in Section 2.1.2, simply using a different combination of forcing, initialisation, and namelist files. However, to run the configuration for longer than 27 years, we need to make a very small adjustment and recompile. In the file

```
MITgcm/ECCOV4/release4/code/tamc.h
```

it is necessary to change the line

```
parameter( nchklev_1 = 5 )
```

to read

```
parameter( nchklev_1 = 10 )
```

which doubles the potential runtime to 54 years. This issue is due to the nature of the MITgcm checkpointing system (which we will also describe in Section 3.2.2) associated with adjoint model compilation. It has not been a problem thus far, however, as the standard ECCO state estimate is currently 26 years long.

Running normal-year-forced ECCO As in Section 2.1.3, We require initialisation and forcing files to run the ECCOv4r4 using this forcing set. Similarly to before, these are stored in `input_init` and `input_forcing` directories, and these can be downloaded from here [NOTE: This link will be updated to a permanent repository such as Zenodo at a later date].

As before, we make a directory in which to run the model and populate it with shortcuts to, or copies of, the necessary files:

```
inputdir=/path/to/ECCOv4r4_CORE_NYF_input

ln -s /path/to/MITgcm/ECCOV4/release4/build/mitgcmuv .

ln -s ${inputdir}/input_init/pickup* .
cp -rp ${inputdir}/input_init/NAMELIST/eedata .
cp -rp ${inputdir}/input_init/NAMELIST/data{,.autodiff,.ctrl,.cal,.cost,.diagnostics,.
    ecco,.exch2,.exf,.ggl90,.gmredi,.kpp,.layers,.optim,.pkg,.salt_plume,.sbo,.seaice,.
    smooth} .
ln -s ${inputdir}/input_init/*.bin .
ln -s ${inputdir}/input_init/tile* .
ln -s ${inputdir}/input_forcing/ .

sed -i -e "s#CORE2_#input_forcing/CORE2_#g" data.exf

ln -s ${inputdir}/input_init/smooth* .
ln -s ${inputdir}/input_init/xx_{kapgm,kapredi,diffkr}.*.??ta .
ln -s ${inputdir}/input_init/r2*.data .
```

As before (Section 2.1.4), we can now submit a run. The default run length (specified in `data`) is 25 years.

3 Differentiating the ECCO configuration and running its adjoint

As mentioned previously, the bespoke design of the ECCOv4r4 configuration for adjoint data assimilation means it has many features which are useful to adjoint sensitivity analysis. Much of its code was written to be *automatically differentiated* (AD) by computer software (as opposed to manually hand-coded ocean adjoint models such as NEMOTAM). Nevertheless, the use of the MITgcm adjoint for purposes other than data assimilation is still relatively undocumented. This section aims to assemble the available information on using the MITgcm for "generic" adjoint sensitivity analysis. Rather than using the adjoint to answer the question "how can one perturb the model in order to efficiently minimise the misfit to observed data?" as in assimilation, this approach uses the adjoint to answer the question "how can one perturb the model in order to maximise the impact on a Quantity of Interest (QoI), such as heat content, or volume transport?". This question is not typically asked with the ultimate goal of actually perturbing the system, but instead the goal of understanding the system's dynamics.

3.1 Differentiating, compiling, and running the adjoint of the model

3.1.1 Compiling the adjoint model (TAF)

At the release of ECCOv4r4, the preferred software for AD was Transformation of Algorithms in Fortran (TAF), proprietary code which requires a licence from FastOpt to run. Assuming this licence has been obtained, the procedure involves running a script which concatenates every source code file of the MITgcm configuration into one long file, transfers it to a remote FastOpt server that performs the differentiation and sends back differentiated code. This code can then be compiled locally, producing, in addition to the `mitgcmuv` executable, an `mitgcmuv_ad` executable for the model adjoint.

The compilation procedure is largely the same as in Sections 2.1.2 and 2.5.1. However, owing to an issue (as of July 2022), it is necessary to intervene after the `genmake2` step and manually edit the Makefile.

Step by step:

- As in Sections 2.1.2 and 2.5.1, install the chosen configuration up to the point of compilation.
- As before, load the necessary modules, create/enter the `build` directory, and run

  ```
  ../../../tools/genmake2 -mods=../code \
  -optfile=../../../tools/build_options/cheyenne -mpi
  ```

  swapping out `cheyenne` for your machine's build option file as necessary
- The above step will generate the `Makefile` which needs to be manually edited in a text editor such that the line `FFLAGS = ...` has `-mcmodel medium` apended to the end. It should look something like this

  ```
  FFLAGS = -W0 -WB -fPIC -convert big_endian -assume byterecl -align -march=corei7
      -axAVX -mcmodel medium
  ```
- We can skip the `make depend` step of before and run `make adall`. A prompt will ask for a passcode to confirm a licence for TAF is held: this might hang while the differentiation is taking place. The end result should be an executable `mitgcmuv_ad` in the `build` directory.

3.1.2 The ECCO `gencost` package and creating compatible ocean QoI files

In order to run the adjoint model, we require a cost function, or, more specifically to our case, an oceanic Quantity of Interest, and a way to supply this to the model. While in previous versions of MITgcm this

was handled by the `cost` package, requiring the user to recompile the model every time a new QoI was desired, the `ecco` package now supports "generic" cost functions not related to state estimation, which can be supplied at runtime, allowing the user to possess a single adjoint model executable for any number of QoIs.

In this section we will provide two examples of QoIs and how to generate (using python) files which provide these QoIs to the adjoint model. A more thorough list of possibilities is provided in the tables of parameters and options for `gencost` found in the MITgcm manual chapter on State Estimation Packages.

The use of `gencost` is controlled in two ways. Firstly, in the `data.ecco` namelist, general options are set, such as what type of QoI is being supplied to the model (the `gencost_barfile` option), such as the mean of some quantity over an area, or the volume transport through some region. The namelist also sets the averaging period of interest (the `gencost_avgperiod` option). Secondly, the namelist is told the location of a set of "mask" files, supplied by the user, which tell the model where and when in space and time to apply the above calculations. In our first example (Section 3.1.3) we will tell `data.ecco` that we are interested in the mean of temperature over an area, and supply "mask" files which are 0 anywhere in space and time that we do not want to calculate this mean temperature, specifically, outside of the upper North Atlantic on time scales longer than one year.

3.1.3 Example 1: Annual-mean North Atlantic Upper Ocean Heat Content

In this example, we will run the ECCOv4r4 configuration with CORE Normal Year Forcing (as in Section 2.5.2), along with its adjoint, to determine the sensitivity of one-year-averaged Upper Ocean Heat Content (UOHC; shallower than 700 m) of the North Atlantic to changes one year ahead. In a forward-model sense, we are asking what the effect of a perturbation is if we apply the perturbation, wait a year, and then measure the UOHC over the next year.

To begin, make a run directory as before. However, as we now wish to run the adjoint model instead of the regular forward model, we should link to the adjoint model executable `mitgcmuv_ad` instead of the forward model executable `mitgcmuv`. They are both found in the `MITgcm/ECCOv4/release4/build` directory.

i. Creating a spatial mask While `gencost` only requires the existence of a spatial mask to run (exiting with an error if it does not exist), it searches for one-dimensonal depth and time masks as well, treating them as `1` everywhere if they are not found. The MITgcm manual suggests that it is necessary to supply a two-dimensional (x,y) spatial mask and a one-dimensional (z) depth mask, but it is possible to combine both into a single three-dimensional mask, which we will do instead for convenience. The time mask will be handled separately, in **ii.**

We will use many of the python-based approaches we have before:

```
import numpy as np
import xarray as xr
import xmitgcm as xm
import ecco_v4_py as ecco

GDS=xr.open_dataset('~/ECCO-GRID.nc')
atlmskC=ecco.get_basin_mask(basin_name='atl',mask=GDS.hFacC,less_output=True).isel(k=0)
atlmskC=atlmskC.where(GDS.YC>0,0) #Only North Atlantic, i.e. latitudes>0
atlmskC=atlmskC.where(GDS.Z >-700 ,0) #Only upper 700m
```

In this first part we simply made an array (`atlmskC`) with the shape of the LLC90 C-point grid ($50 \times 13 \times 90 \times 90$) which has value `0` everywhere outside of the Atlantic, south of the equator, or deeper than 700 m, and `1` in the upper North Atlantic.

Before we save this mask, we have another step. As the `ecco.gencost` feature is limited to calculating

averages of variables over regions, if we simply supply this mask and ask for the average of temperature, we will get back, as expected, the average temperature over the region, rather than the heat content. To get from average temperature to heat content, we thus need to multiply by the volume over which the average is taken (as well as some constants). To calculate this volume, we multiply the two-dimensional area variable `rA` in the `ECCO-GRID.nc` file by the one-dimensional level height variable `drF`, giving a three-dimensional array of the volume of every grid cell. We apply our mask to this volume and sum. We then multiply by the specific heat capacity (3850 $\mathrm{JK^{-}1kg^{-1}}$) and a reference density (1025 $\mathrm{kgm^{-3}}$) to get an approximate value by which to rescale average temperature to heat content.

```
dVC=GDS.rA * GDS.drF # Array of model grid volumes
cp,rho0=3850,1025 # Unit-conversion constants
voltot=dVC*atlmskC # Total volume of masked area
Tbar_to_HC=voltot*cp*rho0 # Avg. temp. to heat content multiplier for region
print('Rescaling factor: '+str('%f' % Tbar_to_HC))
```

The above will print a multiplication factor that we will give to the model to apply for us in **iii**.

Lastly, we write our mask to a binary file, as in Sections 2.4.1 and 2.4.2:

```
atlmskC=atlmskC.rename({'tile':'face'}) # Rename dimension for xmitgcm compatibility
extra_metadata=xm.utils.get_extra_metadata(domain='llc',nx=90)
facets=xm.utils.rebuild_llc_facets(atlmskC,extra_metadata)
compact=xm.utils.llc_facets_3d_spatial_to_compact(\
          facets,'k',extra_metadata=extra_metadata)
xm.utils.write_to_binary(compact,'UOHC_NATL_MASKC')
```

As before, we renamed the `tile` dimension to the less accurate "face" as we require the `xmitgcm` module to write this to a binary file that can be read by the model, and this module does not recognise the name "tile".

ii. Creating a time mask This is quite a lot simpler, but still has some quirks. The `ecco.gencost` approach allows the user to choose a time period to average the QoI over: 'step' (a model timestep, or one hour in the ECCOv4r4 default setup), 'day', 'month', or 'const' (for fields that are constant in time, such as diffusion parameters). Note that while 'year' is present as an option in the MITgcm source code, it has not been fully implemented. Instead, to calculate the year-average of our UOHC QoI, we use the time mask to ask the model to apply the month-average QoI for 12 consecutive months. We can then divide the adjoint output by 12, which is equivalent to the annual average.

To create a time mask, we need to decide an averaging period (in this case, months), then determine how many such periods will span our model run. As we are interested in sensitivity to annual-average heat content one year in advance, we will be running the model for a total of two years, or 24 months. This consists of 12 months of lead time (the one year in advance) followed by 12 months of "measurement time", during which the average of the QoI is calculated. The corresponding time mask is just twelve 0s followed by twelve 1s:

```
time_mask=np.zeros(24)
time_mask[-12:]=1
xm.utils.write_to_binary(time_mask,'UOHC_NATL_MASKT')
```

To check our reasoning is in agreement with what the model sees, we can view any of the `STDOUT` files while the model is running. The line

```
ECCO configuration >>> START <<<
```

is followed by lines showing the dates between which the QoI is applied, as well as the number of periods (`number of records`).

In the run directory, there will now be two mask files: `UOHC_NATL_MASKC` (spatial), and `UOHC_NATL_MASKT` (temporal). Lastly, we configure the namelists before running the adjoint.

iii. Setting the namelist files Edit the `data.ecco` file to read:

```
# ******************
# ECCO cost functions
# ******************
 &ECCO_COST_NML
 &
# ***************************
# ECCO generic cost functions
# ***************************
 gencost_name(1) = 'UOHC_NATL',
 gencost_barfile(1) = 'm_boxmean_theta',
 gencost_avgperiod(1) = 'month',
 gencost_itracer(1) = 1,
 gencost_msk_is3d(1)=.TRUE.,
 gencost_outputlevel(1) = 1,
 gencost_mask(1) = 'UOHC_NATL_MASK'
 mult_gencost(1) = 8187847248019018743808,
#
 &
#
```

The important points here are

- `gencost_barfile` controls what type of QoI this is, in this case average temperature over a "box" or region.
- `gencost_avgperiod` is the temporal average applied in conjunction with the time mask file created above
- `getcost_mask` is the prefix to the mask files created above. Note that the suffixes `T` and `C` are missing: these will be applied by the model when looking for the mask files
- `mult_gencost` This is an optional rescaling factor. The value here is the number we calculated when creating the spatial mask, to convert the volume-mean temperature into the volume-integrated heat content, divided by 12. This division is to convert our 12 consecutive (and cumulative) monthly means into a single annual average.

Additionally, `gencost_name` defines the name of our cost function, `gencost_itracer` is the index of the tracer in use (1 for temperature), `gencost_msk_is3d` specifies that our cost function is 3d, and `gencost_outputlevel` asks for more or less output (typically in the context of data assimilation).

We should also edit the general namelist `data`, which controls the frequency of adjoint output via the `adjDumpFreq` parameter (in seconds). By default, the model outputs a snapshot of adjoint sensitivities once per year, which is not particularly useful for sensitivity analysis. Setting `adjDumpFreq = 432000.0,` returns outputs every five model days. As we also wish to run the model for two years, we should also change `nTimeSteps` to `17520`.

We can submit our adjoint run just as any of our previous model runs, but note that the executable being called should be changed from `mitgcmuv` to `mitgcmuv_ad` in our submission script.

3.1.4 Example 2: Decadal-mean Atlantic meridional volume transport at 26.5°N

It is assumed that Section 3.1.3 has been followed, and so this will be a less detailled description, predominantly focusing on the differences between applying a volume transport QoI and a spatial-mean-type QoI. We will consider ten-year-average volume transport over the upper 1000 m across a constant latitude of 26.5° N in the North Atlantic, as part of a fifty year adjoint run.

The primary difference is that, as our key variable is now velocity (rather than temperature) we are no longer making a mask for the "C"-point grid. We instead need to make two spatial masks, one for y-oriented velocity on the "S" points at the south of the grid cells, and one for x-oriented velocity on the "W" points at the west of the grid cells.

i. Creating spatial masks Our approach is to find the tiles bisected by our latitude choice. We can see by inspection (Figre 1) that these are tiles 2,5,7, and 10, but, as we are only interested in the Atlantic, only 2 and 10. While this is clear to us visually, the particular "j" index of tile 2 and "i" index of tile 10 which are closest to 26.5° are less obvious. The code below determines both the tile and this index for any given latitude.

For the "S" point tiles (0,1,2,3,4,5): Begin by loading the latitude values at the S points for tiles 0-5, and reshaping them into one large 2D array (with shape 180×270)

```
import numpy as np
import xarray as xr
import xmitgcm as xm
import ecco_v4_py as ecco
GDS=xr.open_dataset('~/ECCO_GRID.nc')

QOI_LAT=26.5
latS_reshape=np.hstack([np.vstack(GDS.YG[0:3,:]),np.vstack(GDS.YG[3:6])])
```

Next, find which of the 270 rows of this array is closest to 26.5:

```
latidxS=np.argmin(np.abs(latS_reshape-QOI_LAT),axis=0)
latidxS=int(np.unique(latidxS))
```

The first line returns a 180-length array where every entry is 197. The second line reduces this to a single-entry array, then converts this to an integer (also 197). This is the index on our giant latitude array, but we need to convert this back to a pair of tile indices and a row index on those tiles. As we concatenated three 90-row tiles to make a 270-row array, we can convert the row index on this array back to a tile index using floor division by 90. 197//90=2, so our tile index is 2. In every case, the latitude line will also bisect the tile index plus 3 (in this case tile 5):

```
tileidsS=[latidxS//90,latidxS//90 + 3]
```

To get the row index on these tiles, we simply need the remainder when 197 is divided by 90:

```
jidxS=latidxS % 90
```

So we obtain that, on the "S" grid, the closest indices to 26.5 are $(2, 17, :)$ and $(5, 17, :)$. Repeating this for the "W" grid:

```
latW_reshape=np.hstack([np.vstack(GDS.YG[ 7:10,:]),np.vstack(GDS.YG[10:13])])
latidxW=np.argmin(np.abs(latW_reshape-QOI_LAT),axis=1)
latidxW=int(np.unique(latidxW))
tileidsW=[latidxW//90+7,latidxW//90 + 10]
iidxW=latidxW % 90
```

Which returns the outcome that, on the "W" grid, the closest indices are $(7, :, 73)$ and $(10, :, 73)$ We can now make a mask which is `1` at these locations and `0` elsewhere.

```
atlmskS=ecco.get_basin_mask(basin_name='atl',mask=GDS['hFacS'],less_output=True)
MASKS=atlmskS.where( ((atlmskS.tile==tileidsS[0]) | (atlmskS.tile==tileidsS[1])) \
                    & (atlmskS.j_g==jidxS),0)
atlmskW=ecco.get_basin_mask(basin_name='atl',mask=GDS['hFacW'],less_output=True)
MASKW=atlmskW.where( ((atlmskW.tile==tileidsW[0]) | (atlmskW.tile==tileidsW[1])) \
                    & (atlmskW.i_g==iidxW),0)*-1
```

Note that the "W" mask is multiplied by `-1`. This is because the flow in the positive x-direction on tiles 7-10 is North to South, the opposite of the positive y-direction on tiles 0-5. As we are interested in meridional volume transport, we apply this correction so that northward flow is positive in both cases.

This procedure is admittedly quite involved, but allows us to swap out any QoI latitude (or ocean basin).

Lastly, we mask the region below our maximum depth of 1000m:

```
depmax=1000
MASKS=MASKS.where(GDS.Z>-depmax,0)
MASKW=MASKW.where(GDS.Z>-depmax,0)
```

We can now write these to binary files, as before:

```
extra_metadata=xm.utils.get_extra_metadata(domain='llc',nx=90)

facetsS=xm.utils.rebuild_llc_facets(MASKS.rename({'tile':'face'}),extra_metadata)
facetsW=xm.utils.rebuild_llc_facets(MASKW.rename({'tile':'face'}),extra_metadata)

compactS=xm.utils.llc_facets_3d_spatial_to_compact(\
              facetsS,'k',extra_metadata=extra_metadata)
compactW=xm.utils.llc_facets_3d_spatial_to_compact(\
              facetsW,'k',extra_metadata=extra_metadata)

xm.utils.write_to_binary(compactS,'MVT26_MASKS')
xm.utils.write_to_binary(compactW,'MVT26_MASKW')
```

ii. Other masks and namelists As before, we require a time mask. As we are now interested in a decade average, and wish to run the model for 50 years, this should be 480 zeros followed by 120 ones.

The format for the `data.ecco` namelist is:

```
# ******************
# ECCO cost functions
# ******************
 &ECCO_COST_NML
 &
# ***************************
# ECCO generic cost functions
# ***************************
# Volume transport:
 &ECCO_GENCOST_NML
 gencost_name(1) = 'MVT26',
 gencost_barfile(1) = 'm_horflux_vol',
 gencost_avgperiod(1) = 'month',
 gencost_msk_is3d(1)=.TRUE.,
 gencost_outputlevel(1) = 1,
 gencost_mask(1) = 'MVT26_MASK'
 mult_gencost(1) = 0.000000008333333,
#
```

```
 &
#
```

This is largely the same as before, with the primary difference being the absence of the `gencost_itracer` parameter and the change of `gencost_barfile` to `m_horflux_vol`, which tells the model to look for mask files with the suffixes "W" and "S" (as we created), and to integrate the total transport passing through the non-zero regions of those masks. `mult_gencost` in this example is set to $1 \times 10^{-6} \div 120$, where 1×10^{-6} converts the units from $m^3 s^{-1}$ to Sv, and 120 is the number of cumulative monthly averages we are applying, hence the number we need to divide by to convert this to a decade-averaged quantity.

As before, we edit `data` to ensure that adjoint output (`adjDumpFreq`) is sufficiently frequent, and that the model will run for fifty years (`nTimeSteps=438000`). We can then submit the run.

3.1.5 Increased efficiency on Pleiades

If running on NASA's Pleiades HPC, there are some quirks of the system that result in reduced adjoint model performance. Adjoint models have a very high throughput as they require output from the forward model to be saved with high frequency in order to run through that output backwards. For MITgcm, which uses a checkpointing system (Section 3.2), this output is not required after the model is finished running, and so it can be held in temporary, memory-based storage which is much faster for compute nodes to access. The `/tmp` filesystem on Pleiades is ideal for this purpose.

The location where forward model output is temporarily stored is specified in the main namelist file `data`. In particular, the line

```
adTapeDir='tapes',
```

should be changed to something like

```
 adTapeDir='/tmp/YOUR_USERNAME_tapes',
```

where the inclusion of your username may serve to stop a conflict with other ongoing MITgcm runs. This directory will not exist when the run begins, and so the line

```
 mkdir /tmp/YOUR_USERNAME_tapes
```

should be added to the submission script.

Similarly, the forcing files, which are subject to high-frequency reading, can be copied to `/tmp/` at runtime, by adding the line

```
 cp -rp input_forcing/ /tmp/
```

to the job submission script and prepending `/tmp/` to the forcing file locations specified in `data.exf`.

These changes result in significantly increased performance. Running on 192 CPUs using the "long" queue (where jobs can continue for 120 hours) allows the adjoint to run for around fifty years.

3.1.6 Analysing the adjoint model output

Up to this point the model output has been handled by the `diagnostics` package, which allows a large amount of runtime customisation regarding storage location, filenames, frequency, etc. It is possible to obtain adjoint model output using the diagnostics package, but some modifications have to be made at compile time and compatibility with the DIVA functionality (Section 3.2) is lost.

The `diagnostics` package is not *required* for model output to be produced, but gives the user much more control. The more basic output routines are controlled in the `data` namelist, such as the frequency of adjoint output `adjDumpFreq`. *All* standard adjoint variables (listed below) are then returned at this frequency and simply dumped in the run directory: the user does not control the output location or which variables are returned.

The variables are

```
ADJaqh,ADJclimsst,ADJggl90tke,ADJkapredi,ADJqnet,ADJsalt,ADJtaux,ADJustress,ADJvvel,
ADJatemp,ADJempr,ADJhflux,ADJlwdown,ADJqsw,ADJsflux,ADJtauy,ADJuvel,ADJwvel,ADJclimsss,
ADJetan,ADJkapgm,ADJprecip,ADJrunoff,ADJswdown,ADJtheta,ADJvstress,ADJuwind,ADJvwind
```

These variables are described in detail in the MITgcm manual, but are the adjoint equivalents of their forward model counterparts without the `ADJ` prefix. For instance `ADJtheta` is the adjoint sensitivity to `theta`, or potential temperature, having units equal to the unit of the cost function per unit of temperature, e.g. JK^{-1} in Example 3.1.3.

As we are not using the `diagnostics` package, our python script `mds_to_netcdf.py` will not work. We can write a very similar python script `ADJ_to_netcdf.py` (Appendix A.2) to convert the adjoint output. This script either accepts a list of variables to convert to netCDF, or converts all output from the above list. Additionally, within the script there is an option `surface_only`, which returns only time-varying 2D output at the surface. This takes up one fiftieth the storage space of including all depth levels. As many adjoint sensitivity studies are only concerned with surface perturbations, this may be of utility.

3.2 Stopping and restarting the adjoint model (DIVA)

DIVided Adjoint (DIVA), much like the forward model "pickup" system (Section 2.2.3) , allows a single run of the adjoint model to be spread over multiple jobs: before one job ends, all the necessary information to continue running the model is saved. This is very useful on HPC systems which have short time limits on jobs (for instance Cheyenne, which limits jobs to twelve hours).

3.2.1 Overview of the procedure

DIVA capitalises on the checkpointing procedure of MITgcm. Briefly, the reverse-time nature of the adjoint means information from earlier in time is always needed. The model thus has to first be run forward for the entire duration of the run to store provide this information. In MITgcm, to avoid the storage overhead of storing the model state at every timestep for the entire run, the forward model is instead run in smaller segments, which are stored at lower cost, but at the expense of effectively running the entire simulation multiple times over. The full procedure is described here (MITgcm manual, Automatic Differentiation chapter, "storage vs. recomputation in reverse mode" section). In short:

3. the forward model runs through once, creating a number of checkpoints (effectively model restarts) along the way (tall black lines in the top row of Figure 5).

2. The forward model runs again, restarting from the final checkpoint and creating a series of more frequent checkpoints between it and the run end (medium black lines in the second row of Figure 5)

1. The forward model runs again, restarting from the "new" final checkpoint (very close to the end) but storing the model state along the way. The adjoint model then runs backward using this stored information until it gets back to the checkpoint (and runs out of stored information). At this point this process is repeated, running forward from the penultimate checkpoint, storing, running backwards, etc., until it has run out of the more frequent checkpoints created by (2). (2) is then repeated, creating a new series of more frequent checkpoints between the penultimate and final checkpoints created by (3), so that (1) can loop back over these.

Note the reverse order of the above list corresponding to the "levels" of checkpointing (3, then 2, then 1) referred to in the manual and code.

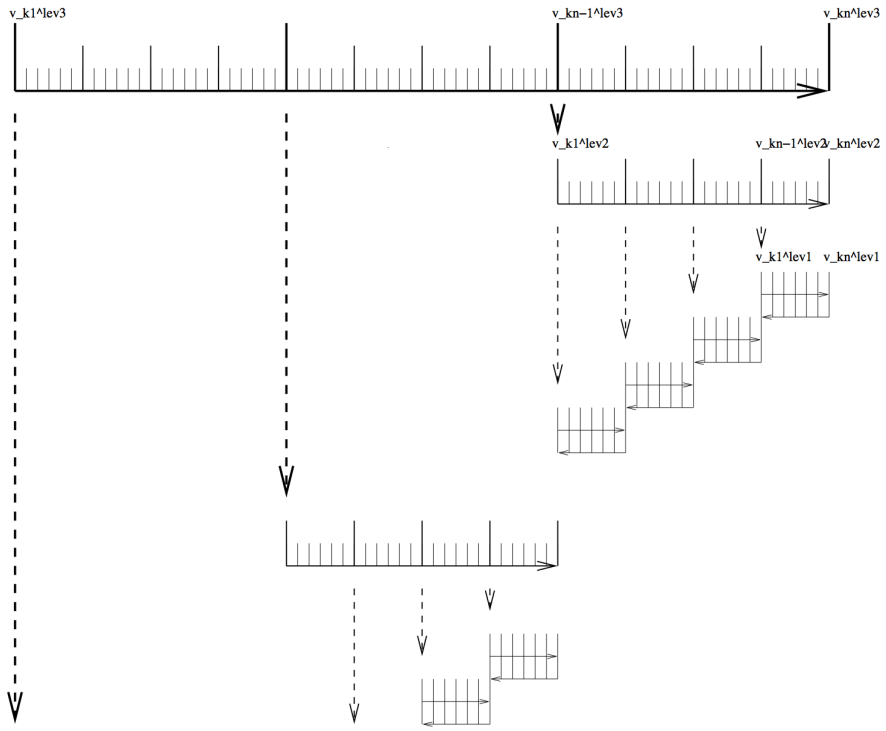

Figure 5: Schematic of the checkpointing procedure used by the adjoint model

3.2.2 Compiling using DIVA

DIVA is a fairly niche feature of MITgcm which is not thoroughly supported or documented. A few patches have to be applied while compiling to make it work. It is recommended to make a new configuration, as before, in which we will enable DIVA, e.g. `MITgcm/ECCOv4/release4_DIVA`. The source code modifications will then take place in the `code` subdirectory.

Firstly, the DIVA package has to be enabled before compilation. Up to this point, our model configurations have been compiled with all the packages we've needed (and some we haven't) and we've decided which packages to enable at runtime by modifying the namelist `data.pkg`. This time we need to incorporate a package which is not enabled by default in any of the configurations we have considered so far. Compile-time package settings are generally controlled by the file `code/PACKAGES_CONFIG.h`, but, as DIVA is more a feature of the `autodiff` package than a wholly separate package, it is enabled in the file `code/AUTODIFF_OPTIONS.h`: Change the following lines

```
 C o use divided adjoint to split adjoint computations
#undef ALLOW_DIVIDED_ADJOINT
#undef ALLOW_DIVIDED_ADJOINT_MPI
```

to

```
 C o use divided adjoint to split adjoint computations
#define ALLOW_DIVIDED_ADJOINT
#define ALLOW_DIVIDED_ADJOINT_MPI
```

As in Section 3.1.1, we need to edit the `Makefile` after the `genmake2` step. As in that section, we add `-mcmodel medium` to the end of the line `FFLAGS = ...`. Additionally, the line beginning `AD_TAF_FLAGS` should be edited to include the flags `-pure` and `-mpi`, for example:

```
AD_TAF_FLAGS = -server fastopt.net -f77 -reverse -mpi -pure -i4 -r4 ...
```

Recall from Section 3.1.1 that the adjoint compilation takes place in two stages. First, the model source code is concatenated into a single file and sent to a remote server to be differentiated and returned. Next, the file that is returned (`ad_taf_output.f`) is compiled. Previously this was all handled in a single command, `make adall` . However, we now need to edit the `ad_taf_output.f` file before it is compiled, and so each stage of the compilation should be handled separately. Begin by running `make depend` , then the first stage can be triggered with the command `make adtaf` .

This returns the `ad_taf_output.f` file. Edit this file such that, right before the only occurrence of

```
 C--------------------------------------------
C read snapshot
C--------------------------------------------
```

the following `if` statement is added:

```
      call cost_averagesfields( endtime,mythid )
     call barrier( mythid )
C-----ADD THE FOLLOWING LINE-------------------
     if (idivbeg .ge. nchklev_3) then
C--------------------------------------------
        call cost_final_ad( mythid )
        if (useecco) then
           call ecco_cost_driver_ad( mythid )
        endif
        if (useprofiles) then
           call profiles_inloop( endtime,mythid )
           call cost_profiles_ad( mythid )
           call profiles_inloop_ad( endtime,mythid )
        endif
        call cost_averagesfields_ad( endtime,mythid )
C----ADD THE FOLLOWING LINE--------------------
     endif
C--------------------------------------------

C--------------------------------------------
C read snapshot
C--------------------------------------------
```

The configuration can then be compiled using `make adall` .

NOTE: The above has been tested and is working. Nevertheless, a further issue has been reported. If, when running the compiled executable `mitgcmuv_ad` the model appears to unexpectedly terminate, it might be worth deactivating the following packages in `AUTODIFF_OPTIONS.h` and compiling as above again:

```
C o tape settings
#undef ALLOW_AUTODIFF_WHTAPEIO
#undef AUTODIFF_USE_OLDSTORE_2D
#undef AUTODIFF_USE_OLDSTORE_3D
#undef EXCLUDE_WHIO_GLOBUFF_2D
#undef ALLOW_INIT_WHTAPEIO
```

3.2.3 Running using DIVA

DIVA splits the procedure of 3.2.1 up into separate executions of the adjoint model. The number of required executions depends on the number of checkpoints created by the first run (e.g. 3 in the above image).

The default number of top-level checkpoints in the ECCOv4r4 configuration is 220, which is set in the file `MITgcm/ECCOV4/release4/build/tamc.h` using the parameter `nchklev_3`. The first run creates the 220 checkpoints, and each subsequent run begins the above procedure from one of them, moving backwards from the end. This means the model executable is run 221 times overall. This results in a substantial overhead as the model is cold-started hundreds of times and has to do a lot of initialisation work each time. This is on top of the fact that the three-level checkpointing system means that the forward model has to be run from start to finish three times (rather than just one) just to generate the checkpoints.

In order to determine where it is in the checkpointing procedure when cold-started, the model first looks for a file `costfinal` created by the first run, which tells it that the first run is complete. If the file exists, subsequent runs create files `divided.ctrl` (which tells it which of the 220 checkpoints it is starting from) and `snapshot?????` (a different file for each processor) which tells it the state of the adjoint where it left off.

In order to run the model using DIVA, `mitgcmuv_ad` has to be executed hundreds of times. There are multiple ways to handle this. For instance, the user can submit a new job for each execution, either by asking each job to submit the next job after the model execution line or by using job dependencies. The disadvantage with this approach is that each subsequent job will have to wait in the queue. It is possible that multiple executions can be fit into the time limit of a single job, so an alternative approach is to run the model as many times as possible within the time limit, then submit another job to pick up near wherever the last execution got cut off.

This code block reads `divided.ctrl` to determine how many executions are left, and if it is greater than 0, runs the model executable. This is repeated in a `while` loop until the job runs out of time, at which point a new job should be submitted:

```
while :
do
   if [ $(cat divided.ctrl | xargs | cut -f 2 -d ' ') -gt 0 ];then
      mpiexec_mpt dplace -s 1 ./mitgcmuv_ad
   fi
done
```

A Python scripts

A.1 Convert standard model MDS output to netCDF

(Copy it from `mds_to_netcdf.py` in the file list)

A.2 Convert adjoint output to netCDF