ECCO Version 4 Python Tutorial Documentation

Release 4.3-20191128

Ian Fenty

May 09, 2023

GETTING STARTED

1 Additional Resources 3

2 Indices and tables 377

This website contains a set of tutorials about how to use the ECCO Central Production Version 4 (ECCO v4) global ocean and sea-ice state estimate. The tutorials were written in Python and make use of the *ecco_v4_py* Python library, a library written specifically for loading, plotting, and analyzing ECCO v4 state estimate fields.

CHAPTER

ONE

ADDITIONAL RESOURCES

The ECCO v4 state estimate is the output of a free-running simulation of a global ca. 1-degree configuration of the MITgcm. Prior to public release, the model output files model are assembled into NetCDF files. If you would like to work directly with the flat binary "MDS" files provided by the model then take a look at the xmitgcm Python package. The xgcm Python package provides tools for operating on model output fields loaded with xmitgcm. If you wish to analyze the MITgcm model output using Matlab then we recommend the gcmfaces toolbox.

The ecco_v4_py package used in this tutorial was inspired by the xmitgcm package and gcmfaces toolboxes.

1.1 The ECCO Ocean and Sea-Ice State Estimate

1.1.1 What is the ECCO Central Production State Estimate?

The Estimating the Circulation and Climate of the Ocean (ECCO) Central Production state estimate is a reconstruction of the three-dimensional time-varying ocean and sea-ice state. Currently in Version 4 Release 3, the state estimate covers the period Jan 1, 1992 to Dec 31, 2015. The state estimate is provided on an approximately 1-degree horizontal grid (cells range from ca. 20 to 110 km in length) and 50 vertical levels of varying thickness.

The ECCO CP state estimate has two defining features: (1) it reproduces a large set of remote sensing and in-situ observational data within their prior quantified uncertainties and (2) the dynamical evolution of its ocean circulation, hydrography, and sea-ice through time perfectly satisfies the laws of physics and thermodynamics. The state estimate is the solution of a free-running ocean and sea-ice general circulation model and consequently can be used to assess budgets of quantities like heat, salt and vorticity.

ECCO Version 4 Release 3 is the most recent edition of the global ocean state estimate and estimation system described by Forget et al. (2015b, 2016).

A brief synopsis describing Release 3 can be found here:

https://ecco.jpl.nasa.gov/drive/files/Version4/Release3/doc/v4r3_estimation_synopsis.pdf

A high-level analysis of the state estimate can be found here: https://ecco.jpl.nasa.gov/drive/files/Version4/Release3/doc/v4r3_overview_plots.pdf

Relation to other ocean reanalyses

ECCO state estimates share many similarities with conventional ocean reanalyses but differ in several key respects. Both are ocean reconstructions that use observational data to fit an ocean model to the data so that the model agrees with the data in a statistical sense. Ocean reanalyses are constructed by directly adjusting the ocean model's state to reduce its misfit to the data. Furthermore, information contained in the data is only explored forward in time. In contrast, ECCO state estimates are constructed by identifying a set of ocean model initial conditions, parameters, and atmospheric boundary conditions such that a free-running simulation of the ocean model reproduces the observations as a result of the governing equations of motion. These equations also provide a means for propagating information contained in the data back in time ("upstream" of when/where observations have been made). Therefore, while both ocean reanalyses and ECCO state estimates reproduce observations of ocean variability, only ECCO state estimates provide an explanation for the underlying physical causes and mechanisms responsible for bringing them into existence (e.g., Stammer et al. 2017).

Conservation properties of ECCO state estimates

By design, ECCO state estimates perfectly satisfy the laws of physics and thermodynamics and therefore conserve heat, salt, volume, and momentum (Wunsch and Heimbach, 2007, 2013). Indeed, it is because of these conservation properties that ECCO state estimates are used to investigate the origins of ocean heat, salt, mass, sea-ice, and regional sea level variability (e.g., Wunsch et al., 2007; Köhl and Stammer, 2008; Piecuch and Ponte 2011; Fenty and Heimbach, 2013b; Wunsch and Heimbach 2013, 2014; Fukumori and Wang 2013; Buckley et al. 2014, 2015; Forget and Ponte, 2015, Fenty et al., 2017; Piecuch et al. 2017).

1.1.2 How is the ECCO Central Production State Estimate Made?

The ECCO ocean reanalysis system is a mature, state-of-the-art data tool for synthesizing a diverse Earth System observations, including satellite and in-situ data, into a complete and physically-consistent description of Earth's global ocean and sea-ice state. Ocean observations, even those from satellites with global coverage, are still sparse in both space and time, relative to the inherent scales of ocean variability. The ECCO reanalysis system is able to reconstruct the global ocean and sea-ice state by synthesizing hundreds of millions of sparse and heterogeneous ocean and sea-ice satellite and in-situ data with an ocean and sea-ice general circulation model. Through iterative high-dimension non-linear optimization using the adjoint of the ocean and sea-ice model the ECCO reanalysis system identifies a particular solution to the system of equations describing ocean and sea-ice dynamics and thermodynamics that reproduces a set of constraining observations in a least-squares sense.

By simultaneously integrating numerous diverse and heterogeneous data streams into the dynamically-consistent framework of the physical model we make optimal use of the data. Users of the ECCO reanalysis are not only provided a comprehensive description of the Earth's changing ocean and sea-ice states but also information about the underlying physical processes responsible for driving those changes.

For a list of the input data used to contrain the ECCO Version 4 Release 3 state estimate see:

https://ecco.jpl.nasa.gov/drive/files/Version4/Release3/doc/v4r3_input_data.pdf

1.2 ECCO v4 state estimate ocean, sea-ice, and atmosphere fields

The complete state estimate consists of a set of ocean, sea-ice, air-sea flux, and atmosphere state variables that are the output from a free-running ocean and sea-ice general circulation model.

1.2.1 Geographical layout

Ocean, sea-ice, air-sea flux, and atmosphere fields are provided in two spatial layouts:

- 13-tile *native* lat-lon-cap 90 (llc90) grid
- 0.5° x 0.5° latitude and longitude grid

13-tile *native* lat-lon-cap 90 grid

The lat-lon-cap (llc) is the decomposition of the spherical Earth into a Cartesian curvilinear coordinate system . It is a topologically non-trivial cubed-sphere rendering in the northern hemisphere and a dipolar grid in the southern hemisphere. Between 70°S and ~57°N, model grid cells are approximately oriented to lines of latitude and longitude. A special Arctic "cap" is situated north of ~57°N.

The Cartesian curvilinear coordinate system is divided into 13 tiles, each consisting of 90x90 grid cells in the horizontal and 50 vertical levels. Horizontal model grid resolution varies spatially from 22km to 110km, with the highest resolutions at high latitudes and lowest resolution in mid latitudes. Vertical grid spacing increases with depth from 10m to 456.5m. The bottom of the deepest model grid cell is 6145m below the surface.

The Cartesian (x,y) coordinates of llc tiles do not coorespond to longitude and latitude. Horizontal velocities are defined relative to the **local orientation** of x and y in the tile. Velocities in the positive x direction are defined as positive u. Velocities in the positive y direction are defined as positive v.

Available fields on the llc90 grid

monthly-averaged ocean and sea-ice fields: https://ecco.jpl.nasa.gov/drive/files/Version4/Release3/nctiles_monthly/README

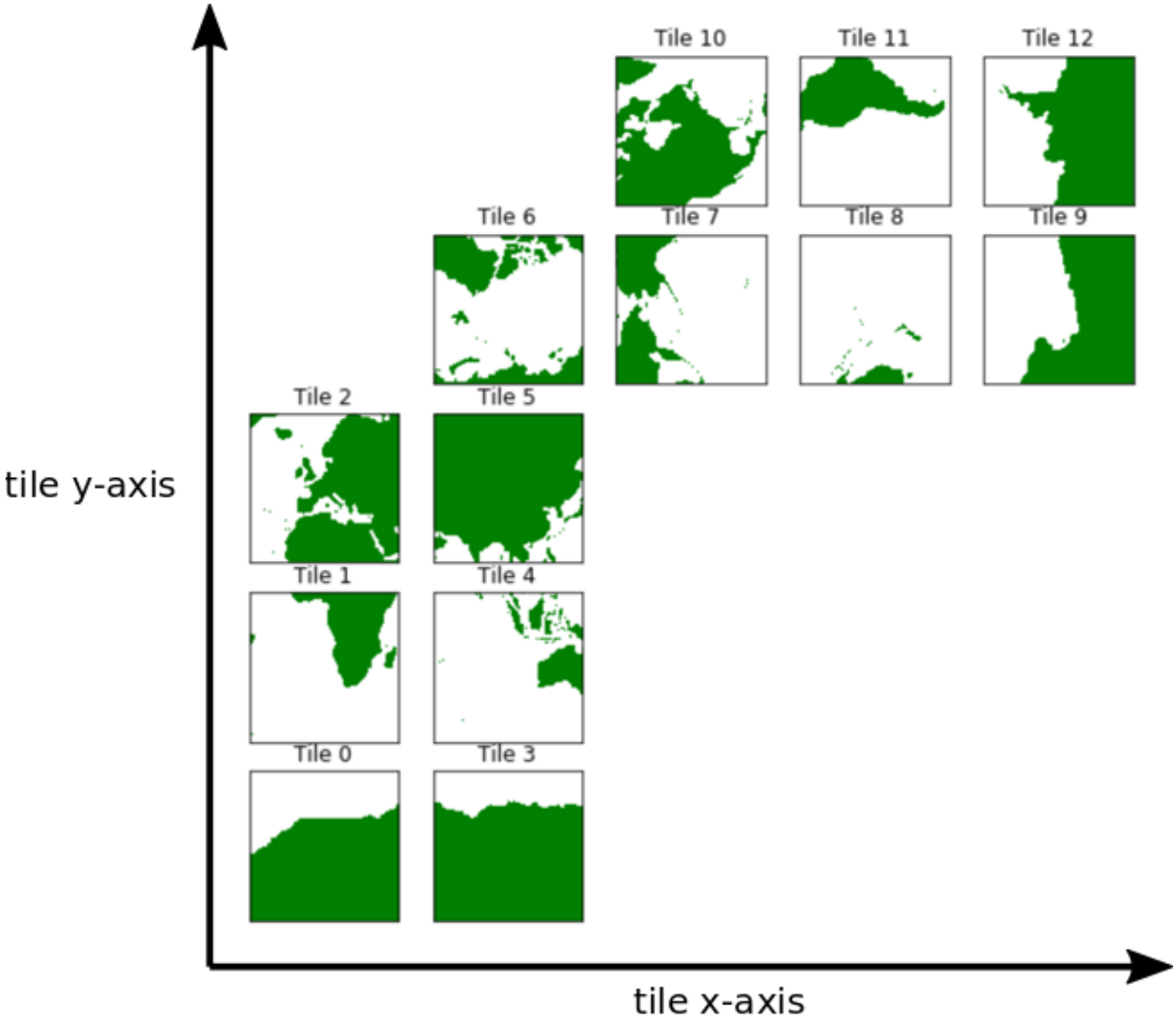
monthly-snapshot ocean and sea-ice fields: https://ecco.jpl.nasa.gov/drive/files/Version4/Release3/nctiles_monthly_snapshots/README

daily-averaged ocean and sea-ice fields: https://ecco.jpl.nasa.gov/drive/files/Version4/Release3/nctiles_daily/README

6-hourly atmosphere fields: https://ecco.jpl.nasa.gov/drive/files/Version4/Release3/input_forcing/README

interpolated 0.5° x 0.5° latitude-longitude grid

Select monthly-average fields from the *native* lat-lon-cap model output have been interpolated to a more user-friendly 0.5° latitude-longitude grid.


Tile 10
Tile 11
Tile 12
Tile 6
Tile 7
Tile 8
Tile 9
Tile 2
Tile 5
tile y-axis
Tile 1
Tile 4
Tile 0
Tile 3
tile x-axis

Available fields on the 0.5° x 0.5° latitude-longitude grid

0.5° x 0.5° monthly-averaged ocean, sea-ice, and atmosphere fields: https://ecco.jpl.nasa.gov/drive/files/Version4/Release3/interp_monthly/README

1.2.2 Temporal frequency of state estimate fields

All three-dimensional ocean, sea-ice, and air-sea flux fields are provided as monthly averages. Select two-dimensional ocean and sea-ice fields are provided as daily averages. Atmospheric state fields are provided as 6-hourly records. In addition, potential temperature (theta), salinity, and free surface height anomaly at the ocean/sea-ice interface (etan) are provided as monthly snapshots to support budget closure calculations.

1.2.3 Custom output

Because the state estimate fields are the output from a free-running ocean model, users can re-run the model to generate custom output on the native lat-lon-cap model grid. Instructions for doing so are provided here: https://ecco.jpl.nasa.gov/drive/files/Version4/Release3/doc/v4r3_reproduction_howto.pdf

1.3 Python and Python Packages

The ECCO Python tutorial is compatible with Python 3. It relies on several packages including **ecco_v4-py** which include codes to facilitate loading, plotting, and performing calculations on ECCOv4 state estimate fields.

1.3.1 Why Python?

Python is an easy to learn open source programming language. In addition to the standard language library, there are thousands of free third-party modules (code libraries) available on code repositories such as Python Package Index (PyPI), _Conda and Conda Forge. Unlike commerical numerical computing environments like Matlab and IDL, Python is free for everyone to use. In addition, Python code can be run on multiple platforms such as Windows, Linux, and OS X.

Here are some links to help you learn more about Python.

- Python 3.x Documentation
- Python 3 Tutorial
- Scientific Python Lectures
- Using the NumPy module for Matlab Users
- Learning Python with Anaconda

1.3.2 Installing Python

There are several ways of installing Python on your machine. You can install compiled binaries directly from the Python website, or one can install via a package manager such as Anaconda or Miniconda. I personally find the Anaconda or Miniconda route to be simplest.

Anaconda

For scientific computing, the Anaconda Python distribution is quite convenient because it comes with a large collection of useful modules, a good open source IDE, Spyder., and the ability to open and execute Jupyter Notebooks

The latest installers for the Anaconda Distribution can be found on the Anaconda website

1.3.3 Downloading the *ecco_v4_py* Python Package

The *ecco_v4_py* package is a library of routines for analyzing the ECCO the Version 4 state estimate. The latest version can always be found on our github repository

Below are three **options** or installing the *ecco_v4_py* Python package.

Attention: Use only one of the options below!

Option 1: Clone into the repository using git (recommended)

Cloning into the *ecco_v4_py* repository using *git* is recommended because

a) you can easily see and modify source code

b) you share your improvements with the community.

To use *git* to clone into the project simply run the following commands (in the example below the Python files will go into ~/ECCOv4-py/)

```
> mkdir ~/ECCOv4-py
> cd ~/ECCOv4-py
> git clone https://github.com/ECCO-GROUP/ECCOv4-py.git
```

Option 2: Download the repository using git (less recommended)

This method downloads the source code but if you make changes it is harder to share those changes with the community using git.

```
> mkdir ~/ECCOv4-py
> cd ~/ECCOv4-py
> wget https://github.com/ECCO-GROUP/ECCOv4-py/archive/master.zip
> unzip master.zip
> rm master.zip
```

Option 3: Use the *conda* package manager (less recommended)

ecco_v4_py is available via the *conda* package management system (see https://anaconda.org/conda-forge/ecco_v4_py) Installing using conda should install all of the required dependencies.

```
conda install ecco_v4_py
```

If for some reason the above command returns an error, include the `-c` option to point to the channel where the package is found (*conda-forge*).

```
conda install -c conda-forge ecco_v4_py
```

Option 4: Use the *pip* package manager (not at all recommended)

ecco_v4_py is available via the *pip* package manager (see https://pypi.org/project/ecco-v4-py/) Before using pip, you must first install the PROJ and GEOS libraries (see next section).

```
pip install ecco_v4_py
```

1.3.4 Installing Dependencies

Danger: While conda is recommended because it automatically installs the required the GEOS (Geometry Engine) and PROJ (generic coordinate transformation software) binary libraries, you can install those libraries yourself.

Instructions for installing the GEOS library can be found on the geos website.

Instructions for installing the PROJ library can be found on the proj website.

Some users have reported difficulties installing these libraries on their platforms. For that reason, we recommend using Options 1-3.

1.3.5 Using the *ecco_v4_py* in your programs

Assuming you downloaded the *ecco_v4_py* routines to `/home/username/ECCOv4-py` then simply add these three lines to the top of your Python programs (or Jupyter Notebooks)

```
import sys
sys.path.append('/home/username/ECCOv4-py')
import ecco_v4_py as ecco
```

If you you installed the package using pip then the *ecco_v4_py* library will be automatically installed and will be ready to import into your Python program via the following commands:

```
import ecco_v4_py as ecco
```

1.4 How to get the ECCO v4 State Estimate

1.4.1 ECCO v4 r3 on PO.DAAC drive and UT Austin FTP mirror

ECCOv4 output fields are provided as NetCDF files. We have been working hard at improving these NetCDF files so that they contain more useful metadata and variable descriptions, fewer extraneous fields, and have more consistent naming.

The Python examples in this tutorial are compatible with the ECCOv4 NetCDF grid files provided in the 'Release3_alt' directories:

The ECCO v4 state estimate is now available in two places.

1. PO.DAAC drive: https://ecco.jpl.nasa.gov/drive/files/Version4/ (po.daac drive, recommended)
2. UT Austin FTP mirror: https://web.corral.tacc.utexas.edu/OceanProjects/ECCO/ECCOv4/

IMPORTANT: The NetCDF files in the ECCO Version 4 'Release 1', 'Release 2', and 'Release3' directories are not fully compatible with the tutorial examples and the ecco-v4-py library. We may update these earlier releases to be compatible with the ecco-v4-py library in the future. As of now, the 'Release3_alt' has the most up-to-date ECCO solution and file format. These files can be found here:

1. PO.DAAC drive: https://ecco.jpl.nasa.gov/drive/files/Version4/Release3_alt (po.daac drive, recommended)
2. UT Austin FTP mirror: https://web.corral.tacc.utexas.edu/OceanProjects/ECCO/ECCOv4/Release3_alt

Please see the ECCO website, ecco.jpl.nasa.gov, for udpates.

1.4.2 fields on the 13-tile lat-lon-cap (llc) *native* model grid

geometric model grid parameters

Calculations involving the state estimate variables often require the geometric model grid parameters. These parameters are packaged together as 13 grid NetCDF files, one for each llc tile, in the *nctiles_grid/* directory.

monthly-averaged ocean and sea-ice variables

Monthly-averaged ocean and sea-ice fields are in provided in subdirectories of *nctiles_monthly/*. Each subdirectory corresponds to a single variable and contains 13 NetCDF files, one for each different llc tile.

daily-averaged ocean and sea-ice variables

Daily-averaged ocean and sea-ice fields are in the subdirectories of *nctiles_daily/*. Each subdirectory corresponds to a single variable and contains 13 NetCDF files, one for each llc tile.

6-hourly atmosphere variables

6-hourly atmospheric fields that can be used as atmospheric boundary conditions for the model are provided in *input_forcing/*. Each atmospheric state variable is divided by year. These files are *not* divided into 13 tiles but are instead provided in the special *native* lat-lon-cap flat binary format required by the model. Tools for reading and plotting files in llc binary format are provided in this tutorial.

For reference, these are the atmospheric state fields provided in *input_forcing*

```
eccov4r3_dlw_YYYY                downward longwave radiation
eccov4r3_dsw_YYYY                downward shortwave radiation
eccov4r3_rain_YYYY               precipitation
eccov4r3_spfh2m_YYYY             2m near-surface atmospheric specific humidity
eccov4r3_tmp2m_degC_YYYY         2m near-surface air temperature
eccov4r3_ustr_YYYY               zonal wind stress
eccov4r3_vstr_YYYY               meridional wind stress
eccov4r3_wspeed_YYYY             near-surface wind speed
```

1.4.3 monthly-averaged *interpolated* 0.5° x 0.5° latitude-longitude variables

Monthly-averaged ocean, sea ice, and air-sea flux terms are in the subdirectory *interp_monthly/*. Each NetCDF file in *interp_monthly* corresponds to a single variable.

1.4.4 Downloading the State Estimate

ECCO v4 solutions are now hosted on the PO.DAAC drive. This service is very useful because one can mount the ECCO file directory on PO.DAAC drive to your local machine.

You can even use wget to download files through PO.DAAC drive. See the ECCO website for more details: https://ecco-group.org

Take note of the location of your files. You'll need to specify their path location to load them in the tutorial.

1.5 Tutorial Overview

1.5.1 What is the format of the tutorials?

All tutorials are provided as Python codes in two formats: (1) standard Python (*.py files) and (2) Python Jupyter Notebooks (*.ipynb files).

Tutorials as Jupyter Notebooks: https://github.com/ECCO-GROUP/ECCO-v4-Python-Tutorial/tree/master/Tutorials_as_Jupyter_Notebooks

Tutorials as Python Files: https://github.com/ECCO-GROUP/ECCO-v4-Python-Tutorial/tree/master/Tutorials_as_Python_Files

Because all of the tutorials are actually Python code, every one can be reproduced locally on your own machine. You'll only need to change the tutorials so that they point to the location of the state estimate output files on your disk and the location of your copy of the *ecco_v4_py* Python package (only if you did not install it with the pip Python package manager).

What are Jupyter notebooks?

From the Jupyter website:

"The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more."

Jupyter notebooks allow Python code to be run in an interactive environment within a browser window. Notebooks are divided in *cells* which can include text or Python code. They are similar to the "command windows" of interactive desktop environments (e.g, Matlab) except that both the text, code, and the output of commands (textual output or figures) are kept together in the same document. One of the best features of Jupyter notebooks is that they store all commands and output. Loading someone else's notebook allows you to reproduce and build upon their work. We hope you take advantage of this feature of the ECCO v4 tutorial!

If you want to see some examples of Juypter Notebooks before continuing further, here are some examples (1) numerical intergration with the Trapezoid Rule, (2) numerical integration with the Crank Nicolson method, (3) symbolic calculations.

This notebook basics page may be helpful for learning how to navigate around in notebooks.

1.5.2 Will I learn Python just from reading these tutorials?

Unlikely! These tutorials are not a comprehensive introduction to Python. **Nevertheless, throughout the tutorial there are many helpful tips to guide the non-native Python speaker.** These tips come early on so it helps to start from the beginning.

1.5.3 What Python should I review before getting started?

There are thousands of Python packages, many of which would no doubt be useful for working with the ECCO v4 state estimate. Howefver, there are *three* that you will should familiarze yourself with on some level before starting: **NumPy**, **Matplotib**, and **xarray**. **NumPy** provides n-dimensional matrices and matrix operations, **Matplotlib** provides plotting tools, and **xarray** provides a framework for labelling and easily accessing subsets of the **NumPy** n-dimensional matrices. **xarray** is not necessary for working with ECCO v4 output as one does not need to have labeled dimensions and coordinatess to analyze arrays. Nevertheless, you may find that working with arrays that have well labeled coordinates and dimensions makes life much, much easier.

NumPy

From the NumPy website

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

a powerful N-dimensional array object sophisticated (broadcasting) functions tools for integrating C/C++ and Fortran code useful linear algebra, Fourier transform, and random number capabilities

Matplotlib

From the Matplotlib website

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shell, the jupyter notebook, web application servers, and four graphical user interface toolkits.

Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code.

For simple plotting the pyplot module provides a MATLAB-like interface, [...]. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

xarray

From the xarray website

Adding dimensions names and coordinate indexes to numpy's ndarray makes many powerful array operations possible.

The N-dimensional nature of xarray's data structures makes it suitable for dealing with multi-dimensional scientific data, and its use of dimension names instead of axis labels (`dim='time'` instead of `axis=0`) makes such arrays much more manageable than the raw numpy ndarray: [...]

1.5.4 What if I don't like the way you do X?

That's ok. We're always open to suggestions.

1.5.5 What if I find a mistake?

Definitely please tell us! You can write Ian.Fenty at jpl.nasa.gov

1.5.6 What if I would like to contribute with a tutorial of my own?

That would be fantastic! We're all in this together. If you have a notebook that you think would be helpful, let us know and we'll do our best to integrate it.

1.5.7 Bonus Tutorials

Two valuable tutorials are included at the end as **Bonus Tutorials**. Both bonus tutorials use the xmitgcm package to load MITgcm model output. Consequently, some of the synatax of their calculations will be different than the syntax used in the main tutorial. For the reader who has completed the main tutorial the bonus tutorials will be easy to understand.

1) "Evaluating budgets in the ECCOv4 model run using xgcm" by Erik Tesdal
2) "Vector calculus in ECCO: The Transport, divergence, vorticity and the Barotropic Vorticity Budget" by Maike Sonnewald

1.6 The Dataset and DataArray objects used in the ECCOv4 Python package.

1.6.1 Objectives

To introduce the two high-level data structures, `Dataset` and `DataArray`, that are used in by the `ecco_v4_py` Python package to load and store the ECCO v4 model grid parameters and state estimate variables.

1.6.2 Introduction

The ECCOv4 files are provided as NetCDF files. The file you have may look a little different than the ones shown here because we have been working hard at improving how what exactly goes into our NetCDF files.

This tutorial document is current as of Sep 2019 with the ECCOv4 NetCDF grid files provided in the following directories:

https://ecco.jpl.nasa.gov/drive/files/Version4/Release3_alt (po.daac drive, recommended)

https://web.corral.tacc.utexas.edu/OceanProjects/ECCO/ECCOv4/Release3_alt/ (mirror at U. Texas, Austin)

In this first tutorial we will start slowly, providing detail at every step. Later tutorials will be assume knowledge of some basic operations introduced here.

Let's get started.

1.6.3 Import external packages and modules

Before using Python libraries we must import them. Usually this is done at the beginning of every Python program or interactive Juypter notebook instance but one can import a library at any point in the code. Python libraries, called **packages**, contain subroutines and/or define data structures that provide useful functionality.

Before we go further, let's import some packages needed for this tutorial:

[1]:
```
# NumPy is the fundamental package for scientific computing with Python.
# It contains among other things:
#    a powerful N-dimensional array object
#    sophisticated (broadcasting) functions
#    tools for integrating C/C++ and Fortran code
#    useful linear algebra, Fourier transform, and random number capabilities
# http://www.numpy.org/
#
# make all functions from the 'numpy' module available with the prefix 'np'
import numpy as np

# xarray is an open source project and Python package that aims to bring the
# labeled data power of pandas to the physical sciences, by providing
# N-dimensional variants of the core pandas data structures.
# Our approach adopts the Common Data Model for self- describing scientific
# data in widespread use in the Earth sciences: xarray.Dataset is an in-memory
# representation of a netCDF file.
# http://xarray.pydata.org/en/stable/
#
# import all function from the 'xarray' module available with the prefix 'xr'
import xarray as xr
```

Load the ECCO Version 4 Python package

The *ecco_v4_py* is a Python package written specifically for working with the NetCDF output provided in the nctiles_monthly directory of the ECCO v4 release

See the "Getting Started" page in the tutorial for instructions about installing the *ecco_v4_py* module on your machine.

[2]:
```
## Import the ecco_v4_py library into Python
## =========================================
## -- If ecco_v4_py is not installed in your local Python library,
##    tell Python where to find it.  For example, if your ecco_v4_py
##    files are in /home/username/ECCOv4-py/ecco_v4_py, then use:
import sys
sys.path.append('/home/ifenty/ECCOv4-py')
import ecco_v4_py as ecco
```

The syntax

```
import XYZ package as ABC
```

allows you to access all of the subroutines and/or objects in a package with perhaps a long complicated name with a shorter, easier name.

Here, we import `ecco_v4_py` as `ecco` because typing `ecco` is easier than `ecco_v4_py` every time. Also, `ecco_v4_py` is actually comprised of multiple python modules and by importing just `ecco_v4_py` we can actually access all of the subroutines in those modules as well. Fancy.

1.6.4 Load a single state estimate variable NetCDF tile file

To load ECCO v4's NetCDF files we will use the *open_dataset* command from the Python package xarray. The *open_dataset* routine creates a `Dataset` object and loads the contents of the NetCDF file, including its metadata, into a data structure.

Let's open one monthly mean THETA file associated with *tile 2* (the North East Atlantic Ocean).

[3]:
```
## Set top-level file directory for the ECCO NetCDF files
## =======================================================
# base_dir = '/home/username/'
base_dir = '/home/ifenty/ECCOv4-release'

## define a high-level directory for ECCO fields
ECCO_dir = base_dir + '/Release3_alt'
```

[4]:
```
## LOAD NETCDF FILE
## ================

# directory containing the file
data_dir= ECCO_dir + '/nctiles_monthly/THETA/'

# filename
fname = 'THETA_2010.nc'

# load the file
ds = xr.open_dataset(data_dir + fname).load()
```

What is *ds*? It is a `Dataset` object which is defined somewhere deep in the `xarray` package:

[5]:
```
type(ds)
```

[5]:
```
xarray.core.dataset.Dataset
```

1.6.5 The Dataset object

According to the xarray documentation, a Dataset is a Python object designed as an "in-memory representation of the data model from the NetCDF file format."

What does that mean? NetCDF files are *self-describing* in the sense that they include information about the data they contain. When `Datasets` are created by loading a NetCDF file they load all of the same data and metadata.

Just as a NetCDF file can contain many variables, a `Dataset` can contain many variables. These variables are referred to as `Data Variables` in the `xarray` nomenclature.

`Datasets` contain three main classes of fields:

1. **Coordinates** : arrays identifying the coordinates of the data variables
2. **Data Variables**: the data variable arrays and their associated coordinates
3. **Attributes** : metadata describing the dataset

Now that we've loaded `GRID.0003.nc` as the *ds* `Dataset` object let's examine its contents.

> **Note:** *You can get information about objects and their contents by typing the name of the variable and hitting* **enter** *in an interactive session of an IDE such as Spyder or by executing the cell of a Jupyter notebook.*

[6]:
```
ds
```

[6]:
```
<xarray.Dataset>
Dimensions:    (i: 90, j: 90, k: 50, nv: 2, tile: 13, time: 12)
Coordinates:
  * j          (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * i          (i) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k          (k) int32 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
    Z          (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    PHrefC     (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
    drF        (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    XC         (tile, j, i) float32 -111.60647 -111.303 ... -111.86579
    YC         (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
    rA         (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
    hFacC      (tile, k, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
  * tile       (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
    time_bnds  (time, nv) datetime64[ns] 2010-01-01 2010-02-01 ... 2011-01-01
    iter       (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
  * time       (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
Dimensions without coordinates: nv
Data variables:
    THETA      (time, tile, k, j, i) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
Attributes:
    product_time_coverage_start:  1992-01-01T12:00:00
    author:                       Ian Fenty and Ou Wang
    Insitution:                   JPL
    product_version:              ECCO Version 4 Release 3 (ECCOv4r3) 1992-2015
```

(continues on next page)

(continued from previous page)

```
    time_units:                 days since 1992-01-01 00:00:00
    Conventions:                CF-1.6
    Project:                    Estimating the Circulation and Climate of t...
    cdm_data_type:              Grid
    geospatial_lon_units:       degrees_east
    Metadata_Conventions:       CF-1.6, Unidata Dataset Discovery v1.0, GDS...
    no_data:                    NaNf
    geospatial_lat_units:       degrees_north
    product_time_coverage_end:  2015-12-31T12:00:00
    geospatial_vertical_min:    0
    nz:                         50
    geospatial_vertical_units:  meter
    geospatial_vertical_max:    6134.5
    date_created:               Mon May 13 02:40:10 2019
    geospatial_lat_max:         89.739395
    geospatial_lat_min:         -89.873055
    nx:                         90
    ny:                         90
    geospatial_lon_max:         179.98691
    geospatial_lon_min:         -179.98895
    time_coverage_start:        2010-01-01T00:00:00
    time_coverage_end:          2011-01-01T00:00:00
```

Examining the Dataset object contents

Let's go through *ds* piece by piece, starting from the top.

1. Object type

```
<xarray.Dataset>
```

The top line tells us what type of object the variable is. *ds* is an instance of a`Dataset` defined in `xarray`.

2. Dimensions

```
Dimensions: (i: 90, j: 90, k: 50, nv: 2, tile: 13, time: 12)
```

The *Dimensions* list shows all of the different dimensions used by all of the different arrays stored in the NetCDF file (and now loaded in the `Dataset` object).

Arrays may use any combination of these dimensions. In the case of this *grid* datasets, we find 1D (e.g., depth), 2D (e.g., lat/lon), and 3D (e.g., mask) arrays.

The lengths of these dimensions are next to their name: `(i: 90, j: 90, k: 50, nv: 2, tile: 13, time: 12)`. There are 50 vertical levels in the ECCO v4 model grid so the `k` corresponds to vertical dimension. `i` and `j` correspond to the horizontal dimensions. The lat-lon-cap grid has 13 tiles. This THETA file has 12 monthly-mean records for 2010. The dimension `nv` is a time dimension that corresponds to the start and end times of the monthly-mean averaging periods. In other words, for every 1 month, there are 2 (nv = 2) time records, one describing when the month started and the other when the month ended.

> **Note:** Each tile in the llc90 grid used by ECCO v4 has 90x90 horizontal grid points. That's where the 90 in llc**90** comes from!

3. Coordinates

Some coordinates have an asterix **"*"** in front of their names. They are known as *dimension coordinates* and are always one-dimensional arrays of length n which specify the length of arrays in the dataset in different dimensions.

```
Coordinates:
  * j          (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * i          (i) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k          (k) int32 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * tile       (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
  * time       (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
```

These coordinates are arrays whose values *label* each grid cell in the arrays. They are used for *label-based indexing* and *alignment*.

Let's look at the three primary spatial coordiates, i, j, k.

[7]:
```
print(ds.i.long_name)
print(ds.j.long_name)
print(ds.k.long_name)
```

```
x-dimension of the t grid
y-dimension of the t grid
z-dimension of the t grid
```

`i` indexes (or labels) the tracer grid cells in the `x` direction, `j` indexes the tracer grid cells in the `y` direction, and similarly `k` indexes the tracer grid cells in the `z` direction.

4. Data Variables

```
Data variables:
    THETA      (time, tile, k, j, i) float32 ...
```

The *Data Variables* are one or more `xarray.DataArray` objects. `DataArray` objects are labeled, multi-dimensional arrays that may also contain metadata (attributes). `DataArray` objects are very important to understand because they are container objects which store the numerical arrays of the state estimate fields. We'll investigate these objects in more detail after completing our survey of this `Dataset`.

In this NetCDF file there is one *Data variables*, `THETA`, which is stored as a five dimensional array (**time, tile, k,j,i**) field of average potential temperature. The llc grid has 13 tiles. Each tile has two horizontal dimensions (i,j) and one vertical dimension (k).

`THETA` is stored here as a 32 bit floating point precision.

> **Note:** The meaning of all MITgcm grid parameters can be found here.

5. Attributes

```
Attributes:
    product_time_coverage_start:  1992-01-01T12:00:00
    author:                       Ian Fenty and Ou Wang
    Insitution:                   JPL
    product_version:              ECCO Version 4 Release 3 (ECCOv4r3) 1992-2015
    time_units:                   days since 1992-01-01 00:00:00
    Conventions:                  CF-1.6
    Project:                      Estimating the Circulation and Climate of t...
    cdm_data_type:                Grid
    geospatial_lon_units:         degrees_east
    Metadata_Conventions:         CF-1.6, Unidata Dataset Discovery v1.0, GDS...
    no_data:                      NaNf
    geospatial_lat_units:         degrees_north
    product_time_coverage_end:    2015-12-31T12:00:00
    geospatial_vertical_min:      0
    nz:                           50
    geospatial_vertical_units:    meter
    geospatial_vertical_max:      6134.5
    date_created:                 Mon May 13 02:40:10 2019
    geospatial_lat_max:           89.739395
    geospatial_lat_min:           -89.873055
    nx:                           90
    ny:                           90
    geospatial_lon_max:           179.98691
    geospatial_lon_min:           -179.98895
    time_coverage_start:          2010-01-01T00:00:00
    time_coverage_end:            2011-01-01T00:00:00
```

The `attrs` variable is a Python dictionary object containing metadata or any auxilliary information.

Metadata is presented as a set of dictionary `key-value` pairs. Here the `keys` are *description, A, B, … missing_value.* while the `values` are the corresponding text and non-text values.

To see the metadata `value` associated with the metadata `key` called "Conventions" we can print the value as follows:

[8]:
```
print (ds.attrs['Conventions'])
```

```
CF-1.6
```

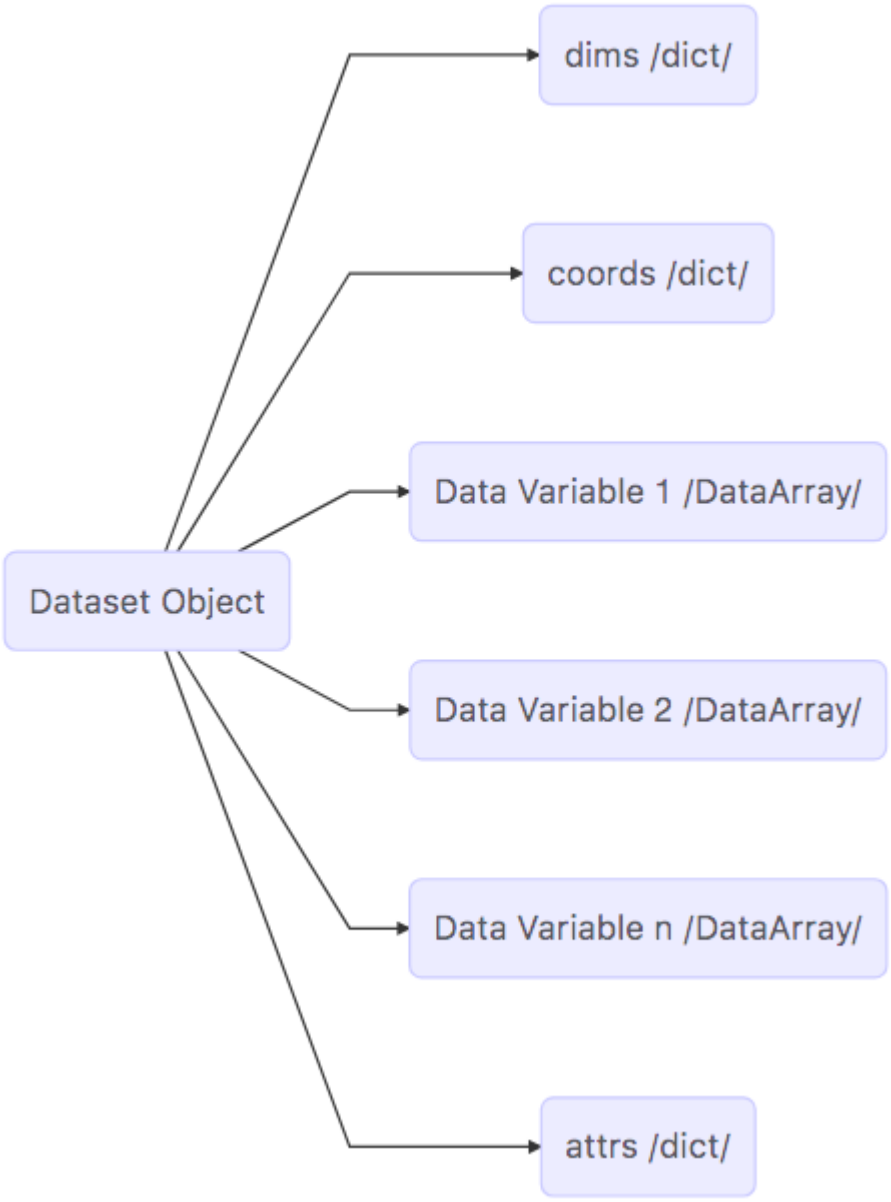
"CF-1.6" tells us that ECCO NetCDF output conforms to the Climate and Forecast Conventions version 1.6. How convenient.

Map of the `Dataset` object

Now that we've completed our survey, we see that a `Dataset` is a really a kind of *container* comprised of (actually pointing to) many other objects.

- dims: A `dict` that maps dimension names (keys) with dimension lengths (values)
- coords: A `dict` that maps dimension names (keys such as **k, j, i**) with arrays that label each point in the dimension (values)
- One or more *Data Variables* that are pointers to `DataArray` objects

- attrs A `dict` that maps different attribute names (keys) with the attributes themselves (values).



1.6.6 The `DataArray` Object

It is worth looking at the `DataArray` object in more detail because `DataArrays` store the arrays that store the ECCO output. Please see the xarray documentation on the DataArray object for more information.

`DataArrays` are actually very similar to `Datasets`. They also contain dimensions, coordinates, and attributes. The two main differences between `Datasets` and `DataArrays` is that `DataArrays` have a **name** (a string) and an array of **values**. The **values** array is a numpy n-dimensional array, an `ndarray`.

Examining the contents of a `DataArray`

Let's examine the contents of one of the coordinates `DataArrays` found in *ds*, *XC*.

[9]:
```
ds.XC
```

[9]:
```
<xarray.DataArray 'XC' (tile: 13, j: 90, i: 90)>
array([[[-111.60647 , -111.303   , -110.94285 , ...,   64.791115,
           64.80521 ,   64.81917 ],
        [-104.8196  , -103.928444, -102.87706 , ...,   64.36745 ,
           64.41012 ,   64.4524  ],
        [ -98.198784,  -96.788055,  -95.14185 , ...,   63.936497,
           64.008224,   64.0793  ],
        ...,
        [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
        [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
        [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ]],

       [[ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
        [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
        [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
        ...,
        [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
        [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
        [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ]],

       [[ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
        [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
        [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
        ...,
        [ -37.730072,  -37.17829 ,  -36.597565, ...,   50.597565,
           51.17829 ,   51.730072],
        [ -37.771988,  -37.291943,  -36.764027, ...,   50.764027,
           51.291943,   51.771988],
        [ -37.837925,  -37.44421 ,  -36.968143, ...,   50.968143,
           51.44421 ,   51.837925]],

       ...,

       [[-127.83792 , -127.77199 , -127.73007 , ..., -127.5     ,
         -127.5     , -127.5     ],
        [-127.44421 , -127.29195 , -127.17829 , ..., -126.5     ,
```

(continues on next page)

(continued from previous page)

```
         -126.5     , -126.5     ],
        [-126.96814 , -126.76402 , -126.597565, ..., -125.5     ,
         -125.5     , -125.5     ],
        ...,
        [ -39.031857,  -39.235973,  -39.402435, ...,  -40.5     ,
          -40.5     ,  -40.5     ],
        [ -38.55579 ,  -38.708057,  -38.82171 , ...,  -39.5     ,
          -39.5     ,  -39.5     ],
        [ -38.162075,  -38.228012,  -38.269928, ...,  -38.5     ,
          -38.5     ,  -38.5     ]],

       [[-127.5     , -127.5     , -127.5     , ..., -127.5     ,
         -127.5     , -127.5     ],
        [-126.5     , -126.5     , -126.5     , ..., -126.5     ,
         -126.5     , -126.5     ],
        [-125.5     , -125.5     , -125.5     , ..., -125.5     ,
         -125.5     , -125.5     ],
        ...,
        [ -40.5     ,  -40.5     ,  -40.5     , ...,  -40.5     ,
          -40.5     ,  -40.5     ],
        [ -39.5     ,  -39.5     ,  -39.5     , ...,  -39.5     ,
          -39.5     ,  -39.5     ],
        [ -38.5     ,  -38.5     ,  -38.5     , ...,  -38.5     ,
          -38.5     ,  -38.5     ]],

       [[-127.5     , -127.5     , -127.5     , ..., -115.850204,
         -115.50567 , -115.166985],
        [-126.5     , -126.5     , -126.5     , ..., -115.78025 ,
         -115.464066, -115.153244],
        [-125.5     , -125.5     , -125.5     , ..., -115.71079 ,
         -115.42275 , -115.139595],
        ...,
        [ -40.5     ,  -40.5     ,  -40.5     , ..., -101.42989 ,
         -106.83081 , -112.28605 ],
        [ -39.5     ,  -39.5     ,  -39.5     , ..., -100.48844 ,
         -106.24874 , -112.090065],
        [ -38.5     ,  -38.5     ,  -38.5     , ...,  -99.42048 ,
         -105.58465 , -111.86579 ]]], dtype=float32)
Coordinates:
  * j        (j) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * i        (i) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
    XC       (tile, j, i) float32 -111.60647 -111.303 ... -105.58465 -111.86579
    YC       (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
    rA       (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
  * tile     (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
Attributes:
    units:          degrees_east
    long_name:      longitude at center of tracer cell
    standard_name:  longitude_at_c_location
    valid_range:    -180., 180.
```

Examining the DataArray

The layout of DataArrays is very similar to those of Datasets. Let's examine each part of *ds.XC*, starting from the top.

1. Object type

<xarray.DataArray>

This is indeed a DataArray object from the xarray package.

Note: You can also find the type of an object with the type command: print type(ds.XC)

[10]:
```
print (type(ds.XC))
```

```
<class 'xarray.core.dataarray.DataArray'>
```

2. Object Name

XC

The top line shows DataArray name, XC.

3. Dimensions

(tile: 13, j: 90, i: 90)

Unlike *THETA*, *XC* does not have time or depth dimensions which makes sense since the longitude of the grid cell centers do not vary with time or depth.

4. The numpy Array

```
array([[[-111.60647 , -111.303   , -110.94285 , ...,   64.791115,
           64.80521 ,   64.81917 ],
        [-104.8196  , -103.928444, -102.87706 , ...,   64.36745 ,
           64.41012 ,   64.4524  ],
        [ -98.198784,  -96.788055,  -95.14185 , ...,   63.936497,
           64.008224,   64.0793  ],
        ...,
```

In Dataset objects there are *Data variables*. In DataArray objects we find numpy **arrays**. Python prints out a subset of the entire array.

Note: DataArrays store **only one** array while DataSets can store **one or more** DataArrays.

We access the numpy array by invoking the .values command on the DataArray.

[11]:
```
print(type(ds.XC.values))
```

```
<class 'numpy.ndarray'>
```

The array that is returned is a numpy n-dimensional array:

[12]:
```
type(ds.XC.values)
```

[12]:
```
numpy.ndarray
```

Being a numpy array, one can use all of the numerical operations provided by the numpy module on it.

> ** Note: ** You may find it useful to learn about the operations that can be made on numpy arrays. Here is a quickstart guide: https://docs.scipy.org/doc/numpy-dev/user/quickstart.html

We'll learn more about how to access the values of this array in a later tutorial. For now it is sufficient to know how to access the arrays!

4. Coordinates

The dimensional coordinates (with the asterixes) are

```
Coordinates:
  * j        (j) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * i        (i) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * tile     (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
```

We find three 1D arrays with coordinate labels for **j**, **i**, and **tile**.

[13]:
```
ds.XC.coords
```

[13]:
```
Coordinates:
  * j        (j) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * i        (i) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
    XC       (tile, j, i) float32 -111.60647 -111.303 ... -105.58465 -111.86579
    YC       (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
    rA       (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
  * tile     (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
```

two other important coordinates here are `tile` and `time`

[14]:
```
print('tile: ')
print(ds.tile.values)
print('time: ')
print(ds.time.values)
```

```
tile:
[ 0  1  2  3  4  5  6  7  8  9 10 11 12]
time:
['2010-01-16T12:00:00.000000000' '2010-02-15T12:00:00.000000000'
 '2010-03-16T12:00:00.000000000' '2010-04-16T12:00:00.000000000'
 '2010-05-16T12:00:00.000000000' '2010-06-15T12:00:00.000000000'
 '2010-07-16T12:00:00.000000000' '2010-08-16T12:00:00.000000000'
 '2010-09-15T12:00:00.000000000' '2010-10-16T12:00:00.000000000'
 '2010-11-16T12:00:00.000000000' '2010-12-16T12:00:00.000000000']
```

The file we loaded was `/nctiles_mean/THETA/THETA_2010.nc`, the 2010 monthly-mean potential temperature field. Here the time coordinates are the center of the averaging periods.

5. Attributes

```
Attributes:
    units:          degrees_east
    long_name:      longitude at center of tracer cell
    standard_name:  longitude_at_c_location
    valid_range:    -180., 180.
```

The XC variable has a long_name (longitude at center of tracer cell) and units (degrees_east) and other information. This metadata was loaded from the NetCDF file. The entire attribute dictionary is accessed using .attrs.

[15]:
```
ds.XC.attrs
```

[15]:
```
{'units': 'degrees_east',
 'long_name': 'longitude at center of tracer cell',
 'standard_name': 'longitude_at_c_location',
 'valid_range': '-180., 180.'}
```

[16]:
```
ds.XC.attrs['units']
```

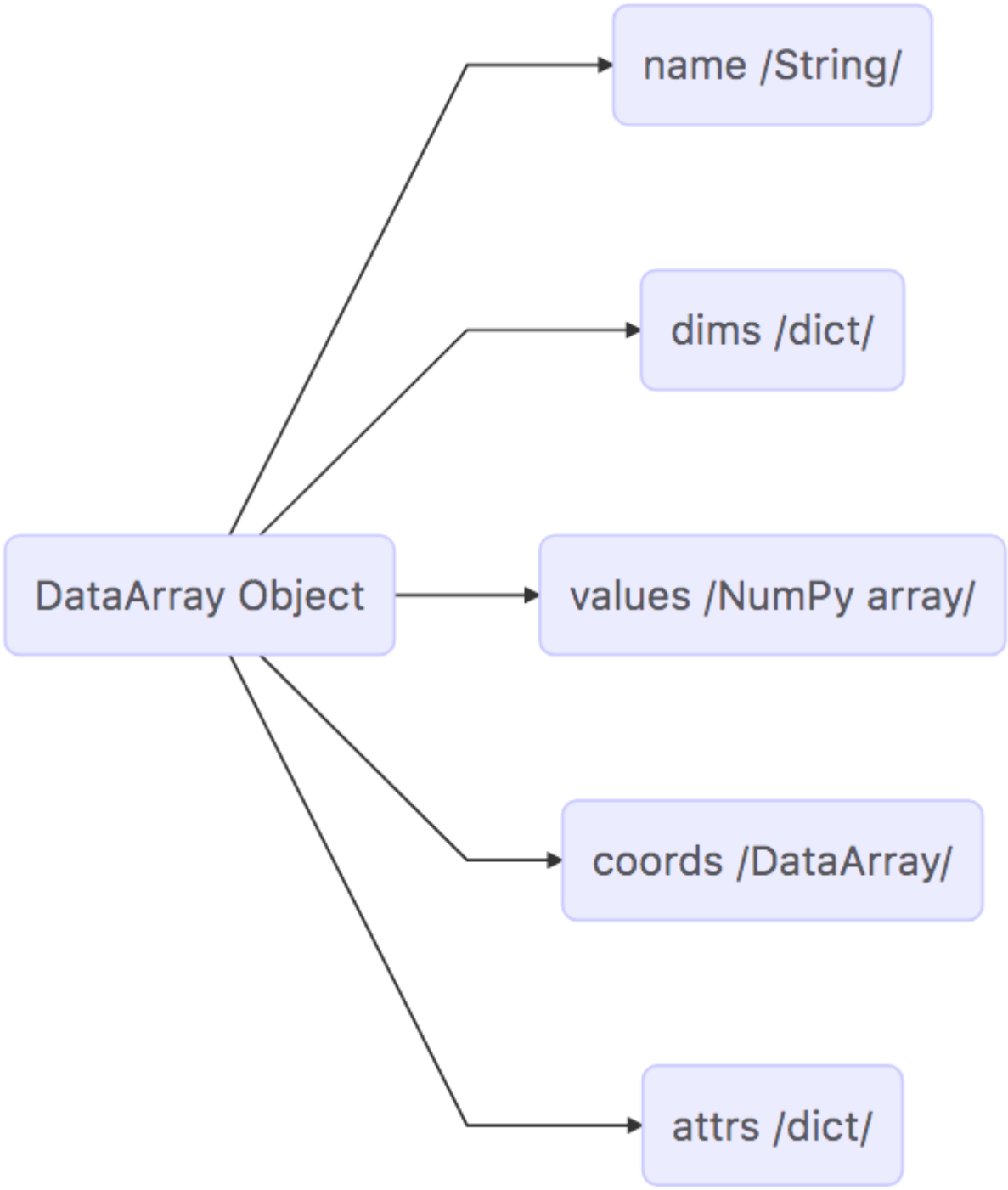
[16]:
```
'degrees_east'
```

[17]:
```
ds.XC.attrs['valid_range']
```

[17]:
```
'-180., 180.'
```

Map of the DataArray Object

The DataArray can be mapped out with the following diagram:


name /String/
dims /dict/
DataArray Object
values /NumPy array/
coords /DataArray/
attrs /dict/

1.6.7 Summary

Now you know the basics of the `Dataset` and `DataArray` objects that will store the ECCO v4 model grid parameters and state estimate output variables. Go back and take a look athe grid *ds* object that we originally loaded. It should make a lot more sense now!

1.7 Coordinates and Dimensions of ECCOv4 NetCDF files

1.7.1 Objectives

Introduce the student ot the idea that the ECCO v4 NetCDF fields have coordinate labels which shows *where* these fields are on the Arakawa-C grid.

1.7.2 Introduction

The ECCOv4 files are provided as NetCDF files. The file you have may look a little different than the ones shown here because we have been working hard at improving how what exactly goes into our NetCDF files.

This tutorial document is current as of Sep 2019 with the ECCOv4 NetCDF grid files provided in the following directories:

https://ecco.jpl.nasa.gov/drive/files/Version4/Release3_alt (po.daac drive, recommended)

https://web.corral.tacc.utexas.edu/OceanProjects/ECCO/ECCOv4/Release3_alt/ (mirror at U. Texas, Austin)

As we showed in the first tutorial, we can use the `open_dataset` method from `xarray` to load a NetCDF tile file into Python as a `Dataset` object. `open_dataset` is very convenient because it automatically parses the NetCDF file and constructs a `Dataset` object using all of the dimensions, coordinates, variables, and metadata information.

In the last tutorial we analyzed the contents of a single ECCOv4 file, the 2010 monthly-averaged potential temperature. Let's load it up again and take a closer look at its coordinates. This time we'll name the new `Dataset` object *theta_dataset* since we are loading the file using `open_dataset`.

[1]:
```
import numpy as np
import xarray as xr
import sys
import matplotlib.pyplot as plt
%matplotlib inline
import json
```

[2]:
```
## Import the ecco_v4_py library into Python
## =========================================
## -- If ecco_v4_py is not installed in your local Python library,
##    tell Python where to find it.  For example, if your ecco_v4_py
##    files are in /Users/ifenty/ECCOv4-py/ecco_v4_py, then use:
sys.path.append('/home/ifenty/ECCOv4-py')

import ecco_v4_py as ecco
```

[3]:
```
## Set top-level file directory for the ECCO NetCDF files
## ======================================================
# base_dir = '/home/username/'
```

(continues on next page)

(continued from previous page)

```
base_dir = '/home/ifenty/ECCOv4-release'

## define a high-level directory for ECCO fields
ECCO_dir = base_dir + '/Release3_alt'
```

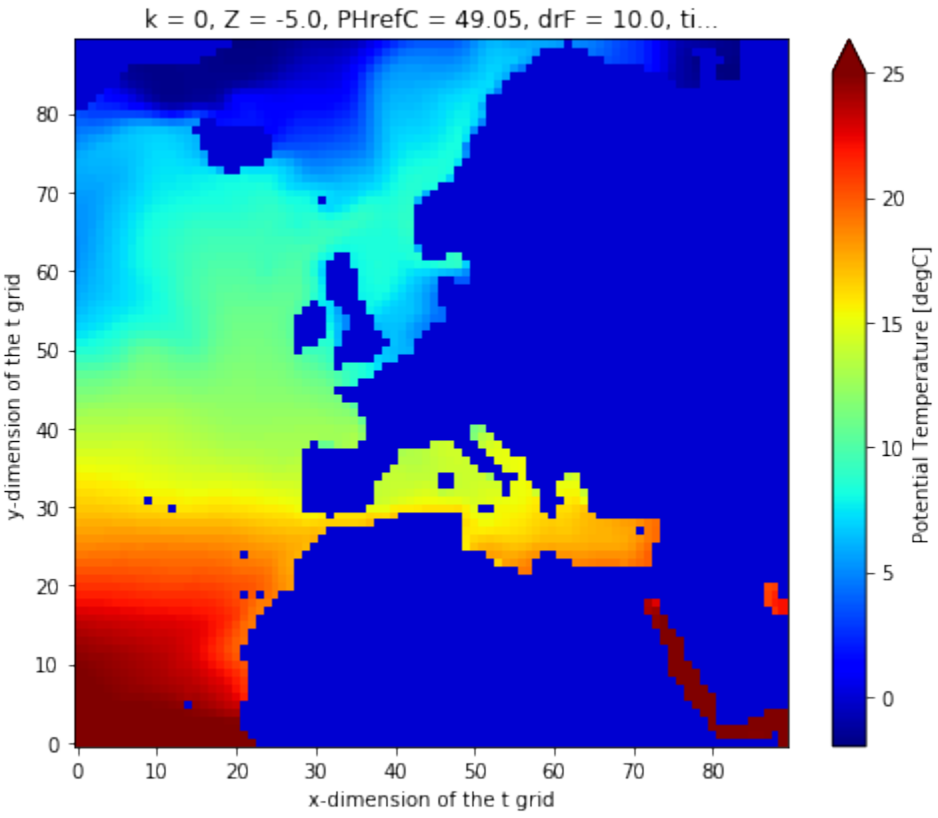
[4]:
```
## LOAD NETCDF FILE
## ================

# directory containing the file
data_dir= ECCO_dir + '/nctiles_monthly/THETA/'

# filename
fname = 'THETA_2010.nc'

# load the file
theta_dataset = xr.open_dataset(data_dir + fname).load()
```

Before we get started, plot the temperature field at the surface layer (k=0) for tile 2, NE Atlantic..

Note :: Don't worry about the complicated looking code below, w'll cover plotting later

[5]:
```
fig=plt.figure(figsize=(8, 6.5))
theta_dataset.THETA.isel(k=0,tile=2,time=0).plot(vmin=-2, vmax=25, cmap='jet')
```

[5]:
```
<matplotlib.collections.QuadMesh at 0x7f85134ef160>
```



1.7.3 The Dimensions and Coordinates of *THETA*

Let's take a closer look at what is inside this dataset. We suppress the metadata (attrs) just to reduce how much is printed to the screen.

[6]:
```
theta_dataset.attrs = []
theta_dataset
```

[6]:
```
<xarray.Dataset>
Dimensions:    (i: 90, j: 90, k: 50, nv: 2, tile: 13, time: 12)
Coordinates:
  * j          (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * i          (i) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k          (k) int32 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
    Z          (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    PHrefC     (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
    drF        (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    XC         (tile, j, i) float32 -111.60647 -111.303 ... -111.86579
    YC         (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
    rA         (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
    hFacC      (tile, k, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
  * tile       (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
    time_bnds  (time, nv) datetime64[ns] 2010-01-01 2010-02-01 ... 2011-01-01
    iter       (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
  * time       (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
Dimensions without coordinates: nv
Data variables:
    THETA      (time, tile, k, j, i) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
```

Dimensions

theta_dataset shows six dimensions, **i**, **j**, **k**, **tile**, **time**, and **nv**. Recall that `Dataset` objects are containers and so it lists all of the **unique** dimensions of the variables it is storing. *theta_dataset* is storing a single *Data variable*, *THETA*. We see that this *THETA* field is five dimensional from the (**tile**, **time**, **k**, **j**, **i**) in the line: ~~~ THETA (time, tile, k, j, i) float32 0.0 0.0 0.0 0.0 … 0.0 0.0 0.0 0.0 ~~~ Examining the coordinates of the `Dataset`, we find that all of them have some combination of the six dimensions. Note **nv** is not a spatial or temporal dimension per se. It is a kind of dummy dimension of length 2 for the coordiante **time_bnds** which has both a starting and ending time for each one averaging period.

Coordinates

Dimension Coordinates

Beyond having three spatial dimensions *theta_dataset* also has *coordinates* in the **i**, **j**, and **k** directions. The most basic coordinates are 1D vectors, one for each dimension, which contain *indices* for the array. Let us call these basic coordinates, *dimension coordinates*. Here we use 0 as the first index of dimension coordinates

```
* j          (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
* i          (i) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
* k          (k) int32 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
* tile       (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
* time       (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
```

Let's examine each *Dimension Coordinate* more closely

Dimension Coordinate `i`

[7]:
```
theta_dataset.i
```

[7]:
```
<xarray.DataArray 'i' (i: 90)>
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
       36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53,
       54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
       72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
      dtype=int32)
Coordinates:
  * i        (i) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
Attributes:
    long_name:      x-dimension of the t grid
    standard_name:  x_grid_index
    swap_dim:       XC
    axis:           X
```

i is an array of integers from 0 to 89 indicating the *x_grid_index* along this tile's *X* axis.

Dimension Coordinate `j`

[8]:
```
theta_dataset.j
```

[8]:
```
<xarray.DataArray 'j' (j: 90)>
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
       36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53,
       54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
       72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
      dtype=int32)
Coordinates:
  * j        (j) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
Attributes:
    long_name:      y-dimension of the t grid
    standard_name:  y_grid_index
    swap_dim:       YC
    axis:           Y
```

j is an array of integers from 0 to 89 indicating the *y_grid_index* along this tile's *Y* axis.

Dimension Coordinate `k`

[9]:
```
theta_dataset.k
```

[9]:
```
<xarray.DataArray 'k' (k: 50)>
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
       36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49], dtype=int32)
Coordinates:
  * k        (k) int32 0 1 2 3 4 5 6 7 8 9 10 ... 40 41 42 43 44 45 46 47 48 49
    Z        (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    PHrefC   (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
    drF      (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
Attributes:
    long_name:      z-dimension of the t grid
    standard_name:  z_grid_index
    swap_dim:       Z
    axis:           Z
```

k is an array of integers from 0 to 49 indicating the *z_grid_index* along this tile's *Z* axis.

Dimension Coordinate `tile`

[10]:
```
theta_dataset.tile
```

[10]:
```
<xarray.DataArray 'tile' (tile: 13)>
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12], dtype=int32)
Coordinates:
  * tile     (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
Attributes:
    standard_name:  tile_index
```

tile is an array of integers from 0 to 12, one for each tile of the lat-lon-cap grid.

Dimension Coordinate `time`

[11]:
```
theta_dataset.time
```

[11]:
```
<xarray.DataArray 'time' (time: 12)>
array(['2010-01-16T12:00:00.000000000', '2010-02-15T12:00:00.000000000',
       '2010-03-16T12:00:00.000000000', '2010-04-16T12:00:00.000000000',
       '2010-05-16T12:00:00.000000000', '2010-06-15T12:00:00.000000000',
       '2010-07-16T12:00:00.000000000', '2010-08-16T12:00:00.000000000',
       '2010-09-15T12:00:00.000000000', '2010-10-16T12:00:00.000000000',
       '2010-11-16T12:00:00.000000000', '2010-12-16T12:00:00.000000000'],
      dtype='datetime64[ns]')
Coordinates:
    iter     (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
  * time     (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
Attributes:
    long_name:      center time of averaging period
```

(continues on next page)

(continued from previous page)

```
    standard_name:  time
    bounds:         time_bnds
    axis:           T
```

In this file the *time* coordinate indicates the *center time of the averaging period*. Recall that we loaded the monthly-mean `THETA` fields for 2010, so the *center time of the averaging periods* are the middle of each month in 2010.

Other Coordinates

Notice some *coordinates* do not have an **"*"** in front of their names:

```
Coordinates:
    Z          (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    PHrefC     (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
    drF        (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    XC         (tile, j, i) float32 -111.60647 -111.303 ... -111.86579
    YC         (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
    rA         (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
    hFacC      (tile, k, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    time_bnds  (time, nv) datetime64[ns] 2010-01-01 2010-02-01 ... 2011-01-01
    iter       (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
```

These are so-called non-dimension coordinates. From the xarray documenation:

```
1. non-dimension coordinates are variables that [may] contain coordinate data,
   but are not a dimension coordinate.

2. They can be multidimensional ... and there is no relationship between
   the name of a non-dimension coordinate and the name(s) of its dimension(s).

3. Non-dimension coordinates can be useful for indexing or plotting; ...
```

Four of these variables contain spatial coordinate data:

- **XC** and **YC**, the longitude and latitudes of the 'c' points (varies with **tile**, **j** and **i**)
- **Z** the center depth of tracer cells (varies only with **k**)
- **PHrefC**, a reference pressure of tracer cells (varies only with **k**)

Two of these variables have *spatial dimensions* but are not coordinates in a traditional sense (and they will be removed in future ECCOv4 releases because their presence here makes no sense).

- **hFacC**, the fraction (0,1) of a tracer grid cell height which is wet (varies with **tile**, **k**, **j**, and **i**)
- **drF**, the distance between the top and bottom of a tracer grid cell (varies only with **k**)
- **rA**, model grid cell area (varies with **tile**, **j** and **i**)

Three non-dimension coordinates do not have any spatial dimensions!

- **iter**, the time step of the model (time iteration) when the record was saved
- **time**, a calendar date and time
- **time_bnds**, a 1x2 array of calendar dates and times indicating the start and end times of the averaging period of the field

When multiple DataArrays from different tiles and times are combined, the dimensions of the merged arrays will expand along the **time** and **tile** dimensions.

Let's quickly look at the **time_bnds** coordinate:

[12]:
```
theta_dataset.time_bnds
```

[12]:
```
<xarray.DataArray 'time_bnds' (time: 12, nv: 2)>
array([['2010-01-01T00:00:00.000000000', '2010-02-01T00:00:00.000000000'],
       ['2010-02-01T00:00:00.000000000', '2010-03-01T00:00:00.000000000'],
       ['2010-03-01T00:00:00.000000000', '2010-04-01T00:00:00.000000000'],
       ['2010-04-01T00:00:00.000000000', '2010-05-01T00:00:00.000000000'],
       ['2010-05-01T00:00:00.000000000', '2010-06-01T00:00:00.000000000'],
       ['2010-06-01T00:00:00.000000000', '2010-07-01T00:00:00.000000000'],
       ['2010-07-01T00:00:00.000000000', '2010-08-01T00:00:00.000000000'],
       ['2010-08-01T00:00:00.000000000', '2010-09-01T00:00:00.000000000'],
       ['2010-09-01T00:00:00.000000000', '2010-10-01T00:00:00.000000000'],
       ['2010-10-01T00:00:00.000000000', '2010-11-01T00:00:00.000000000'],
       ['2010-11-01T00:00:00.000000000', '2010-12-01T00:00:00.000000000'],
       ['2010-12-01T00:00:00.000000000', '2011-01-01T00:00:00.000000000']],
      dtype='datetime64[ns]')
Coordinates:
    time_bnds  (time, nv) datetime64[ns] 2010-01-01 2010-02-01 ... 2011-01-01
    iter       (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
  * time       (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
Dimensions without coordinates: nv
Attributes:
    long_name:      time bounds of averaging period
    standard_name:  time_bounds
```

For time-averaged fields, **time_bnds** is a 2D array provding the *start* and *end* time of each averaging period.

Let's look at the third record (March, 2010)

[13]:
```
theta_dataset.time_bnds[2]
```

[13]:
```
<xarray.DataArray 'time_bnds' (nv: 2)>
array(['2010-03-01T00:00:00.000000000', '2010-04-01T00:00:00.000000000'],
      dtype='datetime64[ns]')
Coordinates:
    time_bnds  (nv) datetime64[ns] 2010-03-01 2010-04-01
    iter       int32 159948
    time       datetime64[ns] 2010-03-16T12:00:00
Dimensions without coordinates: nv
Attributes:
    long_name:      time bounds of averaging period
    standard_name:  time_bounds
```

We see that the time bounds are 2010-03-01 to 2010-04-01, which make sense.

Having this time-bounds information readily available can be very helpful.

1.7.4 The Dimension Coordinates of the Arakawa C-Grid

Dimension coordinates have special meanings. The MITgcm uses the staggered Arakawa-C grid (hereafter c-grid). In c-grid models, variables are staggered in space. Horizontally, variables are associated with three 'locations':

1. tracer cells (e.g. temperature, salinity, density)
2. the 4 lateral faces of tracer cells (e.g., horizontal velocities and fluxes)
3. the 4 corners of tracer cells (e.g., vertical component of vorticity field)

Vertically, there are also two 'locations':

1. tracer cells
2. the 2 top/bottom faces of tracer cells (e.g., vertical velocities and fluxes).

To understand this better, let's review the geometry of c-grid models.

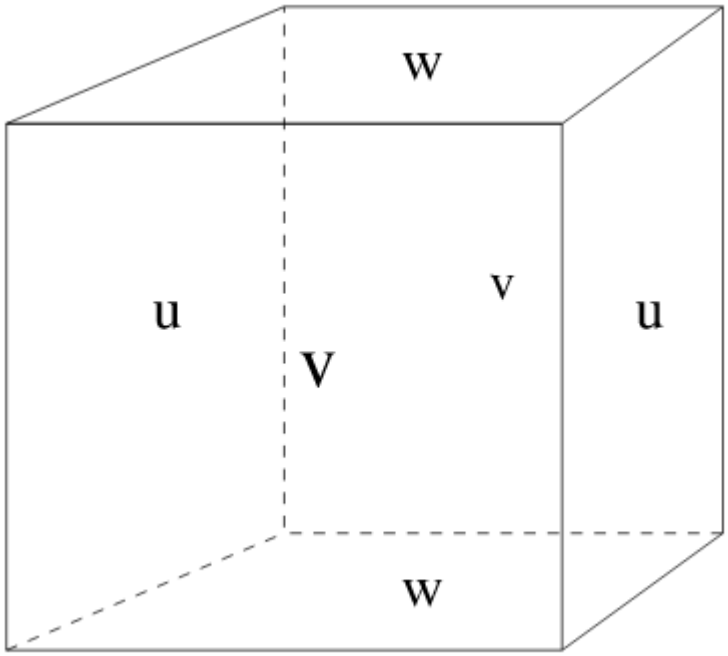
3D staggering of velocity components.

Paraphrasing from https://mitgcm.readthedocs.io/en/latest/algorithm/c-grid.html,

```
In c-grid models, the components of flow (u,v,w) are staggered in
space such that the zonal component falls on the interface between
tracer cells in the zonal direction. Similarly for the meridional
and vertical directions.
```

Why the c-grid?

```
The basic algorithm employed for stepping forward the momentum
equations is based on retaining non-divergence of the flow at
all times. This is most naturally done if the components of flow
are staggered in space in the form of an Arakawa C grid...
```

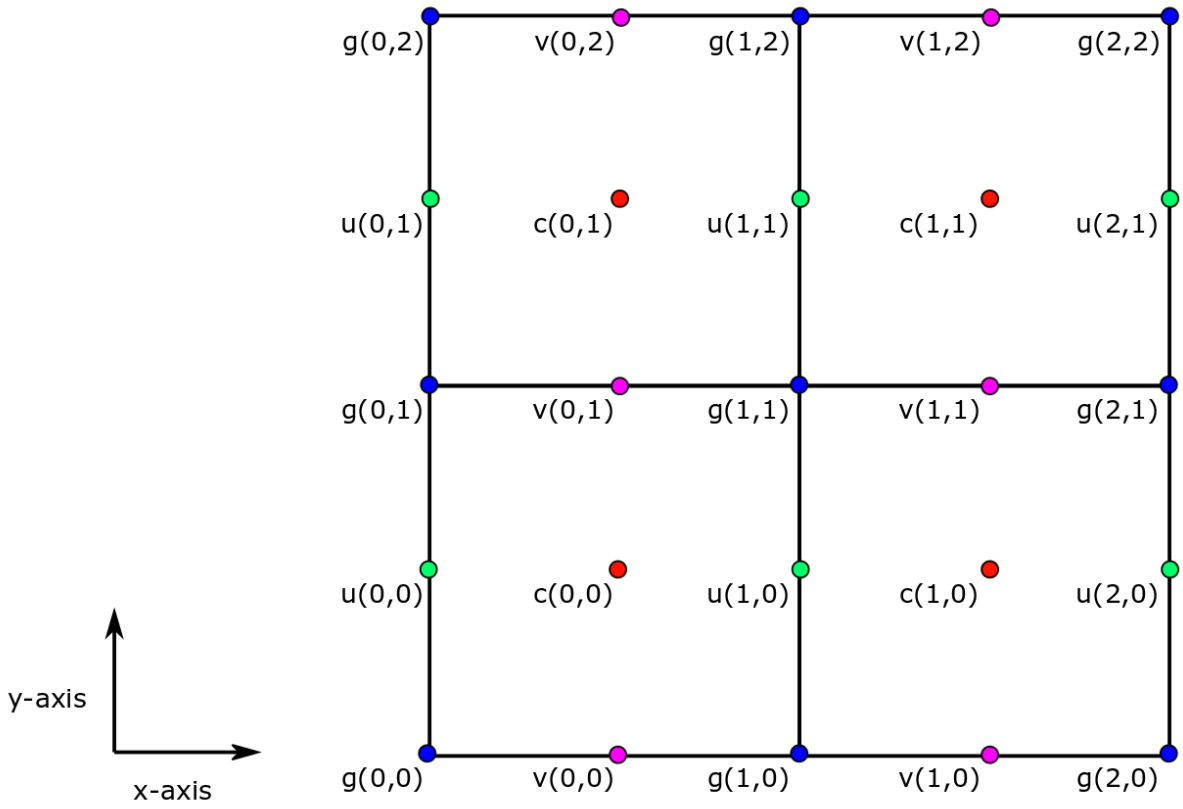


Defining the c-grid coordinate system

As shown, variables on Arakawa-C grids (c-grid) are staggered in space. For convenience we define a coordinate system that distinguishes between these different locations.

The c-grid horizontal coordinates

In the horizontal, variables can take one of four locations: **c**, **u**, **v**, or **g**.



horizontal "c" location

Variables associated with tracer cell area or volumetric averages (e.g., temperature, salinity, sea ice thickness, air-sea net heat flux) and variables associated with vertical velocities (e.g., $\vec{w}$) are identified with **c** locations.

For these variables we define the horizontal dimensions of **i** and **j**, corresponding with the model grid x and y directions, respectively.

horizontal "u" location

Variables associated with the two lateral sides of tracer cells in the x direction are identified with **u** locations.

Define the horizontal dimensions of **u** variables as **i_g** and **j**, corresponding with the model's x and y dimensions, respectively.

> **Important Note**: In the llc90 curvilinear model grid used by ECCOv4, the x direction is **NOT** same as the geographic zonal (east-west) direction.

horizontal "v" location

Variables associated with the two lateral sides of tracer cells in the y direction are identified with **v** locations.

Define the horizontal dimensions of **v** variables as **i** and **j_g**, corresponding with the model's x and y dimensions, respectively.

> **Important Note**: In the llc90 curvilinear model grid used by ECCOv4, the y direction is **NOT** same as the geographic meridional (north-south) direction.

horizontal "g" location

Variables associated with the horizontal corners of tracer grid cells are identified with **g** locations.

Define the horizontal dimensions of **g** variables as **i_g** and **j_g**, corresponding with the model's x and y dimensions, respectively.

The c-grid vertical coordinates

In the horizontal, variables can take one of two locations: **c** and **w**

vertical "c" location

Variables associated with tracer cell volumetric quantities (e.g., temperature, salinity) are identified with **c** locations.

For these variables we define the vertical dimensions of **k** which corresponds with the model grid' z direction.

vertical "w" location

Variables associated with the two top/bottom sides of tracer cells in the z direction are identified with **w** locations.

For these variables we define the vertical dimension of **:math:`k_u`** to indicate the model tracer cell's **u**pper faces in the k direction, respectively.

Two other vertical dimensions are also used. **:math:`k_l`** indicates the model tracer cell's **l**ower faces, and k_{p1} which index all of the **upper** and **lower** faces.

> Note: In ECCOv4 NetCDF files both **:math:`k_u`**(0) and **:math:`k_{p1}`**(0) correspond to the same top face of the model tracer grid cell.

1.7.5 All ECCOv4 coordinates

Now that we have been oriented to the dimensions and coordinates used by ECCOv4, let's load up and examinea a `Dataset` that uses all of them, an ECCOv4 NetCDF grid file.

Open the ECCOv4 grid file associated with tile 2:

[14]:
```
grid_dir = ECCO_dir + '/nctiles_grid/'
print(grid_dir)
## load the grid
grid_dataset = xr.open_dataset(grid_dir + 'ECCOv4r3_grid.nc')
```

(continues on next page)

(continued from previous page)

```
# show contents of grid_dataset
grid_dataset
```

```
/home/ifenty/ECCOv4-release/Release3_alt/nctiles_grid/
```

[14]:
```
<xarray.Dataset>
Dimensions:    (i: 90, i_g: 90, j: 90, j_g: 90, k: 50, k_l: 50, k_p1: 51, k_u: 50, tile: 13)
Coordinates:
  * i          (i) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * i_g        (i_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j          (j) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j_g        (j_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k          (k) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_u        (k_u) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_l        (k_l) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_p1       (k_p1) int64 0 1 2 3 4 5 6 7 8 9 ... 42 43 44 45 46 47 48 49 50
  * tile       (tile) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
    XC         (tile, j, i) float32 ...
    YC         (tile, j, i) float32 ...
    XG         (tile, j_g, i_g) float32 ...
    YG         (tile, j_g, i_g) float32 ...
    CS         (tile, j, i) float32 ...
    SN         (tile, j, i) float32 ...
    Z          (k) float32 ...
    Zp1        (k_p1) float32 ...
    Zu         (k_u) float32 ...
    Zl         (k_l) float32 ...
    rA         (tile, j, i) float32 ...
    dxG        (tile, j_g, i) float32 ...
    dyG        (tile, j, i_g) float32 ...
    Depth      (tile, j, i) float32 ...
    rAz        (tile, j_g, i_g) float32 ...
    dxC        (tile, j, i_g) float32 ...
    dyC        (tile, j_g, i) float32 ...
    rAw        (tile, j, i_g) float32 ...
    rAs        (tile, j_g, i) float32 ...
    drC        (k_p1) float32 ...
    drF        (k) float32 ...
    PHrefC     (k) float32 ...
    PHrefF     (k_p1) float32 ...
    hFacC      (k, tile, j, i) float32 ...
    hFacW      (k, tile, j, i_g) float32 ...
    hFacS      (k, tile, j_g, i) float32 ...
    maskC      (k, tile, j, i) bool ...
    maskW      (k, tile, j, i_g) bool ...
    maskS      (k, tile, j_g, i) bool ...
    maskCtrlW  (k, tile, j, i_g) bool ...
    maskCtrlS  (k, tile, j_g, i) bool ...
    maskCtrlC  (k, tile, j, i) bool ...
Data variables:
    *empty*
Attributes:
```

(continues on next page)

(continued from previous page)

```
    Conventions:  CF-1.6
    title:        netCDF wrapper of MITgcm MDS binary data
    source:       MITgcm
    history:      Created by calling `open_mdsdataset(extra_metadata=None, ll...
```

Dimensions

```
Dimensions: (i: 90, i_g: 90, j: 90, j_g: 90, k: 50, k_l: 50, k_p1: 51, k_u: 50, tile: 13)
```

The *Dimensions* list now lists the six different coordinates and their dimension used by variables stored in this new grid tile `Dataset` object.

Dimension Coordinates

```
* k_p1     (k_p1) int64 0 1 2 3 4 5 6 7 8 9 ... 41 42 43 44 45 46 47 48 49 50
* j_g      (j_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
* i_g      (i_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
* k        (k) int64 0 1 2 3 4 5 6 7 8 9 10 ... 40 41 42 43 44 45 46 47 48 49
* j        (j) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
* k_u      (k_u) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
* i        (i) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
* k_l      (k_l) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
* tile     (tile) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
```

We note that horizontal coordindates have 90 indices [0..89], *ku* and *kl* have 50 [0..49] and *kp1* has 51 [0..50]

Non-Dimension Spatial Coordinates

Some non-dimension coordinates are spatial coordinates. * XC, YC : longitude and latitude of tracer cell centers * XG, YG : longitude and latitude of tracer cell corners * Z_l, Z_u, Z_p1 : depths of tracer cell lower and upper faces * Z : depths of tracer cell centers * PHrefC, PHrefF : reference pressures at tracer cell centers and upper and lower faces

```
XC       (tile, j, i) float32 -111.60647 -111.303 ... -105.58465 -111.86579
YC       (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
XG       (tile, j_g, i_g) float32 -115.0 -115.0 ... -102.928925 -108.95171
YG       (tile, j_g, i_g) float32 -88.17569 -88.31587 ... -87.9892 -88.02409
Zl       (k_l) float32 0.0 -10.0 -20.0 -30.0 ... -4834.0 -5244.5 -5678.0
Zu       (k_u) float32 -10.0 -20.0 -30.0 -40.0 ... -5244.5 -5678.0 -6134.5
Z        (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
Zp1      (k_p1) float32 0.0 -10.0 -20.0 -30.0 ... -5244.5 -5678.0 -6134.5
PHrefC   (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
PHrefF   (k_p1) float32 0.0 98.1 196.2 ... 51448.547 55701.18 60179.445
```

While others describe the model grid geometry (areas, distances, distances, and geometric factors) and other information (model depth). They are not coordinates in any meaningful sense but they *are essential for quantitative analysis*.

horizontal distances:

```
dxG      (j_g, i) float32 ...
dyG      (j, i_g) float32 ...
```

(continues on next page)

(continued from previous page)

```
dxC      (j, i_g) float32 ...
dyC      (j_g, i) float32 ...
```

vertical distances:

```
drC      (k_p1) float32 ...
drF      (k) float32 ...
```

areas:

```
rAs      (j_g, i) float32 ...
rAw      (j, i_g) float32 ...
rA       (j, i) float32 ...
rAz      (j_g, i_g) float32 ...
```

geometric factors

```
hFacS    (k, j_g, i) float32 ...
hFacC    (k, j, i) float32 ...
hFacW    (k, j, i_g) float32 ...
```

seafloor depth

```
Depth    (j, i) float32 ...
```

These ancilliary fields are classified as non-dimension coordinates because we wanted to reserve *data variables* for variables that are part of the model solution.

In the end, it doesn't matter what you call these ancillary variables – what matters is that you have them on hand for calculations. If it makes you feel beter, use the function `.reset_coords()` to convert all non-dimension coordinates to data variables.
~~~ grid_dataset.reset_coords() ~~~

Non-Dimension Model Geometry Coordinates

Non-Dimension Model Geometry "Coordinates" are not coordinates in the sense that they help you orient in space or time, but they provide measures of the model grid such as distances and areas. Let's examine one of these grid geometric variables, *dxG*:

[15]:
```
grid_dataset.dxG
```

[15]:
```
<xarray.DataArray 'dxG' (tile: 13, j_g: 90, i: 90)>
[105300 values with dtype=float32]
Coordinates:
  * i        (i) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * j_g      (j_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * tile     (tile) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
    dxG      (tile, j_g, i) float32 ...
    dyC      (tile, j_g, i) float32 ...
    rAs      (tile, j_g, i) float32 ...
Attributes:
```

(continues on next page)
~~~

(continued from previous page)

```
    standard_name:  cell_x_size_at_v_location
    long_name:      cell x size
    units:          m
    coordinate:     YG XC
```

dxG has coordinates **tile**, **j_g** and **i** which means that it is a **v** location variable. dxG is the horizontal distance between g points (tracer cell corners) in the tile's x direction.

For reference, see the chart below from the MITgcm documentation, Figure 2.6

dxG = Δx_g in subfigure (a) below:

Figure 2.6 Staggering of horizontal grid descriptors (lengths and areas). The grid lines indicate the tracer cell boundaries and are the reference grid for all panels. a) The area of a tracer cell, , is bordered by the lengths and . b) The area of a vorticity cell, , is bordered by the lengths and . c) The area of a u cell, , is bordered by the lengths and . d) The area of a v cell, , is bordered by the lengths and .

1.7.6 Dimensions and Coordinates of *UVEL*

So far we looked *THETA* which is a c variable. Let's examine *UVEL*, horizontal velocity in the tile's x direction. As you've probably guessed, *UVEL* is a u variable:

Load tile 2 of the 2010 March average horizonatal velocity in the x direction.

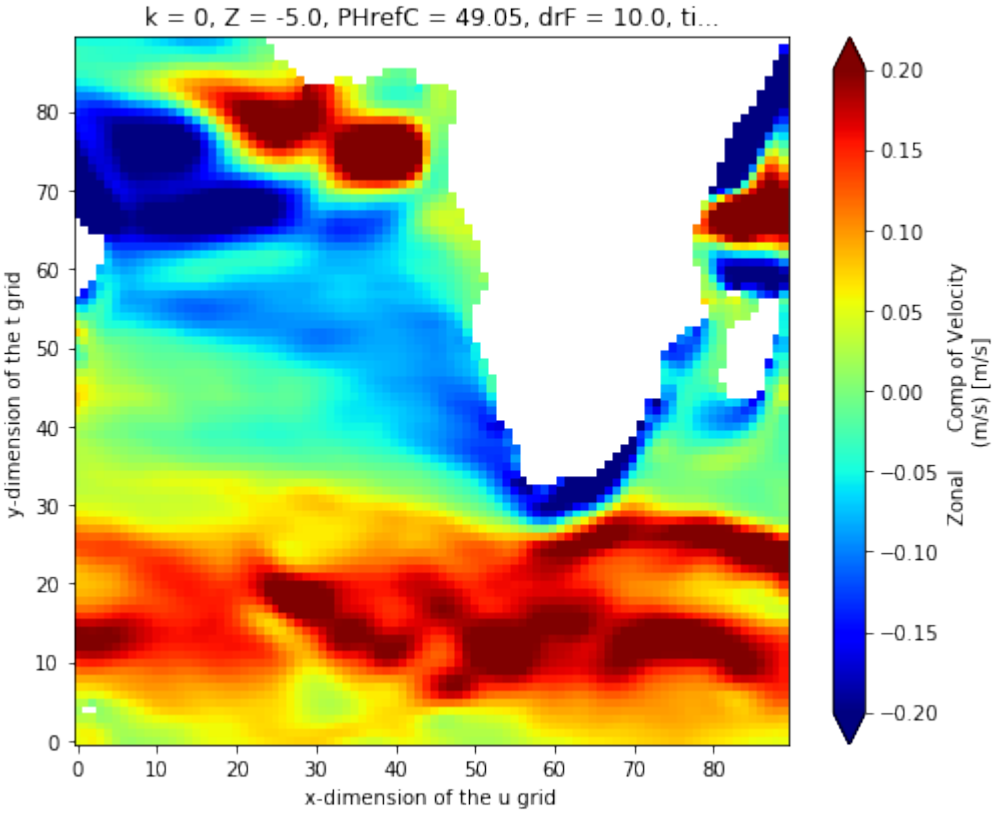
[16]:
```
# Directory of the UVEL files
data_dir= ECCO_dir + '/nctiles_monthly//UVEL/'

fname = 'UVEL_2010.nc'
uvel_dataset = xr.open_dataset(data_dir + fname).load()
uvel_dataset.attrs = []
```

UVEL context

Plot tile 1 time-mean horizontal velocity at the top-most model grid cell in the tile's x direction

[17]:
```
fig=plt.figure(figsize=(8, 6.5))
ud_masked = uvel_dataset.UVEL.where(uvel_dataset.hFacW > 0, np.nan)
ud_masked.isel(k=0,tile=1, time=0).plot(cmap='jet', vmin=-.2,vmax=.2)
```

[17]:
```
<matplotlib.collections.QuadMesh at 0x7f85133c1f28>
```



Let's look at the dimensions and coordinates of *UVEL*

[18]:
```
uvel_dataset.data_vars
```

[18]:
```
Data variables:
    UVEL     (time, tile, k, j, i_g) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
```

[19]:
```
uvel_dataset.coords
```

[19]:
```
Coordinates:
  * i_g         (i_g) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j           (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k           (k) int32 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
    Z           (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    PHrefC      (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
    drF         (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    dxC         (tile, j, i_g) float32 15583.418 15588.104 ... 23406.256
    rAw         (tile, j, i_g) float32 361699460.0 362790240.0 ... 364760350.0
    dyG         (tile, j, i_g) float32 23210.262 23273.26 ... 15595.26 15583.685
    hFacW       (tile, k, j, i_g) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
  * tile        (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
    time_bnds   (time, nv) datetime64[ns] 2010-01-01 2010-02-01 ... 2011-01-01
    iter        (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
  * time        (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
```

Dimension coordinates

As expected, UVEL has horizontal dimensions of **i_g, j**. Because UVEL is the velocity at the left and right faces of 3D tracer cells.

```
* i_g        (i_g) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
* j          (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
* k          (k) int32 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
* tile       (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
* time       (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
```

1.7.7 Dimensions and Coordinates of *VVEL*

Finally, let's examine *VVEL*, horizontal velocity in the tile's y direction. As you've probably guessed, *VVEL* is a v variable:

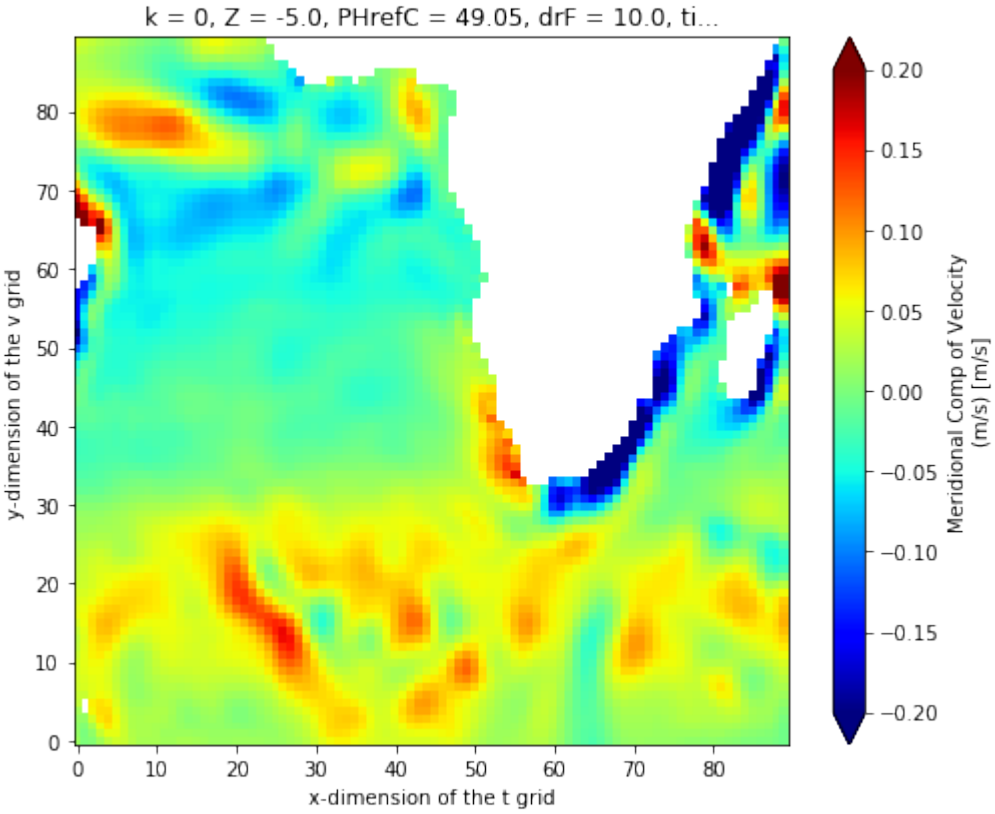
[20]:
```
# Directory of the VVEL files
data_dir= ECCO_dir + '/nctiles_monthly//VVEL/'

fname = 'VVEL_2010.nc'
vvel_dataset = xr.open_dataset(data_dir + fname).load()
vvel_dataset.attrs = []
```

VVEL context

Plot the time-mean horizontal velocity in the tile's y direction at model level 5 (55m)

[21]:
```
fig=plt.figure(figsize=(8, 6.5))
vd_masked = vvel_dataset.VVEL.where(vvel_dataset.hFacS > 0, np.nan)
vd_masked.isel(k=0,tile=1, time=0).plot(cmap='jet', vmin=-.2,vmax=.2)
```

[21]:
```
<matplotlib.collections.QuadMesh at 0x7f85133156a0>
```



[22]:
```
vvel_dataset.data_vars
```

[22]:
```
Data variables:
    VVEL     (time, tile, k, j_g, i) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
```

[23]:
```
vvel_dataset.coords
```

[23]:
```
Coordinates:
  * j_g         (j_g) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * i           (i) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k           (k) int32 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
    Z           (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    PHrefC      (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
    drF         (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    rAs         (tile, j_g, i) float32 179944260.0 180486990.0 ... 364150620.0
    dxG         (tile, j_g, i) float32 15584.907 15589.316 ... 23142.107
    dyC         (tile, j_g, i) float32 11563.718 11593.785 ... 15578.138
    hFacS       (tile, k, j_g, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
  * tile        (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
    time_bnds   (time, nv) datetime64[ns] 2010-01-01 2010-02-01 ... 2011-01-01
    iter        (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
  * time        (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
```

Dimension coordinates

As expected, *VVEL* has horizontal dimensions of **i, j_g**. Because *VVEL* is the velocity at the front and rear faces of 3D tracer cells we use **k** for its vertical coordinate.

```
* i          (i) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
* j_g        (j_g) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
* k          (k) int32 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
```

1.7.8 Summary

ECCOv4 variables are on the staggered Arakawa-C grid. Different dimension labels and coordinates are applied to state estimate variables so that one can easily identify where on the c-grid any particular variable is situated.

1.8 Loading the ECCOv4 native model grid parameters

1.8.1 Objectives

Introduce two methods of loading the ECCOv4 native model grid parameters.

1.8.2 Introduction

The ECCOv4 model grid parameters are provided as a single NetCDF file. The file you have may look a little different than the one shown here because we have been working hard at improving how what exactly goes into our NetCDF files.

This tutorial document is current as of Sep 2019 with the ECCOv4 NetCDF grid files provided in the following directories:

https://ecco.jpl.nasa.gov/drive/files/Version4/Release3_alt (po.daac drive, recommended)

https://web.corral.tacc.utexas.edu/OceanProjects/ECCO/ECCOv4/Release3_alt/ (mirror at U. Texas, Austin)

1.8.3 Two methods to load the ECCOv4 model grid parameter NetCDF file

Because the ECCOv4 model grid parameter data is provided in a single file you can use the `open_dataset` routine from `xarray` to open it.

Alternatively, our subroutine `load_ecco_grid_nc` allows you to (optionally) specify a subset of vertical levels or a subset of tiles to load.

We'll show both methods. Let's start with `open_dataset`.

First set up your environment.

[1]:
```
import numpy as np
import xarray as xr
import sys
import matplotlib.pyplot as plt
%matplotlib inline
```

[2]:
```
## Import the ecco_v4_py library into Python
## =========================================

## -- If ecco_v4_py is not installed in your local Python library,
##    tell Python where to find it.  For example, if your ecco_v4_py
##    files are in /Users/ifenty/ECCOv4-py/ecco_v4_py, then use:

sys.path.append('/home/ifenty/ECCOv4-py')
import ecco_v4_py as ecco
```

[3]:
```
## Set top-level file directory for the ECCO NetCDF files
## ======================================================

## -- If files are on a local machine, use something like
# base_dir = '/Users/ifenty/'
base_dir = '/home/ifenty/ECCOv4-release/'

## define a high-level directory for ECCO fields
ECCO_dir = base_dir + '/Release3_alt/'
```

Method 1: Loading the model grid parameters using `load_ecco_grid_nc`

Method 2 is super simple, just use `open_dataset`:

[4]:
```
grid_dir = ECCO_dir + 'nctiles_grid/'

## load the grid
grid = xr.open_dataset(grid_dir + 'ECCOv4r3_grid.nc')
grid
```

[4]:
```
<xarray.Dataset>
Dimensions:    (i: 90, i_g: 90, j: 90, j_g: 90, k: 50, k_l: 50, k_p1: 51, k_u: 50, tile:␣
↪13)
Coordinates:
  * i          (i) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * i_g        (i_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j          (j) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j_g        (j_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k          (k) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_u        (k_u) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_l        (k_l) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_p1       (k_p1) int64 0 1 2 3 4 5 6 7 8 9 ... 42 43 44 45 46 47 48 49 50
  * tile       (tile) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
    XC         (tile, j, i) float32 ...
    YC         (tile, j, i) float32 ...
    XG         (tile, j_g, i_g) float32 ...
    YG         (tile, j_g, i_g) float32 ...
    CS         (tile, j, i) float32 ...
    SN         (tile, j, i) float32 ...
    Z          (k) float32 ...
    Zp1        (k_p1) float32 ...
    Zu         (k_u) float32 ...
```

(continues on next page)

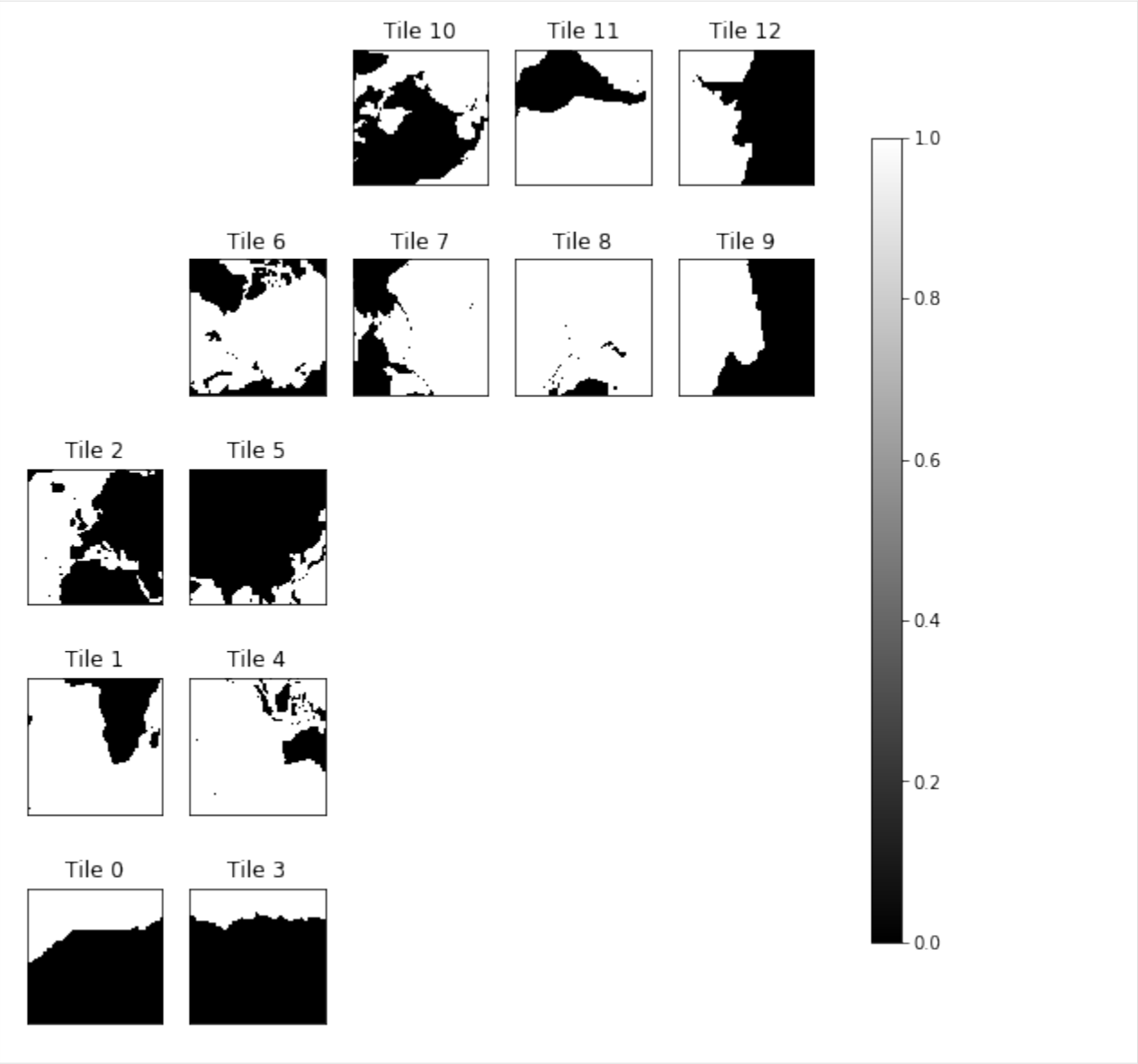
(continued from previous page)

```
    Zl          (k_l) float32 ...
    rA          (tile, j, i) float32 ...
    dxG         (tile, j_g, i) float32 ...
    dyG         (tile, j, i_g) float32 ...
    Depth       (tile, j, i) float32 ...
    rAz         (tile, j_g, i_g) float32 ...
    dxC         (tile, j, i_g) float32 ...
    dyC         (tile, j_g, i) float32 ...
    rAw         (tile, j, i_g) float32 ...
    rAs         (tile, j_g, i) float32 ...
    drC         (k_p1) float32 ...
    drF         (k) float32 ...
    PHrefC      (k) float32 ...
    PHrefF      (k_p1) float32 ...
    hFacC       (k, tile, j, i) float32 ...
    hFacW       (k, tile, j, i_g) float32 ...
    hFacS       (k, tile, j_g, i) float32 ...
    maskC       (k, tile, j, i) bool ...
    maskW       (k, tile, j, i_g) bool ...
    maskS       (k, tile, j_g, i) bool ...
    maskCtrlW   (k, tile, j, i_g) bool ...
    maskCtrlS   (k, tile, j_g, i) bool ...
    maskCtrlC   (k, tile, j, i) bool ...
Data variables:
    *empty*
Attributes:
    Conventions:  CF-1.6
    title:        netCDF wrapper of MITgcm MDS binary data
    source:       MITgcm
    history:      Created by calling `open_mdsdataset(extra_metadata=None, ll...
```

Let's plot two of the model grid parameter fields `hFacC` (tracer cell thickness factor) and `rA` (grid cell surface area)

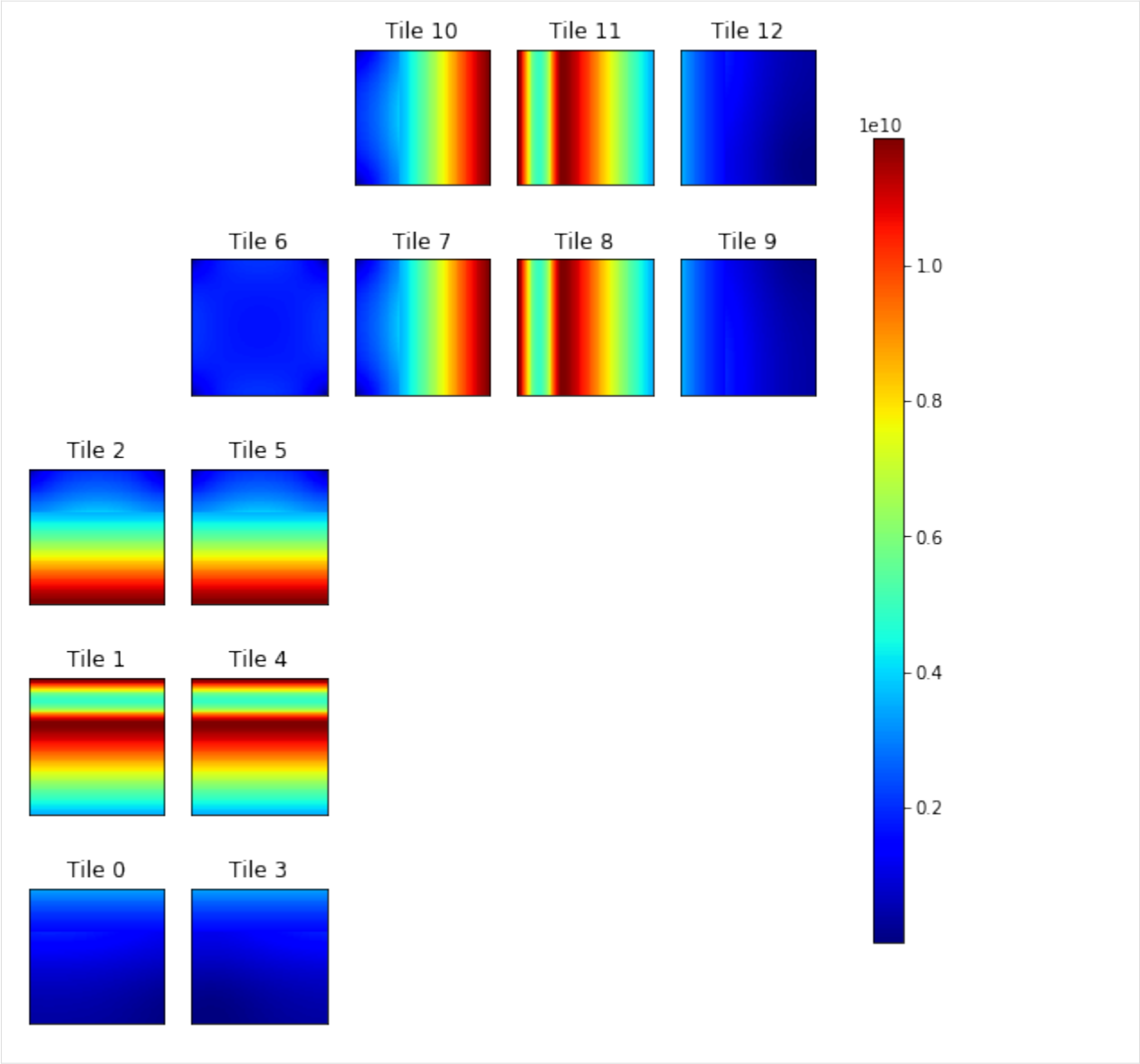
First we plot `hFac`:

[5]:
```
ecco.plot_tiles(grid.hFacC.sel(k=0), show_colorbar=True, cmap='gray');
```



[6]:
```
ecco.plot_tiles(grid.rA, show_colorbar=True);
'Model grid cell surface area [m^2]'
```

[6]:
```
'Model grid cell surface area [m^2]'
```



Method 2: Loading the model grid parameters using `load_ecco_grid_nc`

A more advanced routine, `load_ecco_grid_nc`, allows you to load only a subset of tiles and vertical levels. If no optional parameters are given, the entire grid object is loaded, just like `open_dataset`

[7]:
```
grid_dir = ECCO_dir + 'nctiles_grid'

grid = ecco.load_ecco_grid_nc(grid_dir, 'ECCOv4r3_grid.nc')
grid
```

[7]:
```
<xarray.Dataset>
Dimensions:    (i: 90, i_g: 90, j: 90, j_g: 90, k: 50, k_l: 50, k_p1: 51, k_u: 50, tile:␣
↪13)
Coordinates:
  * i          (i) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
```

(continues on next page)

(continued from previous page)

```
  * i_g        (i_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j          (j) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j_g        (j_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k          (k) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_u        (k_u) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_l        (k_l) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_p1       (k_p1) int64 0 1 2 3 4 5 6 7 8 9 ... 42 43 44 45 46 47 48 49 50
  * tile       (tile) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
    XC         (tile, j, i) float32 -111.60647 -111.303 ... -111.86579
    YC         (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
    XG         (tile, j_g, i_g) float32 -115.0 -115.0 ... -102.928925 -108.95171
    YG         (tile, j_g, i_g) float32 -88.17569 -88.31587 ... -88.02409
    CS         (tile, j, i) float32 0.06157813 0.06675376 ... -0.9983638
    SN         (tile, j, i) float32 -0.99810225 -0.9977695 ... -0.057182025
    Z          (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    Zp1        (k_p1) float32 0.0 -10.0 -20.0 -30.0 ... -5244.5 -5678.0 -6134.5
    Zu         (k_u) float32 -10.0 -20.0 -30.0 -40.0 ... -5244.5 -5678.0 -6134.5
    Zl         (k_l) float32 0.0 -10.0 -20.0 -30.0 ... -4834.0 -5244.5 -5678.0
    rA         (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
    dxG        (tile, j_g, i) float32 15584.907 15589.316 ... 23142.107
    dyG        (tile, j, i_g) float32 23210.262 23273.26 ... 15595.26 15583.685
    Depth      (tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    rAz        (tile, j_g, i_g) float32 179944260.0 180486990.0 ... 364150620.0
    dxC        (tile, j, i_g) float32 15583.418 15588.104 ... 23406.256
    dyC        (tile, j_g, i) float32 11563.718 11593.785 ... 15578.138
    rAw        (tile, j, i_g) float32 361699460.0 362790240.0 ... 364760350.0
    rAs        (tile, j_g, i) float32 179944260.0 180486990.0 ... 364150620.0
    drC        (k_p1) float32 5.0 10.0 10.0 10.0 ... 399.0 422.0 445.0 228.25
    drF        (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    PHrefC     (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
    PHrefF     (k_p1) float32 0.0 98.1 196.2 ... 51448.547 55701.18 60179.445
    hFacC      (k, tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    hFacW      (k, tile, j, i_g) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    hFacS      (k, tile, j_g, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    maskC      (k, tile, j, i) bool False False False ... False False False
    maskW      (k, tile, j, i_g) bool False False False ... False False False
    maskS      (k, tile, j_g, i) bool False False False ... False False False
    maskCtrlW  (k, tile, j, i_g) bool False False False ... False False False
    maskCtrlS  (k, tile, j_g, i) bool False False False ... False False False
    maskCtrlC  (k, tile, j, i) bool False False False ... False False False
Data variables:
    *empty*
Attributes:
    Conventions:              CF-1.6
    geospatial_lat_max:       90.0
    geospatial_lat_min:       -90.0
    geospatial_lon_max:       180.0
    geospatial_lon_min:       -179.99919
    geospatial_vertical_max:  -5.0
    geospatial_vertical_min:  -5906.25
    history:                  Created by calling `open_mdsdataset(extra_metad...
    nx:                       90
```

(continues on next page)

(continued from previous page)

```
    ny:                     90
    nz:                     50
    source:                 MITgcm
    title:                  netCDF wrapper of MITgcm MDS binary data
```

Alternatively we can load just a subset of tiles and vertical levels.

[8]:
```
grid_subset = ecco.load_ecco_grid_nc(grid_dir, 'ECCOv4r3_grid.nc', tiles_to_load = [1,␣
↪10, 12], k_subset=[0,1,2,3])
grid_subset
```

[8]:
```
<xarray.Dataset>
Dimensions:    (i: 90, i_g: 90, j: 90, j_g: 90, k: 4, k_l: 4, k_p1: 4, k_u: 4, tile: 3)
Coordinates:
  * i          (i) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * i_g        (i_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j          (j) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j_g        (j_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k          (k) int64 0 1 2 3
  * k_u        (k_u) int64 0 1 2 3
  * k_l        (k_l) int64 0 1 2 3
  * k_p1       (k_p1) int64 0 1 2 3
  * tile       (tile) int64 1 10 12
    XC         (tile, j, i) float32 -37.5 -36.5 -35.5 ... -105.58465 -111.86579
    YC         (tile, j, i) float32 -56.73891 -56.73891 ... -88.07871 -88.10267
    XG         (tile, j_g, i_g) float32 -38.0 -37.0 ... -102.928925 -108.95171
    YG         (tile, j_g, i_g) float32 -57.005695 -57.005695 ... -88.02409
    CS         (tile, j, i) float32 1.0 1.0 1.0 ... -0.98536175 -0.9983638
    SN         (tile, j, i) float32 -0.0 6.52406e-15 ... -0.1704766 -0.057182025
    Z          (k) float32 -5.0 -15.0 -25.0 -35.0
    Zp1        (k_p1) float32 0.0 -10.0 -20.0 -30.0
    Zu         (k_u) float32 -10.0 -20.0 -30.0 -40.0
    Zl         (k_l) float32 0.0 -10.0 -20.0 -30.0
    rA         (tile, j, i) float32 3624512000.0 3624512000.0 ... 361119100.0
    dxG        (tile, j_g, i) float32 60542.324 60542.324 ... 23142.107
    dyG        (tile, j, i_g) float32 59441.125 59441.125 ... 15595.26 15583.685
    Depth      (tile, j, i) float32 3284.1084 3485.7 3485.7 ... 0.0 0.0 0.0
    rAz        (tile, j_g, i_g) float32 3584245000.0 ... 364150620.0
    dxC        (tile, j, i_g) float32 60975.85 60975.85 ... 23865.428 23406.256
    dyC        (tile, j_g, i) float32 59201.66 59201.66 ... 15585.765 15578.138
    rAw        (tile, j, i_g) float32 3624512000.0 3624512000.0 ... 364760350.0
    rAs        (tile, j_g, i) float32 3584245000.0 3584245000.0 ... 364150620.0
    drC        (k_p1) float32 5.0 10.0 10.0 10.0
    drF        (k) float32 10.0 10.0 10.0 10.0
    PHrefC     (k) float32 49.05 147.15 245.25 343.35
    PHrefF     (k_p1) float32 0.0 98.1 196.2 294.3
    hFacC      (k, tile, j, i) float32 1.0 1.0 1.0 1.0 1.0 ... 0.0 0.0 0.0 0.0
    hFacW      (k, tile, j, i_g) float32 1.0 1.0 1.0 1.0 1.0 ... 0.0 0.0 0.0 0.0
    hFacS      (k, tile, j_g, i) float32 1.0 1.0 1.0 1.0 1.0 ... 0.0 0.0 0.0 0.0
    maskC      (k, tile, j, i) bool True True True True ... False False False
    maskW      (k, tile, j, i_g) bool True True True True ... False False False
    maskS      (k, tile, j_g, i) bool True True True True ... False False False
    maskCtrlW  (k, tile, j, i_g) bool True True True True ... False False False
```

(continues on next page)

(continued from previous page)

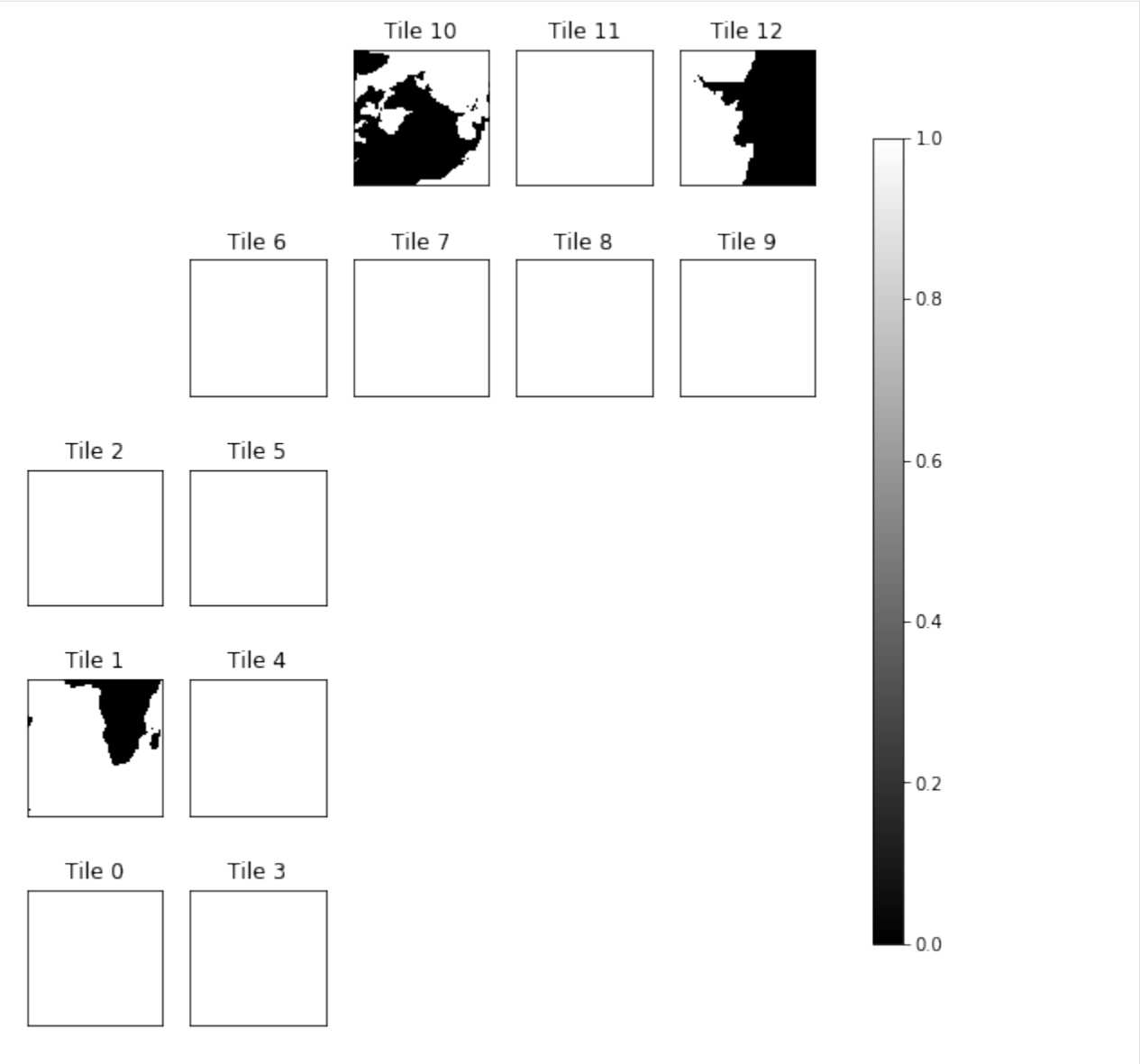
```
    maskCtrlS  (k, tile, j_g, i) bool True True True True ... False False False
    maskCtrlC  (k, tile, j, i) bool True True True True ... False False False
Data variables:
    *empty*
Attributes:
    Conventions:              CF-1.6
    geospatial_lat_max:       72.043434
    geospatial_lat_min:       -88.02409
    geospatial_lon_max:       51.0
    geospatial_lon_min:       -128.0
    geospatial_vertical_max:  -5.0
    geospatial_vertical_min:  -35.0
    history:                  Created by calling `open_mdsdataset(extra_metad...
    nx:                       90
    ny:                       90
    nz:                       4
    source:                   MITgcm
    title:                    netCDF wrapper of MITgcm MDS binary data
```

notice that `grid_subset` only has 3 tiles (9,10, 11) and 4 depth levels (0, 1, 2, 3), as expected.

Let's plot `hFacC` and `rA` again

[9]:
```
ecco.plot_tiles(grid_subset.hFacC.sel(k=0), show_colorbar=True, cmap='gray');
'Model grid cell surface area [m^2] in tiles 1, 10, and 12 '
```

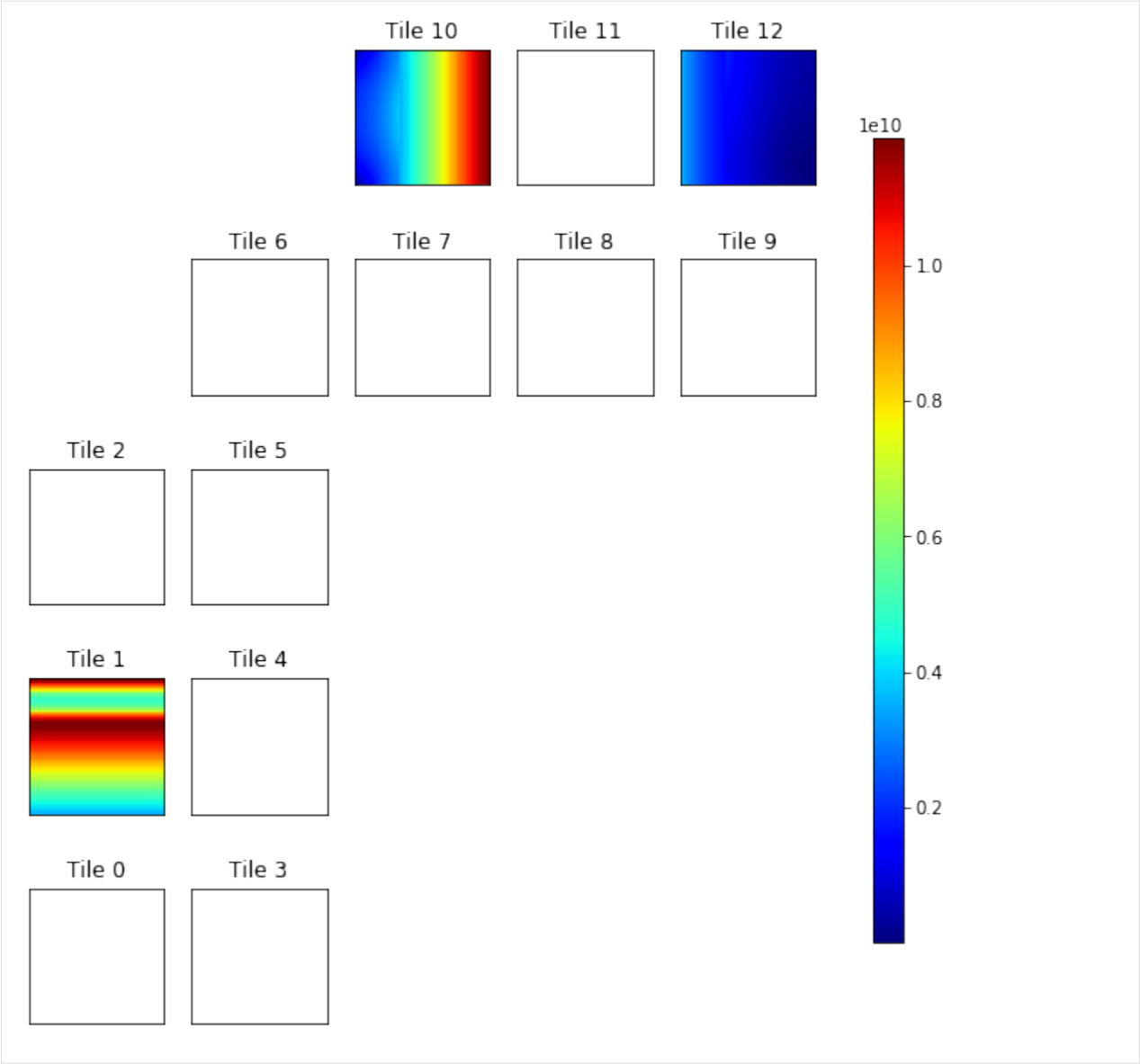
[9]:
```
'Model grid cell surface area [m^2] in tiles 1, 10, and 12 '
```



Notice that 10 of the 13 tiles are blank because they were not loaded.

[10]:
```
ecco.plot_tiles(grid_subset.rA, show_colorbar=True);
'Model grid cell surface area [m^2]'
```

[10]:
```
'Model grid cell surface area [m^2]'
```



1.8.4 Summary

Now you know two ways to load the ECCOv4 grid parameter file!

1.9 Loading the ECCOv4 state estimate fields on the native model grid

1.9.1 Objectives

Introduce several methods for loading the ECCOv4 state estimate files on the native model grid.

1.9.2 Introduction

ECCOv4 native-grid state estimate fields are packaged together as NetCDF files. We have been improving how these files are created, what fields they contain, and how they are distributed in directories. As of this writing, Sep 2019, the latest version of these files can be found here:

https://ecco.jpl.nasa.gov/drive/files/Version4/Release3_alt (po.daac drive, recommended)

https://web.corral.tacc.utexas.edu/OceanProjects/ECCO/ECCOv4/Release3_alt/ (mirror at U. Texas, Austin)

This tutorial document is current with the files in the above directories as of September 2019.

1.9.3 NetCDF File Format

ECCOv4 state estimate fields are provided as NetCDF files with one variable and one year per file. State estimate fields are provided as **monthly means**, **daily means** and **monthly snapshots**. The file directories for NetCDF files should look something like (although exact naming may vary):

- /nctiles_monthly/**VARIABLE_NAME/VARIABLE_NAME_YYYY**.nc
- /nctiles_monthly_snapshots/**VARIABLE_NAME/VARIABLE_NAME_YYYY**.nc
- /nctiles_daily/**VARIABLE_NAME/VARIABLE_NAME_YYYY**.nc

While the model grid files are provided here:

- /nctiles_grid/ECCOv4r3_grid.nc

Typical file sizes are: ~~~ 3D monthly-mean and monthly-snapshot fields: 265mb (50 levels x 12 months x 25 years x 13 tiles) 2D monthly-mean fields : 6mb (1 level x 12 months x 25 years x 13 tiles) 2D daily-mean fields : 150mb (1 level x 365 days x 12 years x 13 tiles) ~~~

The advantages of aggregating one year of data into a single NetCDF file is that the I/O time per data element. One nice feature of using the `xarray` and `Dask` libraries is that we do not have to load the entire file contents into RAM to work with them.

1.9.4 Two methods to load one ECCOv4 NetCDF file

In ECCO NetCDF files, all 13 tiles for a given year are aggregated into a single file. Therefore, we can use the `open_dataset` routine from `xarray` to open a single NetCDF variable file.

Alternatively, the subroutine `load_ecco_var_from_years_nc` allows you to optionally specify a subset of vertical levels or tiles to load using optional arguments.

We'll show both methods. First `open_dataset` then `load_ecco_var_from_years_nc`

First set up the environment, load model grid parameters.

[1]:
```
import numpy as np
import xarray as xr
import sys
import matplotlib.pyplot as plt
%matplotlib inline
```

[2]:
```
## Import the ecco_v4_py library into Python
## =========================================

## -- If ecco_v4_py is not installed in your local Python library,
##    tell Python where to find it.  For example, if your ecco_v4_py
##    files are in /Users/ifenty/ECCOv4-py/ecco_v4_py, then use:
sys.path.append('/home/ifenty/ECCOv4-py')

import ecco_v4_py as ecco
```

[3]:
```
## Set top-level file directory for the ECCO NetCDF files
## ======================================================
# base_dir = '/home/username/'
base_dir = '/home/ifenty/ECCOv4-release/'

## define a high-level directory for ECCO fields
ECCO_dir = base_dir + '/Release3_alt/'

## define the directory with the model grid
grid_dir = ECCO_dir + 'nctiles_grid/'

## load the grid
grid = ecco.load_ecco_grid_nc(grid_dir, 'ECCOv4r3_grid.nc')
```

Loading a single ECCOv4 variable NetCDF file using `open_dataset`

[4]:
```
SSH_dir = ECCO_dir + '/nctiles_monthly/SSH/'
SSH_dataset = xr.open_dataset(SSH_dir + '/SSH_2010.nc')
SSH_dataset.SSH
```

[4]:
```
<xarray.DataArray 'SSH' (time: 12, tile: 13, j: 90, i: 90)>
[1263600 values with dtype=float32]
Coordinates:
  * j        (j) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * i        (i) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
    XC       (tile, j, i) float32 ...
    YC       (tile, j, i) float32 ...
    rA       (tile, j, i) float32 ...
  * tile     (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
    iter     (time) int32 ...
  * time     (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
Attributes:
    units:          m
```

(continues on next page)

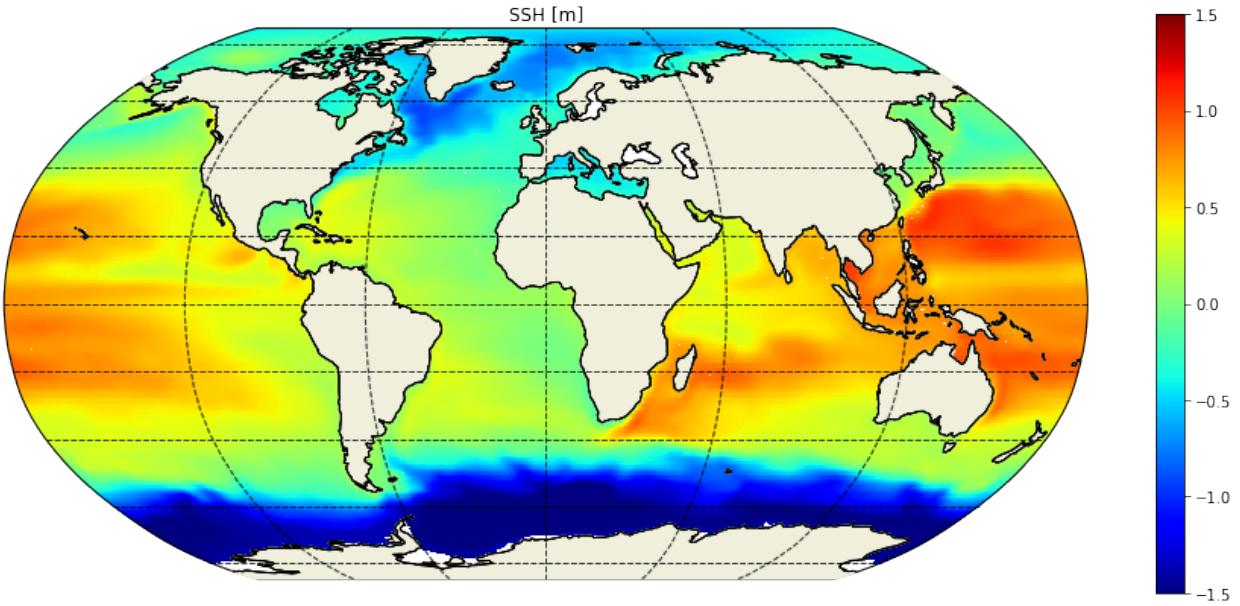
(continued from previous page)

```
    long_name:      Surface Height Anomaly adjusted with global steric height...
    standard_name:  sea_surface_height
```

SSH_dataset.SSH contains 12 months of data across 13 tiles.

Let's plot the first time record of this file.

[5]:
```
SSH  = SSH_dataset.SSH.isel(time=0)
# mask to nan where hFacC(k=0) = 0
SSH  = SSH.where(grid.hFacC.isel(k=0))

fig  = plt.figure(figsize=(16,7))
ecco.plot_proj_to_latlon_grid(grid.XC, grid.YC, SSH, show_colorbar=True, cmin=-1.5,␣
↪cmax=1.5);plt.title('SSH [m]');
```



Loading a single ECCOv4 variable NetCDF file using `load_ecco_var_from_years_nc`

We'll now load the same 2010 SSH file using `load_ecco_var_from_years_nc`. This time we specify the directory containing the NetCDF file, the variable that want to load and the year of interest.

[6]:
```
# single year:  2010
SSH_dataset_2010 = ecco.load_ecco_var_from_years_nc(SSH_dir, \
                          'SSH', years_to_load = [2010]).load()
SSH_dataset_2010.attrs = []
SSH_dataset_2010
```

[6]:
```
<xarray.Dataset>
Dimensions:    (i: 90, j: 90, nv: 2, tile: 13, time: 12)
Coordinates:
  * j          (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * i          (i) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
    XC         (tile, j, i) float32 -111.60647 -111.303 ... -111.86579
```

(continues on next page)

(continued from previous page)

```
    YC          (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
    rA          (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
  * tile        (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
    time_bnds   (time, nv) datetime64[ns] 2010-01-01 2010-02-01 ... 2011-01-01
    iter        (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
  * time        (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
Dimensions without coordinates: nv
Data variables:
    SSH         (time, tile, j, i) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
```

1.9.5 Loading a subset of single ECCOv4 variable NetCDF file using `load_ecco_var_from_years_nc`

One benefit of using `load_ecco_var_from_years_nc` over `open_dataset` is that you can optionally specify a subset of vertical levels or tiles to load using optional arguments with the *tiles_to_load* and *k_subset* optional arguments.

By default

- *tiles_to_load* = [0, 1, … 12]
- k_subset = []

To load a subset of tiles, specify the desired tile indices in *tiles_to_load*. For example, to load tiles 3,4 and 5: ~~~~ tiles_to_load = [3, 4, 5] ~~~~

To load a subset of depth levels, specify the desired depth level indices in *k_subset*. For example, to load the top 5 levels: ~~~~ k_subset = [0,1,2,3,4] ~~~~

In the following example we load `THETA` for tiles 7,8,9 and depth levels 0:34.

[7]:
```
theta_dir= ECCO_dir + '/nctiles_monthly/THETA/'
theta_subset = ecco.load_ecco_var_from_years_nc(theta_dir, \
                                                'THETA', years_to_load = [2010], \
                                                tiles_to_load = [ 7,8,9], \
                                                k_subset = [0,1,2,3,4]).load()
theta_subset.attrs = []
theta_subset
```

[7]:
```
<xarray.Dataset>
Dimensions:    (i: 90, j: 90, k: 5, nv: 2, tile: 3, time: 12)
Coordinates:
  * j          (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * i          (i) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k          (k) int32 0 1 2 3 4
    Z          (k) float32 -5.0 -15.0 -25.0 -35.0 -45.0
    PHrefC     (k) float32 49.05 147.15 245.25 343.35 441.45
    drF        (k) float32 10.0 10.0 10.0 10.0 10.0
    XC         (tile, j, i) float32 142.16208 142.22801 ... -115.5476 -115.18083
    YC         (tile, j, i) float32 67.47211 67.33552 ... -80.43542 -80.43992
    rA         (tile, j, i) float32 212633870.0 351016450.0 ... 47093870.0
    hFacC      (tile, k, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
  * tile       (tile) int32 7 8 9
    time_bnds  (time, nv) datetime64[ns] 2010-01-01 2010-02-01 ... 2011-01-01
    iter       (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
```

(continues on next page)

(continued from previous page)

```
  * time       (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
Dimensions without coordinates: nv
Data variables:
    THETA      (time, tile, k, j, i) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
```

As expected, `theta_subset` has 3 tiles and 5 vertical levels.

1.9.6 Loading multiple years of single ECCOv4 variable using `load_ecco_var_from_years_nc`

Another benefit of `load_ecco_var_from_years` is that you can load more than one year of output. First we'll show loading *two* years then *all* of the years of output available in the file directory

[8]:
```
# two years:  2010 and 2011
SSH_2010_2011 = ecco.load_ecco_var_from_years_nc(SSH_dir, \
                                                 'SSH',
                                                  years_to_load = [2010, 2011]).load()
SSH_2010_2011.attrs = []
SSH_2010_2011
```

[8]:
```
<xarray.Dataset>
Dimensions:    (i: 90, j: 90, nv: 2, tile: 13, time: 24)
Coordinates:
    rA         (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
    XC         (tile, j, i) float32 -111.60647 -111.303 ... -111.86579
    YC         (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
  * i          (i) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j          (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * tile       (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
    time_bnds  (time, nv) datetime64[ns] 2010-01-01 2010-02-01 ... 2012-01-01
    iter       (time) int32 158532 159204 159948 160668 ... 173844 174564 175308
  * time       (time) datetime64[ns] 2010-01-16T12:00:00 ... 2011-12-16T12:00:00
Dimensions without coordinates: nv
Data variables:
    SSH        (time, tile, j, i) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
```

Notice that SSH_2010_2011 has 24 time records

[9]:
```
# all years
SSH_all = ecco.load_ecco_var_from_years_nc(SSH_dir, \
                                           'SSH',
                                           years_to_load = 'all').load()
SSH_all.attrs = []
SSH_all
```

[9]:
```
<xarray.Dataset>
Dimensions:    (i: 90, j: 90, nv: 2, tile: 13, time: 288)
Coordinates:
    rA         (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
    XC         (tile, j, i) float32 -111.60647 -111.303 ... -111.86579
    YC         (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
  * i          (i) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
```

(continues on next page)

(continued from previous page)

```
  * j          (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * tile       (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
    time_bnds  (time, nv) datetime64[ns] 1992-01-01 1992-02-01 ... 2015-12-31
    iter       (time) int32 732 1428 2172 2892 ... 208164 208908 209628 210360
  * time       (time) datetime64[ns] 1992-01-16T12:00:00 ... 2015-12-16T12:00:00
Dimensions without coordinates: nv
Data variables:
    SSH        (time, tile, j, i) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
```

Now that we have all 288 month records of SSH.

1.9.7 Loading one or more years of more than variable using recursive_load_ecco_var_from_years_nc

With `recursive_load_ecco_var_from_years_nc` one can specify one or more variables to load, one or more years to load, while also requesting only a subset of tiles and vertical levels.

Let's demonstrate by first loading all of the information for monthly-averaged `SSH` and `THETA` fields for the years 2010, 2011, and 2012. We'll then load SSH and OBP for *all* of the years and tile 6. Finally, we'll load all of the monthly-mean variables for the year 2010.

Loading SSH and THETA for 2010-2012

[10]:
```
def is_year_tryexcept(s):
    try:
        return int(s) >= 1900
    except ValueError:
        return False
```

[11]:
```
x='ADVr_SLT_1992'
y=x.split('_')
for yy in y:
    print(is_year_tryexcept(yy))
```

```
False
False
True
```

[12]:
```
nctiles_monthly_dir = ECCO_dir + 'nctiles_monthly/'
SSH_THETA_2010_2012 = \
        ecco.recursive_load_ecco_var_from_years_nc(nctiles_monthly_dir, \
                                                   vars_to_load=['SSH','THETA'], \
                                                   years_to_load = [2010, 2011, 2012],␣
↪less_output=True).load()
```

```
loading files of  THETA
loading files of  SSH
```

[13]:
```
SSH_THETA_2010_2012
```

[13]:
```
<xarray.Dataset>
Dimensions:    (i: 90, j: 90, k: 50, nv: 2, tile: 13, time: 36)
Coordinates:
    rA         (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
    drF        (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    YC         (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
    PHrefC     (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
    Z          (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    hFacC      (tile, k, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    XC         (tile, j, i) float32 -111.60647 -111.303 ... -111.86579
  * i          (i) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j          (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k          (k) int32 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * tile       (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
    time_bnds  (time, nv) datetime64[ns] 2010-01-01 2010-02-01 ... 2013-01-01
    iter       (time) int32 158532 159204 159948 160668 ... 182628 183348 184092
  * time       (time) datetime64[ns] 2010-01-16T12:00:00 ... 2012-12-16T12:00:00
Dimensions without coordinates: nv
Data variables:
    THETA      (time, tile, k, j, i) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    SSH        (time, tile, j, i) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
Attributes:
    Conventions:                 CF-1.6
    Insitution:                  JPL
    Metadata_Conventions:        CF-1.6, Unidata Dataset Discovery v1.0, GDS...
    Project:                     Estimating the Circulation and Climate of t...
    author:                      Ian Fenty and Ou Wang
    cdm_data_type:               Grid
    date_created:                Mon May 13 18:32:24 2019
    geospatial_lat_max:          89.739395
    geospatial_lat_min:          -89.873055
    geospatial_lat_units:        degrees_north
    geospatial_lon_max:          179.98691
    geospatial_lon_min:          -179.98895
    geospatial_lon_units:        degrees_east
    geospatial_vertical_max:     -5.0
    geospatial_vertical_min:     -5906.25
    geospatial_vertical_units:   meter
    no_data:                     NaNf
    nx:                          90
    ny:                          90
    nz:                          50
    product_time_coverage_end:   2015-12-31T12:00:00
    product_time_coverage_start: 1992-01-01T12:00:00
    product_version:             ECCO Version 4 Release 3 (ECCOv4r3) 1992-2015
    time_coverage_end:           2013-01-01T00:00:00
    time_coverage_start:         2010-01-01T00:00:00
    time_units:                  days since 1992-01-01 00:00:00
```

We see three years (36 months) of data and 13 tiles and 50 vertical levels.

Loading SSH and OBP for all years and tile 6

Now let's demonstrate how the `recursive_load_ecco_var_from_years_nc` routine enable us to load all of the years of output for multiple variables. The trick is to specify the directory that contains all of the variables. The routine will recursively search all subdirectories for these fields. Note, this only works if the subdirectories are of the same temporal period (monthly mean, daily mean, or snapshots).

[14]:
```
SSH_OBP_2010_tile_6 = \
        ecco.recursive_load_ecco_var_from_years_nc(nctiles_monthly_dir, \
                                                   vars_to_load=['SSH', 'OBP'], \
                                                   years_to_load = 'all', \
                                                   tiles_to_load = 6).load()
```

```
loading files of  SSH
loading files of  OBP
```

[15]:
```
SSH_OBP_2010_tile_6
```

[15]:
```
<xarray.Dataset>
Dimensions:    (i: 90, j: 90, nv: 2, tile: 1, time: 288)
Coordinates:
    rA         (tile, j, i) float32 246414940.0 412417600.0 ... 246414940.0
    XC         (tile, j, i) float32 52.0 52.331654 ... -127.66834 -128.0
    YC         (tile, j, i) float32 67.57341 67.67698 ... 67.67698 67.57341
  * i          (i) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j          (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * tile       (tile) int32 6
    time_bnds  (time, nv) datetime64[ns] 1992-01-01 1992-02-01 ... 2015-12-31
    iter       (time) int32 732 1428 2172 2892 ... 208164 208908 209628 210360
  * time       (time) datetime64[ns] 1992-01-16T12:00:00 ... 2015-12-16T12:00:00
Dimensions without coordinates: nv
Data variables:
    SSH        (time, tile, j, i) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    OBP        (time, tile, j, i) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
Attributes:
    Conventions:                 CF-1.6
    Insitution:                  JPL
    Metadata_Conventions:        CF-1.6, Unidata Dataset Discovery v1.0, GDS...
    Project:                     Estimating the Circulation and Climate of t...
    author:                      Ian Fenty and Ou Wang
    cdm_data_type:               Grid
    date_created:                Tue May 14 10:19:12 2019
    geospatial_lat_max:          89.739395
    geospatial_lat_min:          67.57341
    geospatial_lat_units:        degrees_north
    geospatial_lon_max:          179.9739
    geospatial_lon_min:          -179.98895
    geospatial_lon_units:        degrees_east
    geospatial_vertical_max:     0
    geospatial_vertical_min:     0
    geospatial_vertical_units:   meter
    no_data:                     NaNf
    nx:                          90
    ny:                          90
```

(continues on next page)

(continued from previous page)

```
    nz:                           1
    product_time_coverage_end:    2015-12-31T12:00:00
    product_time_coverage_start:  1992-01-01T12:00:00
    product_version:              ECCO Version 4 Release 3 (ECCOv4r3) 1992-2015
    time_coverage_end:            2015-12-31T00:00:00
    time_coverage_start:          1992-01-01T00:00:00
    time_units:                   days since 1992-01-01 00:00:00
```

1.9.8 Loading the entire (or large fractions) of the entire ECCOv4 solution using Dask

We can load the entire ECCOv4 solution into our workspace and perform calcuations with the entire solution thanks to the amazing `Dask` library implemented in the `xarray` package. Why is this useful? Because it is unlikely that the machine you are working on has enough RAM to load the entire ECCOv4 solution at one time. By using `Dask` we can *virtually* load all of the ECCO fields into memory. `Dask` even enables us to do calculations with these fields even though the entire data is never stored in memory.

Here is some more information about these features from the `Dask` website, https://docs.dask.org/

1. **Larger-than-memory:** Lets you work on datasets that are larger than your available memory by breaking up your array into many small pieces, operating on those pieces in an order that minimizes the memory footprint of your computation, and effectively streaming data from disk.*
2. **Blocked Algorithms:** Perform large computations by performing many smaller computations

Finally, in cluster environments `Dask` can distribute computations across many cores to speed up large redundant calculation.

An in-depth description of `Dask` is outside the scope of this tutorial. For the moment, let us compare the operation of loading **all** the 2D daily-mean fields for 2010 both using `Dask` and without using `Dask`. Without `Dask` these fields will be loaded into memory. With `Dask` we will only load a minimum of the `Datasets`, the Dimensions and Coordinates.

To demonstate some of the advantages of using `DASK` to load and analyze ECCO fields, let's load several monthly-mean variable for two years with and without `Dask`. Then we'll load the monthly-mean fields for four years using Dask. At the end we'll compare their *times to load* and *memory footprints*.

Example 1a: Load two years monthly-mean ECCO fields into memory without `Dask`

The `.load()` suffix appended to the end of these load commands is a command to fully load fields into memory. Without it, `Dask` only *virtually* loads the fields.

[16]:
```
mon_mean_dir = ECCO_dir + 'nctiles_monthly/'
import time
t_0 = time.time()

large_subset_no_dask = \
    ecco.recursive_load_ecco_var_from_years_nc(mon_mean_dir, \
                                               vars_to_load=['THETA','SSH','UVEL','VVEL',
↪\
                                                             'ADVr_SLT','ADVr_TH',\
                                                             'ADVx_SLT','ADVy_SLT',\
                                                             'ADVxHEFF','ADVxSNOW', \
                                                             'OBP','SIarea','SIheff'],
```

(continues on next page)

(continued from previous page)

```
                                                years_to_load=[2010, 2011],\
                                                less_output=True).load()

delta_t_2_yrs_no_dask =  time.time() - t_0
print(delta_t_2_yrs_no_dask)
```

```
loading files of  THETA
loading files of  SSH
loading files of  UVEL
loading files of  VVEL
loading files of  ADVr_SLT
loading files of  ADVr_TH
loading files of  ADVxHEFF
loading files of  ADVxSNOW
loading files of  ADVx_SLT
loading files of  SIarea
loading files of  OBP
loading files of  SIheff
11.277666807174683
```

Example 1b: Load two years of monthly-mean ECCO fields into memory using Dask

This time we will omit the `.load()` suffix and use Dask.

[17]:
```
t_0 = time.time()
large_subset_with_dask = ecco.recursive_load_ecco_var_from_years_nc(mon_mean_dir, \
                                            vars_to_load=['THETA','SSH','UVEL','VVEL',\
                                                          'ADVr_SLT','ADVr_TH',\
                                                          'ADVx_SLT','ADVy_SLT',\
                                                          'ADVxHEFF','ADVxSNOW', \
                                                          'OBP','SIarea','SIheff'],
                                            years_to_load=[2010, 2011])
delta_t_2_yrs_with_dask =  time.time() - t_0
print(delta_t_2_yrs_with_dask)
```

```
loading files of  THETA
loading files of  SSH
loading files of  UVEL
loading files of  VVEL
loading files of  ADVr_SLT
loading files of  ADVr_TH
loading files of  ADVxHEFF
loading files of  ADVxSNOW
loading files of  ADVx_SLT
loading files of  SIarea
loading files of  OBP
loading files of  SIheff
3.1377756595611572
```

Example 2: Load 5 years of monthly-mean field with DASK

[18]:
```
t_0 = time.time()
all_fields_with_dask = ecco.recursive_load_ecco_var_from_years_nc(mon_mean_dir, \
                                            vars_to_load=['THETA','SSH','UVEL','VVEL',\
                                                          'ADVr_SLT','ADVr_TH',\
                                                          'ADVx_SLT','ADVy_SLT',\
                                                          'ADVxHEFF','ADVxSNOW', \
                                                          'OBP','SIarea','SIheff'], \
                                            years_to_load=[2008, 2009, 2010, 2011, 2012],␣
↪\
                                            dask_chunk=True)
delta_t_5_yrs_with_dask =  time.time() - t_0
print (delta_t_5_yrs_with_dask)
```

```
loading files of  THETA
loading files of  SSH
loading files of  UVEL
loading files of  VVEL
loading files of  ADVr_SLT
loading files of  ADVr_TH
loading files of  ADVxHEFF
loading files of  ADVxSNOW
loading files of  ADVx_SLT
loading files of  SIarea
loading files of  OBP
loading files of  SIheff
6.462316274642944
```

Results

Now we examine the time it took to load these fields and the comparative memory footprints

[19]:
```
print ('loaded 2 years without dask in ', np.round(delta_t_2_yrs_no_dask), 'sec')
print ('loaded 2 years with_dask in    ', np.round(delta_t_2_yrs_with_dask), 'sec')
print ('loaded 5 years with_dask in    ', np.round(delta_t_5_yrs_with_dask), 'sec')
```

```
loaded 2 years without dask in  11.0 sec
loaded 2 years with_dask in     3.0 sec
loaded 5 years with_dask in     6.0 sec
```

The real advantage of using Dask comes when examining the size of these objects in memory. The 'pympler' package allows you to see the memory footprint of these objects.

https://pythonhosted.org/Pympler/index.html

To proceed, install the pympler library into your Python environment

[20]:
```
# Estimate the memory footprint in MB
from pympler import asizeof

s = 0
F = large_subset_no_dask
for i in F.variables.keys():
```

(continues on next page)

(continued from previous page)

```
    s += asizeof.asizeof(large_subset_no_dask[i])
print('large_subset_no_dask   : ', np.round(s/2**20), 'mb')

F = large_subset_with_dask
s = 0
for i in F.variables.keys():
    s += asizeof.asizeof(F[i])
print('large_subset_with_dask : ', np.round(s/2**20), 'mb')

F = all_fields_with_dask
s = 0
for i in F.variables.keys():
    s += asizeof.asizeof(F[i])
print('all_fields_with_dask   : ', np.round(s/2**20), 'mb')
```

```
large_subset_no_dask   :  3120.0 mb
large_subset_with_dask :  171.0 mb
all_fields_with_dask   :  172.0 mb
```

Using `Dask`, we were able to load 5 years of data faster than 2 years without Dask.

In terms of memory, the 5 years of data in *all_fields_with_dask* object takes a fraction of the memory of the 2 years of data in *large_subset_no_dask*. With much less memory reserved to hold all of the fields, we have more memory avaiable for calculations on the parts of the fields that we care about.

Go ahead and experiment with using `Dask` to load the daily-averaged fields. Because all of the daily-averaged fields in the standard ECCO product are 2D, loading them with `Dask` takes very little time!

1.9.9 Summary

Now you know efficient ways to load ECCOv4 NetCDF files, both when you are reading variables split into *tiles* and when you are reading variables aggregated by year.

Using `Dask` we showed that one can prepare a work environment where ECCO model variables are accessible for calculations even without fully loading the fields into memory.

1.10 ECCOv4 Loading llc binary files in the 'compact' format

1.10.1 Objective

To teach how to loading ECCO binary files written by the MITgcm in the llc 'compact' format.

1.10.2 Introduciton

When the MITgcm saves diagnostic and other fields to files it does so using the so-called 'compact' format. The compact format distributes the arrays from the 13 lat-lon-cap tiles in a somewhat unintuitive manner. Fortunately, it is not difficult to extract the 13 tiles from 'compact' format files. This tutorial will show you how to use the 'read_llc_to_tiles' subroutine to read and re-organize MITgcm's files written in compact format into a more familiar 13-tile layout.

1.10.3 Objectives

By the end of the tutorial you will be able to read llc compact binary files of any dimension, plot them, and convert them into DataArrays.

[1]:
```
## Import the ecco_v4_py library into Python
## =========================================

## -- If ecco_v4_py is not installed in your local Python library,
##    tell Python where to find it.  For example, if your ecco_v4_py
##    files are in /Users/ifenty/ECCOv4-py/ecco_v4_py, then use:
import sys

sys.path.append('/Users/ifenty/git_repos/my_forks/ECCOv4-py')
import ecco_v4_py as ecco

import matplotlib.pyplot as plt
import numpy as np
import xarray as xr
```

1.10.4 The *read_llc_to_tiles* subroutine

read_llc_to_tiles reads a llc compact format binary file and converts to a numpy ndarray of dimension: [N_recs, N_z, N_tiles, llc, llc]

For ECCOv4 our convention is:

```
'N_recs' = number of time levels
'N_z' = number of depth levels
'N_tiles' = 13
'llc' = 90
```

By default the routine will try to load a single 2D slice of a llc90 compact binary file: (N_recs = 1, N_z =1, N_tiles = 13, and llc=90).

There are several other options which you can learn about using the 'help' command:

[2]:
```
help(ecco.read_llc_to_tiles)
```

```
Help on function read_llc_to_tiles in module ecco_v4_py.read_bin_llc:

read_llc_to_tiles(fdir, fname, llc=90, skip=0, nk=1, nl=1, filetype='>f', less_
↪output=False, use_xmitgcm=False)
    Loads an MITgcm binary file in the 'tiled' format of the
    lat-lon-cap (LLC) grids with dimension order:

        [N_recs, N_z, N_tiles, llc, llc]

    where if either N_z or N_recs =1, then that dimension is collapsed
    and not present in the returned array.

    if use_xmitgcm == True

        data are read in via the low level routine
        xmitgcm.utils.read_3d_llc_data and returned as dask array.

        Hint: use data_tiles.compute() to load into memory.

    if use_xmitgcm == False

        Loads an MITgcm binary file in the 'compact' format of the
        lat-lon-cap (LLC) grids and converts it to the '13 tiles' format
        of the LLC grids.

    Parameters
    ----------
    fdir : string
        A string with the directory of the binary file to open
    fname : string
        A string with the name of the binary file to open
    llc : int
        the size of the llc grid.  For ECCO v4, we use the llc90 domain
        so `llc` would be `90`.
        Default: 90
    skip : int
        the number of 2D slices (or records) to skip.
        Records could be vertical levels of a 3D field, or different 2D fields, or both.
    nk : int
        number of 2D slices (or records) to load in the third dimension.
        if nk = -1, load all 2D slices
        Default: 1 [singleton]
    nl : int
        number of 2D slices (or records) to load in the fourth dimension.
        Default: 1 [singleton]
    filetype: string
        the file type, default is big endian (>) 32 bit float (f)
        alternatively, ('<d') would be little endian (<) 64 bit float (d)
    less_output : boolean
        A debugging flag.  False = less debugging output
        Default: False
```

(continues on next page)

(continued from previous page)

```
    use_xmitgcm : boolean
        option to use the routine xmitgcm.utils.read_3d_llc_data into a dask
        array, i.e. not into memory.
        Otherwise read in as a compact array, convert to faces, then to tiled format
        Default: False

    Returns
    -------
    data_tiles
        a numpy array of dimension 13 x nl x nk x llc x llc, one llc x llc array
        for each of the 13 tiles and nl and nk levels.
```

1.10.5 Related routines

Two related routines which you might find useful:

1. *read_llc_to_compact*: Loads an MITgcm binary file in the 'compact' format of the lat-lon-cap (LLC) grids and preserves its original dimension
2. *read_llc_to_faces* : Loads an MITgcm binary file in the 'compact' format of the lat-lon-cap (LLC) grids and converts it to the '5 faces' dictionary.

For the remainder of the tutorial we will only use *read_llc_to_tiles*.

1.10.6 Example 1: Load a 2D llc 'compact' binary file

The file 'bathy_eccollc_90x50_min2pts.bin' contains the 2D array of bathymetry for the model.

[3]:
```
input_dir = '/Users/ifenty/tmp/input_init/'
input_file = 'bathy_eccollc_90x50_min2pts.bin'
```

read_llc_to_tiles actually runs several other subroutines: *load_binary_array* which does the lower level reading of the binary file, *llc_compact_to_faces* which converts the array to 5 'faces', and finally *llc_faces_to_tiles* which extracts the 13 tiles from the 5 faces:

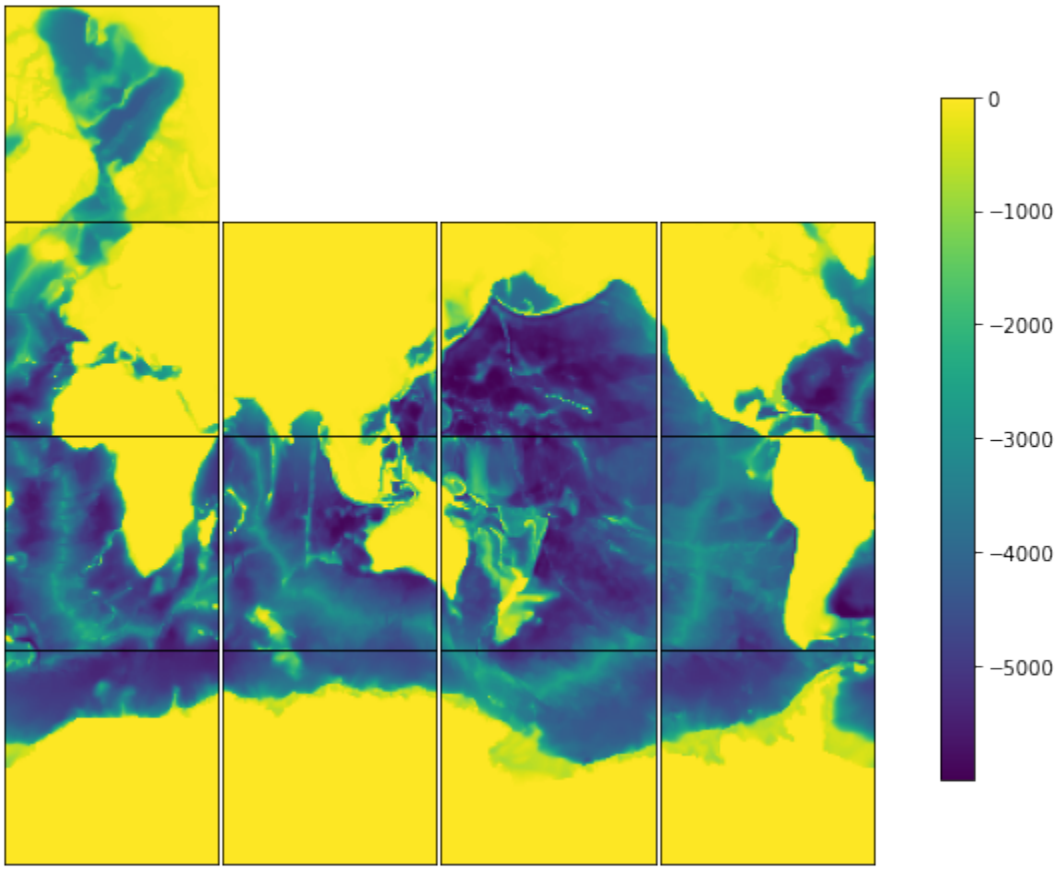
[4]:
```
bathy = ecco.read_llc_to_tiles(input_dir, input_file)
```

```
load_binary_array: loading file /Users/ifenty/tmp/input_init/bathy_eccollc_90x50_min2pts.
↪bin
load_binary_array: data array shape  (1170, 90)
load_binary_array: data array type  >f4
llc_compact_to_faces: dims, llc  (1170, 90) 90
llc_compact_to_faces: data_compact array type  >f4
llc_faces_to_tiles: data_tiles shape  (13, 90, 90)
llc_faces_to_tiles: data_tiles dtype  >f4
```

bathy is a numpy float32 array with dimension [13, 90, 90]

Plot the 13 tiles bathymetry data

[5]:
```
# Use plot_tiles to make a quick plot of the 13 tiles.  See the tutorial on plotting for␣
↪more examples.
ecco.plot_tiles(bathy, layout='latlon',rotate_to_latlon=True,show_tile_labels=False,␣
↪show_colorbar=True);
```

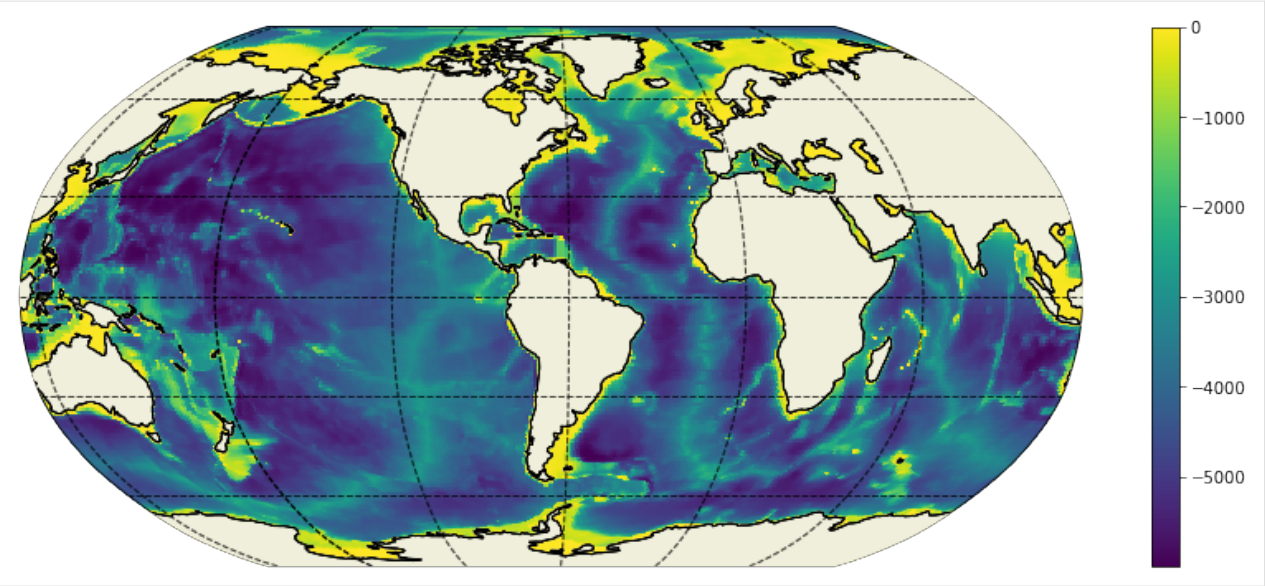


1.10.7 Load ecco-grid information to make a fancier lat-lon plot

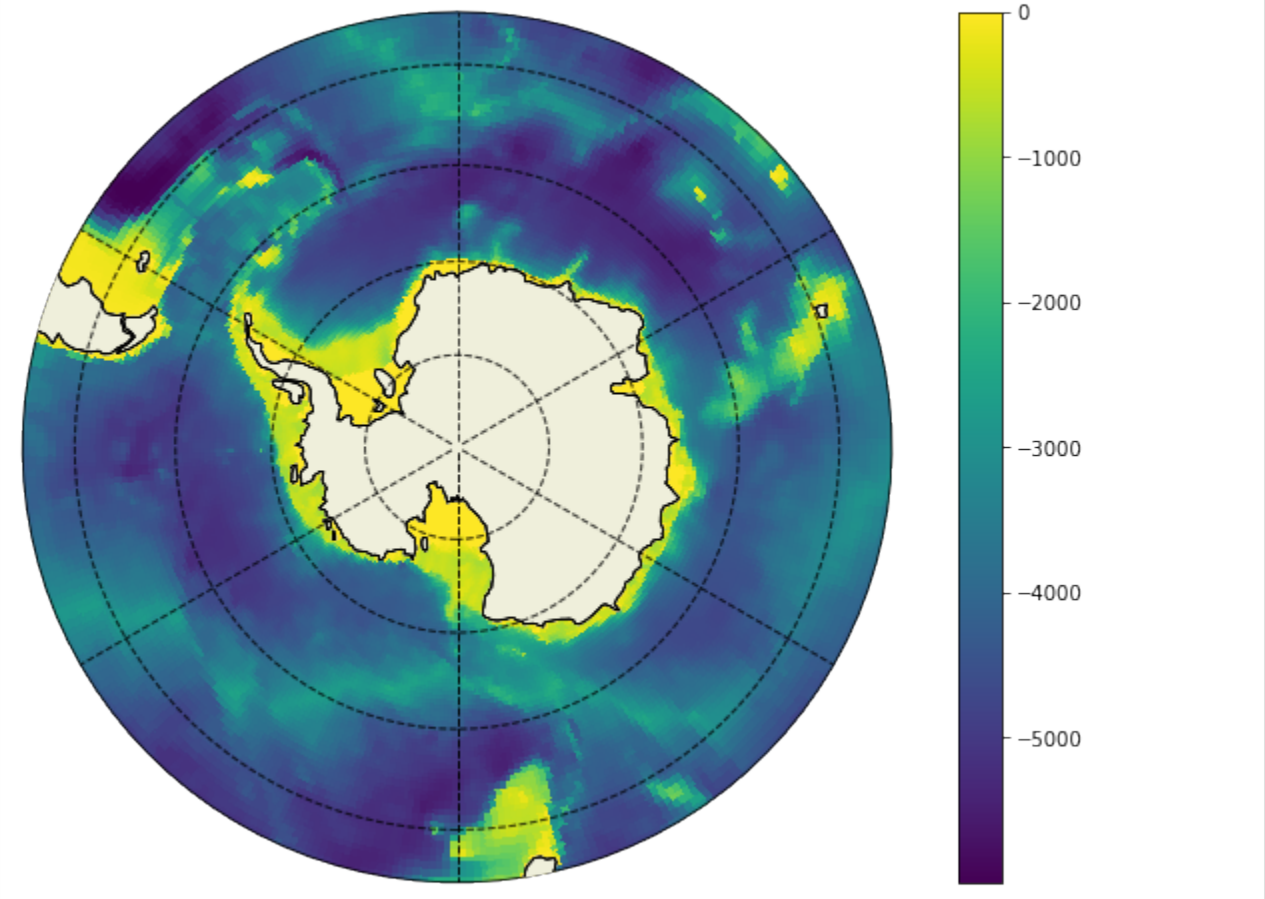
With the longitudes (XC) and latitudes (YC) and the 13 tile ndarray we can plot the field in different geographic projections. See the tutorial on plotting for more examples.

[6]:
```
ecco_grid_dir = '/Users/ifenty/tmp/nctiles_grid/'
ecco_grid = ecco.load_ecco_grid_nc(input_dir, 'ECCO-GRID.nc')
```

[7]:
```
plt.figure(figsize=(15,6));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, bathy, show_colorbar=True,␣
↪user_lon_0=-66);
```



[8]:
```
plt.figure(figsize=(12,8));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, bathy, show_colorbar=True,\
                              projection_type='stereo', lat_lim =-45, less_output=True,
↪dx=.25,dy=.25);
```



1.10.8 Convert the ndarray into a DataArray

Converting the ndarray to a DataArray can be useful for broadcasting operations. One approach is to create the DataArray manually by specifying the names and values of the new dimension coordinates:

[9]:
```
tile = range(13)
i = range(90)
j = range(90)
```

[10]:
```
# Convert numpy array to an xarray DataArray with matching dimensions as the monthly␣
↪mean fields
bathy_DA = xr.DataArray(bathy,coords={'tile': tile,
                                      'j': j,
                                      'i': i}, dims=['tile','j','i'])
```

[11]:
```
print(bathy_DA.dims)
```

```
('tile', 'j', 'i')
```

Another approach is to use the routine *llc_tiles_to_xda*. *llc_tiles_to_xda* uses

[12]:
```
bathy.shape
```

[12]:
```
(13, 90, 90)
```

[13]:
```
bathy_DA2 = ecco.llc_tiles_to_xda(bathy, var_type='c',grid_da=ecco_grid.XC)
print(bathy_DA2.dims)
print(bathy_DA2.coords)
```

```
('tile', 'j', 'i')
Coordinates:
  * j        (j) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * i        (i) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * tile     (tile) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
    XC       (tile, j, i) float32 -111.60647 -111.303 ... -105.58465 -111.86579
    YC       (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
    CS       (tile, j, i) float32 0.06157813 0.06675376 ... -0.9983638
    SN       (tile, j, i) float32 -0.99810225 -0.9977695 ... -0.057182025
    Depth    (tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    rA       (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
```

1.10.9 Example 2: Load a 3D 'compact' llc binary file with 3rd dimension = Time

The file 'runoff-2d-Fekete-1deg-mon-V4-SMOOTH.bin' contains the 12 month climatology of river runoff, dimensions of [time, j, i].

[14]:
```
input_file = 'runoff-2d-Fekete-1deg-mon-V4-SMOOTH.bin'
```

specify the length of the n_recs dimension, nl, as 12. By default, nk = 1

[15]:
```
runoff = ecco.read_llc_to_tiles(input_dir, input_file, nl=12)
```

```
load_binary_array: loading file /Users/ifenty/tmp/input_init/runoff-2d-Fekete-1deg-mon-
↪V4-SMOOTH.bin
load_binary_array: data array shape  (12, 1, 1170, 90)
load_binary_array: data array type  >f4
llc_compact_to_faces: dims, llc  (12, 1, 1170, 90) 90
llc_compact_to_faces: data_compact array type  >f4
llc_faces_to_tiles: data_tiles shape  (12, 1, 13, 90, 90)
llc_faces_to_tiles: data_tiles dtype  >f4
```

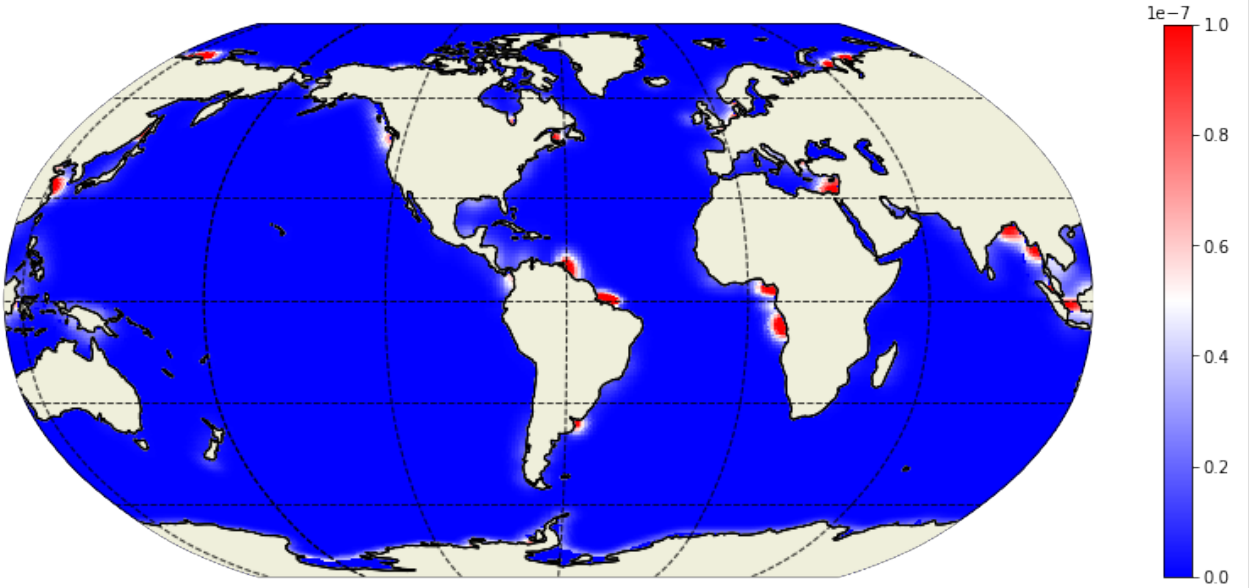
Shape is n_recs (12), n_z (1), n_tiles (13), llc (90), llc (90)

[16]:
```
print(runoff.shape)
```

```
(12, 1, 13, 90, 90)
```

Plot the November runoff climatology

[17]:
```
plt.figure(figsize=(15,6));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, runoff[10,:], cmin=0,cmax=1e-7,
↪\
                              show_colorbar=True, cmap='bwr',user_lon_0=-66);
```



1.10.10 Convert the ndarray into a DataArray

Converting the ndarray to a DataArray can be useful for broadcasting operations. Two methods: 'manual' and using *llc_tiles_to_xda*.

Method 1: Manual

[18]:
```
tile = range(1,14)
i = range(90)
j = range(90)
time = range(12) # months
k = [0];
```

[19]:
```
# Convert numpy array to an xarray DataArray with matching dimensions as the monthly␣
↪mean fields
runoff_DA = xr.DataArray(runoff,coords={'time': time,
                                        'k': k,
                                        'tile': tile,
                                        'j': j,
                                        'i': i}, dims=['time','k','tile','j','i'])
```

[20]:
```
print(runoff_DA.dims)
print(runoff_DA.shape)
```

```
('time', 'k', 'tile', 'j', 'i')
(12, 1, 13, 90, 90)
```

Method 2: *llc_tiles_to_xda*

runoff is a 3D array with the 3rd dimension being time and therefore we need to pass *llc_tiles_to_xda* a similarly-dimensioned DataArray or tell the subroutine that the 3rd dimension is depth. The 'ecco_grid' DataSet doesn't have any similary-dimensioned DataArrays (no DataArrays with a time dimension). Therefore we will tell the routine that the new 4th dimension should be time:

[21]:
```
##### specify that the 5th dimension should be time
runoff_DA2 = ecco.llc_tiles_to_xda(runoff, var_type='c',dim4='depth', dim5='time')
print(runoff_DA2.dims)
print(runoff_DA2.coords)
```

```
('time', 'k', 'tile', 'j', 'i')
Coordinates:
  * k        (k) int64 0
  * time     (time) int64 0 1 2 3 4 5 6 7 8 9 10 11
  * tile     (tile) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
  * j        (j) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * i        (i) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
```

1.10.11 Example 3: Load a 3D 'compact' llc binary file with 3rd dimension = Depth

The file 'total_kapredi_r009bit11.bin' is a 50 depth level array of the adjusted Redi parameter from Release 1 (first guess + adjustments), dimensions of [depth, j, i]. Release 4's adjusted Redi parameter can be found at https://archive.podaac.earthdata.nasa.gov/podaac-ops-cumulus-protected/ECCO_L4_OCEAN_3D_MIX_COEFFS_LLC0090GRID_V4R4/OCEAN_3D_MIXING_COEFFS_ECCO_V4r4_native_llc0090.nc.

[22]:
```
input_file = 'total_kapredi_r009bit11.bin'
```

specify the number of depth levels as 50. n_recs defaults to 1 and is dropped by default.

[23]:
```
kapredi = ecco.read_llc_to_tiles(input_dir, input_file, nk=50)
```

```
load_binary_array: loading file /Users/ifenty/tmp/input_init/total_kapredi_r009bit11.bin
load_binary_array: data array shape  (50, 1170, 90)
load_binary_array: data array type  >f4
llc_compact_to_faces: dims, llc  (50, 1170, 90) 90
llc_compact_to_faces: data_compact array type  >f4
llc_faces_to_tiles: data_tiles shape  (50, 13, 90, 90)
llc_faces_to_tiles: data_tiles dtype  >f4
```

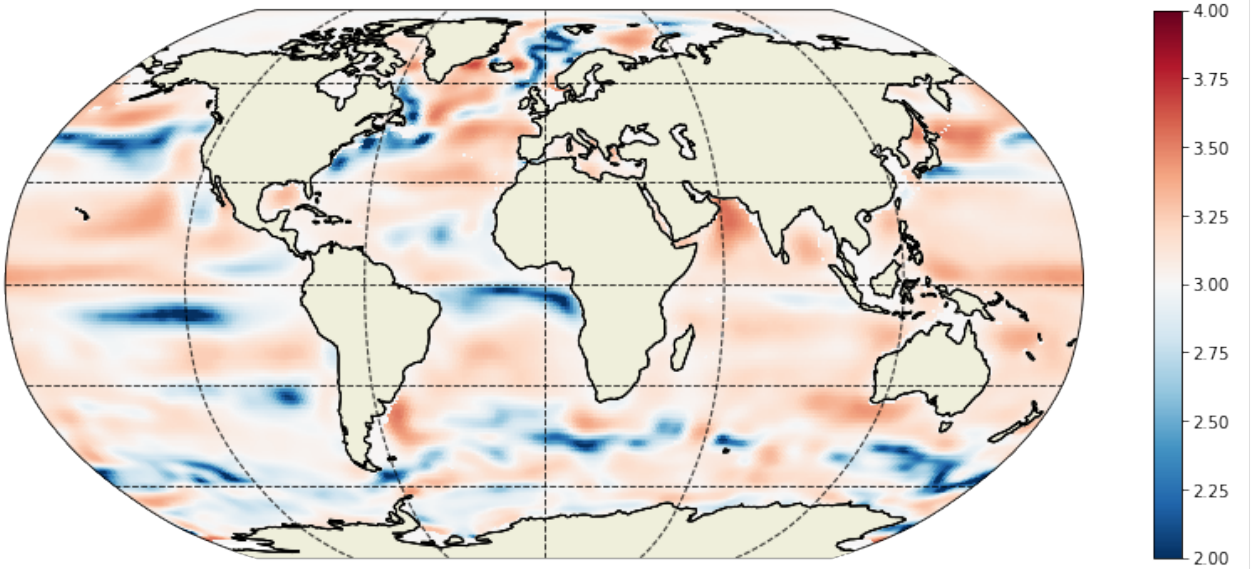
[24]:
```
print(kapredi.shape)
```

```
(50, 13, 90, 90)
```

Plot log10 of the parameter at the 10th depth level (105m)

[25]:
```
plt.figure(figsize=(15,6));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, np.log10(kapredi[10,:]),\
                              cmin=2,cmax=4,show_colorbar=True);
```

```
<ipython-input-25-b09b1aa1ac7e>:2: RuntimeWarning: divide by zero encountered in log10
  ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, np.log10(kapredi[10,:]),\
```



1.10.12 Convert the ndarray into a DataArray

Converting the ndarray to a DataArray can be useful for broadcasting operations. Two methods: 'manual' and using *llc_tiles_to_xda*.

Method 1: Manual

[26]:
```
tile = range(1,14)
i = range(90)
j = range(90)
k = range(50)
```

[27]:
```
# Convert numpy array to an xarray DataArray with matching dimensions as the monthly␣
↪mean fields
kapredi_DA = xr.DataArray(kapredi,coords={'k': k,
                                          'tile': tile,
                                          'j': j,
                                          'i': i},dims=['k','tile','j','i'])
```

[28]:
```
print(kapredi_DA.dims)
print(kapredi_DA.shape)
```

```
('k', 'tile', 'j', 'i')
(50, 13, 90, 90)
```

Method 2: *llc_tiles_to_xda*

kapredi is a 4D array (depth, tile, j, i) and therefore we need to pass *llc_tiles_to_xda* a similarly-dimensioned DataArray or tell the subroutine that the 4th dimension is depth

[29]:
```
# use similarly-dimensioned DataArray: hFacC
kapredi_DA2 = ecco.llc_tiles_to_xda(kapredi, var_type='c',grid_da=ecco_grid.hFacC)
print(kapredi_DA2.dims)
print(kapredi_DA2.coords)
```

```
('k', 'tile', 'j', 'i')
Coordinates:
  * k        (k) int64 0 1 2 3 4 5 6 7 8 9 10 ... 40 41 42 43 44 45 46 47 48 49
  * j        (j) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * i        (i) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * tile     (tile) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
    XC       (tile, j, i) float32 -111.60647 -111.303 ... -105.58465 -111.86579
    YC       (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
    CS       (tile, j, i) float32 0.06157813 0.06675376 ... -0.9983638
    SN       (tile, j, i) float32 -0.99810225 -0.9977695 ... -0.057182025
    Z        (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    Depth    (tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    rA       (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
    PHrefC   (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
    drF      (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    hFacC    (k, tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    maskC    (k, tile, j, i) bool False False False False ... False False False
```

[30]:
```
# specify that the 4th dimension should be depth
kapredi_DA3 = ecco.llc_tiles_to_xda(kapredi, var_type='c', dim4='depth')
print(kapredi_DA3.dims)
print(kapredi_DA3.coords)
```

```
('k', 'tile', 'j', 'i')
Coordinates:
  * k        (k) int64 0 1 2 3 4 5 6 7 8 9 10 ... 40 41 42 43 44 45 46 47 48 49
  * tile     (tile) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
  * j        (j) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * i        (i) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
```

1.10.13 Parting thoughts

1. *read_llc_to_tiles* can also be used to read ECCO '*.data*' generated when re-running the ECCO model.
2. Converting from numpy ndarrays or xarray DataArrays in the 'tile' format to a 5-faces format or compact format can be made with routines like: *llc_tiles_to_compact* and *llc_tiles_to_faces*

llc_tiles_to_faces

[31]:
```
help(ecco.llc_tiles_to_faces)
```

```
Help on function llc_tiles_to_faces in module ecco_v4_py.llc_array_conversion:

llc_tiles_to_faces(data_tiles, less_output=False)
    Converts an array of 13 'tiles' from the lat-lon-cap grid
    and rearranges them to 5 faces.  Faces 1,2,4, and 5 are approximately
    lat-lon while face 3 is the Arctic 'cap'

    Parameters
    ----------
    data_tiles :
        An array of dimension 13 x nl x nk x llc x llc

    If dimensions nl or nk are singular, they are not included
        as dimensions of data_tiles

    less_output : boolean
        A debugging flag.  False = less debugging output
        Default: False

    Returns
    -------
    F : dict
        a dictionary containing the five lat-lon-cap faces

        F[n] is a numpy array of face n, n in [1..5]

        dimensions of each 2D slice of F

        - f1,f2: 3*llc x llc
        -    f3: llc x llc
        - f4,f5: llc x 3*llc
```

llc_tiles_to_compact

[32]:
```
help(ecco.llc_tiles_to_compact)
```

```
Help on function llc_tiles_to_compact in module ecco_v4_py.llc_array_conversion:

llc_tiles_to_compact(data_tiles, less_output=False)
    Converts a numpy binary array in the 'compact' format of the
    lat-lon-cap (LLC) grids and converts it to the '13 tiles' format
    of the LLC grids.

    Parameters
    ----------
    data_tiles : ndarray
        a numpy array organized by, at most,
        13 tiles x nl x nk x llc x llc

        where dimensions 'nl' and 'nk' are optional.

    less_output : boolean, optional, default False
        A debugging flag.  False = less debugging output

    Returns
    -------
    data_compact : ndarray
        a numpy array of dimension nl x nk x 13*llc x llc

    Note
    ----
    If dimensions nl or nk are singular, they are not included
    as dimensions in data_compact
```

1.11 Combining multiple `Datasets`

1.11.1 Objectives:

Show how to combine multiple ECCO v4 state estimate `Datasets` after loading.

1.11.2 Loading multiple `Datasets`

In previous tutorials we've loaded single lat-lon-cap NetCDF tile files for ECCO state estimate variables and model grid parameters. Here we will demonstrate merging the resulting `Datasets` together. Some benefits of merging `Datasets` include having a tidier workspace and simplying subsetting operations, the subject of a future tutorial.

First, we'll load three ECCOv4 NetCDF state estimate variables and the model grid. For this exercise use the ECCOv4 NetCDF files in Format 2 (one NetCDF file corresponds to one variable and one year).

First let's define our environment

[1]:
```
import numpy as np
import xarray as xr
import sys
import matplotlib.pyplot as plt
import json
```

[2]:
```
## Import the ecco_v4_py library into Python
## =========================================
#import ecco_v4_py as ecco

## -- If ecco_v4_py is not installed in your local Python library,
##    tell Python where to find it.  For example, if your ecco_v4_py
##    files are in /Users/ifenty/ECCOv4-py/ecco_v4_py, then use:

sys.path.append('/home/ifenty/ECCOv4-py')
import ecco_v4_py as ecco
```

[3]:
```
## Set top-level file directory for the ECCO NetCDF files
## ======================================================================
# base_dir = '/home/username/'
base_dir = '/home/ifenty/ECCOv4-release'

## define a high-level directory for ECCO fields
ECCO_dir = base_dir + '/Release3_alt'
```

c point: SSH

[4]:
```
SSH_dir= ECCO_dir + '/nctiles_monthly/SSH/'
var = 'SSH'

# make use of the ``load_ecco_var_from_years_nc`` routine to
# load a single variable
ecco_dataset_A = ecco.load_ecco_var_from_years_nc(SSH_dir, \
                                                  var, \
                                                  tiles_to_load=[0,1,2,3],\
                                                  years_to_load=[2010])
```

to see the data variables in a dataset, use `.data_vars`:

[5]:
```
ecco_dataset_A.data_vars
```

[5]:
```
Data variables:
    SSH      (time, tile, j, i) float32 dask.array<chunksize=(12, 4, 90, 90), meta=np.
↪ndarray>
```

As expected, *ecco_dataset_A* has one `data variable`, *SSH*, which has dimensions **i**, **j**, **tile**, and **time**. Its coordinates:

[6]:
```
ecco_dataset_A.SSH.coords
```

[6]:
```
Coordinates:
  * j        (j) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * i        (i) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
```

(continues on next page)

(continued from previous page)

```
    XC       (tile, j, i) float32 dask.array<chunksize=(4, 90, 90), meta=np.ndarray>
    YC       (tile, j, i) float32 dask.array<chunksize=(4, 90, 90), meta=np.ndarray>
    rA       (tile, j, i) float32 dask.array<chunksize=(4, 90, 90), meta=np.ndarray>
  * tile     (tile) int32 0 1 2 3
    iter     (time) int32 dask.array<chunksize=(12,), meta=np.ndarray>
  * time     (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
```

u point: ADVx_TH

ADVx_TH is the horizontal advective flux of potential temperature in each tile's x direction.

[7]:
```
ADVx_dir= ECCO_dir + '/nctiles_monthly/ADVx_TH/'
var = 'ADVx_TH'

# make use of the ``load_ecco_var_from_years_nc`` routine to
# load a single variable
ecco_dataset_B= ecco.load_ecco_var_from_years_nc(ADVx_dir, \
                                                 var, \
                                                 years_to_load=2010).load();
ecco_dataset_B.attrs=[]

ecco_dataset_B.data_vars
```

[7]:
```
Data variables:
    ADVx_TH  (time, tile, k, j, i_g) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
```

ecco_dataset_A has one `data variable`, *ADVx_TH*, which has dimensions **i_g**, **j**, **k** **tile**, and **time**. *ADVx_TH* is a three-dimensional field.

Notice that the coordinates of *ecco_dataset_B* is distinct from *ecco_dataset_A*. Specifically, *ecco_dataset_B* has a **k** dimension, and several new non-dimension coords such as **drF** and **dyG**

[8]:
```
ecco_dataset_B.coords
```

[8]:
```
Coordinates:
  * i_g        (i_g) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j          (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k          (k) int32 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
    Z          (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    PHrefC     (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
    drF        (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    dxC        (tile, j, i_g) float32 15583.418 15588.104 ... 23406.256
    rAw        (tile, j, i_g) float32 361699460.0 362790240.0 ... 364760350.0
    dyG        (tile, j, i_g) float32 23210.262 23273.26 ... 15595.26 15583.685
    hFacW      (tile, k, j, i_g) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
  * tile       (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
    time_bnds  (time, nv) datetime64[ns] 2010-01-01 2010-02-01 ... 2011-01-01
    iter       (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
  * time       (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
```

v point: ADVy_TH

ADVy_TH is the horizontal advective flux of potential temperature in each tile's y direction.

[9]:
```
ADVy_dir= ECCO_dir + '/nctiles_monthly/ADVy_TH/'
var = 'ADVy_TH'

ecco_dataset_C= ecco.load_ecco_var_from_years_nc(ADVy_dir, \
                                                 var, \
                                                 years_to_load=2010).load();
ecco_dataset_C.attrs=[]
ecco_dataset_C.data_vars
```

[9]:
```
Data variables:
    ADVy_TH  (time, tile, k, j_g, i) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
```

ecco_dataset_C has one data variable, *ADVy_TH*, which has dimensions **i**, **j_g**, **k** **tile**, and **time**. *ADVy_TH* is a three-dimensional field.

Examining the dimensions and coordinates of these Datasets

Each of our three Datasets contain a single DataArray. Each of these DataArray objects has different horizontal dimension labels.

- **i** and **j** for *SSH*
- **i_g** and **j** for *ADVx_TH*
- **i** and **j_g** for *ADVy_TH*

[10]:
```
print(ecco_dataset_A.data_vars)
print(ecco_dataset_B.data_vars)
print(ecco_dataset_C.data_vars)
```

```
Data variables:
    SSH      (time, tile, j, i) float32 dask.array<chunksize=(12, 4, 90, 90), meta=np.
↪ndarray>
Data variables:
    ADVx_TH  (time, tile, k, j, i_g) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
Data variables:
    ADVy_TH  (time, tile, k, j_g, i) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
```

1.11.3 Merging multiple Datasets from state estimate variables

Using the xarray method merge we can create a single Dataset with multiple DataArrays.

[11]:
```
# merge together
ecco_dataset_ABC = xr.merge([ecco_dataset_A, ecco_dataset_B, ecco_dataset_C]).load()
```

Examining the merged `Dataset`

As before, let's look at the contents of the new merged `Dataset`

[12]:
```
ecco_dataset_ABC
```

[12]:
```
<xarray.Dataset>
Dimensions:    (i: 90, i_g: 90, j: 90, j_g: 90, k: 50, nv: 2, tile: 13, time: 12)
Coordinates:
  * tile       (tile) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
  * j          (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * i          (i) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
    XC         (tile, j, i) float32 -111.60647 -111.303 -110.94285 ... nan nan
    YC         (tile, j, i) float32 -88.24259 -88.382515 -88.52242 ... nan nan
    rA         (tile, j, i) float32 362256450.0 363300960.0 ... nan nan
    time_bnds  (time, nv) datetime64[ns] 2010-01-01 2010-02-01 ... 2011-01-01
    iter       (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
  * time       (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
  * i_g        (i_g) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k          (k) int32 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
    Z          (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    PHrefC     (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
    drF        (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    dxC        (tile, j, i_g) float32 15583.418 15588.104 ... 23406.256
    rAw        (tile, j, i_g) float32 361699460.0 362790240.0 ... 364760350.0
    dyG        (tile, j, i_g) float32 23210.262 23273.26 ... 15595.26 15583.685
    hFacW      (tile, k, j, i_g) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
  * j_g        (j_g) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
    rAs        (tile, j_g, i) float32 179944260.0 180486990.0 ... 364150620.0
    dxG        (tile, j_g, i) float32 15584.907 15589.316 ... 23142.107
    dyC        (tile, j_g, i) float32 11563.718 11593.785 ... 15578.138
    hFacS      (tile, k, j_g, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
Dimensions without coordinates: nv
Data variables:
    SSH        (time, tile, j, i) float32 0.0 0.0 0.0 0.0 ... nan nan nan nan
    ADVx_TH    (time, tile, k, j, i_g) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0
    ADVy_TH    (time, tile, k, j_g, i) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0
```

1. Dimensions

`Dimensions: (i: 90, i_g: 90, j: 90, j_g: 90, k: 50, nv: 2, tile: 13, time: 12)`

ecco_dataset_merged is a container of `Data Arrays` and as such it lists all of the unique dimensions its `Data Arrays`. In other words, *Dimensions* shows all of the dimensions used by its variables.

2. Dimension Coordinates

```
Coordinates:
Coordinates:
  * j          (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * i          (i) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * tile       (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
  * time       (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
  * i_g        (i_g) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k          (k) int32 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * j_g        (j_g) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
```

Notice that the **tile** and **time** coordinates are unchanged. `merge` recognizes identical coordiantes and keeps them.

3. Non-Dimension Coordinates

```
Coordinates:
    XC         (tile, j, i) float32 -111.60647 -111.303 -110.94285 ... nan nan
    YC         (tile, j, i) float32 -88.24259 -88.382515 -88.52242 ... nan nan
    rA         (tile, j, i) float32 362256450.0 363300960.0 ... nan nan
    time_bnds  (time, nv) datetime64[ns] 2010-01-01 2010-02-01 ... 2011-01-01
    iter       (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
    Z          (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    PHrefC     (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
    drF        (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    dxC        (tile, j, i_g) float32 15583.418 15588.104 ... 23406.256
    rAw        (tile, j, i_g) float32 361699460.0 362790240.0 ... 364760350.0
    dyG        (tile, j, i_g) float32 23210.262 23273.26 ... 15595.26 15583.685
    hFacW      (tile, k, j, i_g) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    rAs        (tile, j_g, i) float32 179944260.0 180486990.0 ... 364150620.0
    dxG        (tile, j_g, i) float32 15584.907 15589.316 ... 23142.107
    dyC        (tile, j_g, i) float32 11563.718 11593.785 ... 15578.138
    hFacS      (tile, k, j_g, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
```

The list of non-dimension coordinates is now much longer. Like *Dimensions*, the non-dimension coordinates of the merged `Dataset` contain all of the non-dimension coordinates of the `Data Arrays`.

4. Attributes

Notice that all of the high level `Dataset` attributes are gone! Each of the three `Datasets` had different attributes and the `merge` routine simply drops them. The attributes of the *Data variables* remain intact:

[13]:
```
# (this json command makes Python dictionaries easier to read)
print(json.dumps(ecco_dataset_ABC.SSH.attrs, indent=2,sort_keys=True))
```

```
{
  "long_name": "Surface Height Anomaly adjusted with global steric height change and sea-
↪ice load",
  "standard_name": "sea_surface_height",
  "units": "m"
}
```

1.11.4 Adding the model grid `Dataset`

Let's use the `merge` routine to combine a `Dataset` of the model grid parameters with `output_merged`.

Load the model grid parameters

[14]:
```
# Load the llc90 grid parameters
grid_dir= ECCO_dir + '/nctiles_grid/'
grid_dataset = ecco.load_ecco_grid_nc(grid_dir, 'ECCOv4r3_grid.nc')
grid_dataset.coords
```

[14]:
```
Coordinates:
  * i          (i) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * i_g        (i_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j          (j) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j_g        (j_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k          (k) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_u        (k_u) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_l        (k_l) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_p1       (k_p1) int64 0 1 2 3 4 5 6 7 8 9 ... 42 43 44 45 46 47 48 49 50
  * tile       (tile) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
    XC         (tile, j, i) float32 -111.60647 -111.303 ... -111.86579
    YC         (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
    XG         (tile, j_g, i_g) float32 -115.0 -115.0 ... -102.928925 -108.95171
    YG         (tile, j_g, i_g) float32 -88.17569 -88.31587 ... -88.02409
    CS         (tile, j, i) float32 0.06157813 0.06675376 ... -0.9983638
    SN         (tile, j, i) float32 -0.99810225 -0.9977695 ... -0.057182025
    Z          (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    Zp1        (k_p1) float32 0.0 -10.0 -20.0 -30.0 ... -5244.5 -5678.0 -6134.5
    Zu         (k_u) float32 -10.0 -20.0 -30.0 -40.0 ... -5244.5 -5678.0 -6134.5
    Zl         (k_l) float32 0.0 -10.0 -20.0 -30.0 ... -4834.0 -5244.5 -5678.0
    rA         (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
    dxG        (tile, j_g, i) float32 15584.907 15589.316 ... 23142.107
    dyG        (tile, j, i_g) float32 23210.262 23273.26 ... 15595.26 15583.685
    Depth      (tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    rAz        (tile, j_g, i_g) float32 179944260.0 180486990.0 ... 364150620.0
    dxC        (tile, j, i_g) float32 15583.418 15588.104 ... 23406.256
    dyC        (tile, j_g, i) float32 11563.718 11593.785 ... 15578.138
    rAw        (tile, j, i_g) float32 361699460.0 362790240.0 ... 364760350.0
    rAs        (tile, j_g, i) float32 179944260.0 180486990.0 ... 364150620.0
    drC        (k_p1) float32 5.0 10.0 10.0 10.0 ... 399.0 422.0 445.0 228.25
    drF        (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    PHrefC     (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
    PHrefF     (k_p1) float32 0.0 98.1 196.2 ... 51448.547 55701.18 60179.445
    hFacC      (k, tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    hFacW      (k, tile, j, i_g) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    hFacS      (k, tile, j_g, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    maskC      (k, tile, j, i) bool False False False ... False False False
    maskW      (k, tile, j, i_g) bool False False False ... False False False
    maskS      (k, tile, j_g, i) bool False False False ... False False False
    maskCtrlW  (k, tile, j, i_g) bool False False False ... False False False
    maskCtrlS  (k, tile, j_g, i) bool False False False ... False False False
    maskCtrlC  (k, tile, j, i) bool False False False ... False False False
```

Merge `grid_all_tiles` with `output_merged`

[15]:
```
ecco_dataset_ABCG= xr.merge([ecco_dataset_ABC, grid_dataset])
ecco_dataset_ABCG
```

[15]:
```
<xarray.Dataset>
Dimensions:    (i: 90, i_g: 90, j: 90, j_g: 90, k: 50, k_l: 50, k_p1: 51, k_u: 50, nv: 2,
↪ tile: 13, time: 12)
Coordinates:
  * tile       (tile) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
  * j          (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * i          (i) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
    XC         (tile, j, i) float32 -111.60647 -111.303 ... -111.86579
    YC         (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
    rA         (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
    time_bnds  (time, nv) datetime64[ns] 2010-01-01 2010-02-01 ... 2011-01-01
    iter       (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
  * time       (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
  * i_g        (i_g) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k          (k) int32 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
    Z          (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    PHrefC     (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
    drF        (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    dxC        (tile, j, i_g) float32 15583.418 15588.104 ... 23406.256
    rAw        (tile, j, i_g) float32 361699460.0 362790240.0 ... 364760350.0
    dyG        (tile, j, i_g) float32 23210.262 23273.26 ... 15595.26 15583.685
    hFacW      (tile, k, j, i_g) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
  * j_g        (j_g) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
    rAs        (tile, j_g, i) float32 179944260.0 180486990.0 ... 364150620.0
    dxG        (tile, j_g, i) float32 15584.907 15589.316 ... 23142.107
    dyC        (tile, j_g, i) float32 11563.718 11593.785 ... 15578.138
    hFacS      (tile, k, j_g, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
  * k_u        (k_u) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_l        (k_l) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_p1       (k_p1) int64 0 1 2 3 4 5 6 7 8 9 ... 42 43 44 45 46 47 48 49 50
    XG         (tile, j_g, i_g) float32 -115.0 -115.0 ... -102.928925 -108.95171
    YG         (tile, j_g, i_g) float32 -88.17569 -88.31587 ... -88.02409
    CS         (tile, j, i) float32 0.06157813 0.06675376 ... -0.9983638
    SN         (tile, j, i) float32 -0.99810225 -0.9977695 ... -0.057182025
    Zp1        (k_p1) float32 0.0 -10.0 -20.0 -30.0 ... -5244.5 -5678.0 -6134.5
    Zu         (k_u) float32 -10.0 -20.0 -30.0 -40.0 ... -5244.5 -5678.0 -6134.5
    Zl         (k_l) float32 0.0 -10.0 -20.0 -30.0 ... -4834.0 -5244.5 -5678.0
    Depth      (tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    rAz        (tile, j_g, i_g) float32 179944260.0 180486990.0 ... 364150620.0
    drC        (k_p1) float32 5.0 10.0 10.0 10.0 ... 399.0 422.0 445.0 228.25
    PHrefF     (k_p1) float32 0.0 98.1 196.2 ... 51448.547 55701.18 60179.445
    hFacC      (k, tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    maskC      (k, tile, j, i) bool False False False ... False False False
    maskW      (k, tile, j, i_g) bool False False False ... False False False
    maskS      (k, tile, j_g, i) bool False False False ... False False False
    maskCtrlW  (k, tile, j, i_g) bool False False False ... False False False
    maskCtrlS  (k, tile, j_g, i) bool False False False ... False False False
    maskCtrlC  (k, tile, j, i) bool False False False ... False False False
```

(continues on next page)

(continued from previous page)

```
Dimensions without coordinates: nv
Data variables:
    SSH        (time, tile, j, i) float32 0.0 0.0 0.0 0.0 ... nan nan nan nan
    ADVx_TH    (time, tile, k, j, i_g) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0
    ADVy_TH    (time, tile, k, j_g, i) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0
```

Examining the merged `Dataset`

The result of this last merge is a single `Dataset` with 3 *Data variables*, and a complete set of model grid parameters (distances and areas).

1.11.5 Merging and memory

Merging `Datasets` together does not make copies of the data in memory. Instead, merged `Datasets` are in fact just a reorganized collection of pointers. You may want to delete the original variables to clear your namespace, but it is not necessary.

1.11.6 Summary

Now you know how to merge multiple `Datasets` using the `merge` command. We demonstrated merging of `Datasets` constructed from three different variables types and the model grid parameters.

1.12 Saving `Datasets` and `DataArrays` to NetCDF

1.12.1 Objectives

Introduce an easy method for saving `Datasets` and `DataArrays` objects to NetCDF

1.12.2 Introduction

Saving your `Datasets` and `DataArrays` objects to NetCDF files couldn't be simpler. The `xarray` module that we've been using to load NetCDF files provides methods for saving your `Datasets` and `DataArrays` as NetCDF files.

Here is the manual page on the subjet: http://xarray.pydata.org/en/stable/generated/xarray.Dataset.to_netcdf.html

The method `._to_netcdf( )` is available to **both** `Datasets` and `DataArrays` objects. So useful!

Syntax

```
your_dataset.to_netcdf('/your_filepath/your_netcdf_filename.nc')
```

1.12.3 Saving an existing `Dataset` to NetCDF

First, let's set up the environment and load a `Dataset`

[1]:
```
import numpy as np
import xarray as xr
import sys
import matplotlib.pyplot as plt
import json
import sys
```

[2]:
```
## Import the ecco_v4_py library into Python
## =========================================

#import ecco_v4_py as ecco

## -- If ecco_v4_py is not installed in your local Python library,
##    tell Python where to find it.  For example, if your ecco_v4_py
##    files are in /Users/ifenty/ECCOv4-py/ecco_v4_py, then use:
sys.path.append('/home/ifenty/ECCOv4-py')
import ecco_v4_py as ecco
```

load a single tile, monthly averaged *THETA* for March 2010 for model tile 2.

[3]:
```
## Set top-level file directory for the ECCO NetCDF files
## =====================================================================
# base_dir = '/home/username/'
base_dir = '/home/ifenty/ECCOv4-release'

## define a high-level directory for ECCO fields
ECCO_dir = base_dir + '/Release3_alt'
```

[4]:
```
## LOAD NETCDF FILE
## ================

# directory of the file
data_dir= ECCO_dir + '/nctiles_monthly/THETA/'

# filename
fname = 'THETA_2010.nc'

# load the dataset file using xarray
theta_dataset = xr.open_dataset(data_dir + fname).load()
```

Now that we've loaded *theta_dataset*, let's save it in the **/tmp** file directory with a new name.

[5]:
```
new_filename_1 = './test_output.nc'
print ('saving to ', new_filename_1)
theta_dataset.to_netcdf(path=new_filename_1)
print ('finished saving')
```

```
saving to  ./test_output.nc
finished saving
```

It's really that simple!

1.12.4 Saving a new custom `Dataset` to NetCDF

Now let's create a new custom `Dataset` that with *THETA*, *SSH* and model grid parameter variables for a few tiles and depth level 10.

[6]:
```
data_dir= ECCO_dir + '/nctiles_monthly/'

SSH_THETA_201003 = \
    ecco.recursive_load_ecco_var_from_years_nc(data_dir, \
                                               ['SSH', 'THETA'], \
                                               tiles_to_load = [0,1,2],
                                               years_to_load = 2010)
grid_dir = ECCO_dir + '/nctiles_grid/'
grid = ecco.load_ecco_grid_nc(grid_dir, 'ECCOv4r3_grid.nc')
grid.close()

custom_dataset = xr.merge([SSH_THETA_201003, grid])
```

```
loading files of  THETA
loading files of  SSH
```

and now we can easily save it:

[7]:
```
new_filename_2 = './test_output_2.nc'
print ('saving to ', new_filename_2)
custom_dataset.to_netcdf(path=new_filename_2)
custom_dataset.close()
print ('finished saving')
```

```
saving to  ./test_output_2.nc
finished saving
```

[8]:
```
custom_dataset
```

[8]:
```
<xarray.Dataset>
Dimensions:    (i: 90, i_g: 90, j: 90, j_g: 90, k: 50, k_l: 50, k_p1: 51, k_u: 50, nv: 2,
↪ tile: 13, time: 12)
Coordinates:
  * tile       (tile) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
  * j          (j) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * i          (i) int32 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k          (k) int32 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
    Z          (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    PHrefC     (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
    drF        (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    XC         (tile, j, i) float32 -111.60647 -111.303 ... -111.86579
    YC         (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
    rA         (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
    hFacC      (tile, k, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    time_bnds  (time, nv) datetime64[ns] dask.array<chunksize=(12, 2), meta=np.ndarray>
    iter       (time) int32 dask.array<chunksize=(12,), meta=np.ndarray>
  * time       (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
  * i_g        (i_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j_g        (j_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k_u        (k_u) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
```

(continues on next page)

(continued from previous page)

```
  * k_l         (k_l) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_p1        (k_p1) int64 0 1 2 3 4 5 6 7 8 9 ... 42 43 44 45 46 47 48 49 50
    XG          (tile, j_g, i_g) float32 -115.0 -115.0 ... -102.928925 -108.95171
    YG          (tile, j_g, i_g) float32 -88.17569 -88.31587 ... -88.02409
    CS          (tile, j, i) float32 0.06157813 0.06675376 ... -0.9983638
    SN          (tile, j, i) float32 -0.99810225 -0.9977695 ... -0.057182025
    Zp1         (k_p1) float32 0.0 -10.0 -20.0 -30.0 ... -5244.5 -5678.0 -6134.5
    Zu          (k_u) float32 -10.0 -20.0 -30.0 -40.0 ... -5244.5 -5678.0 -6134.5
    Zl          (k_l) float32 0.0 -10.0 -20.0 -30.0 ... -4834.0 -5244.5 -5678.0
    dxG         (tile, j_g, i) float32 15584.907 15589.316 ... 23142.107
    dyG         (tile, j, i_g) float32 23210.262 23273.26 ... 15595.26 15583.685
    Depth       (tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    rAz         (tile, j_g, i_g) float32 179944260.0 180486990.0 ... 364150620.0
    dxC         (tile, j, i_g) float32 15583.418 15588.104 ... 23406.256
    dyC         (tile, j_g, i) float32 11563.718 11593.785 ... 15578.138
    rAw         (tile, j, i_g) float32 361699460.0 362790240.0 ... 364760350.0
    rAs         (tile, j_g, i) float32 179944260.0 180486990.0 ... 364150620.0
    drC         (k_p1) float32 5.0 10.0 10.0 10.0 ... 399.0 422.0 445.0 228.25
    PHrefF      (k_p1) float32 0.0 98.1 196.2 ... 51448.547 55701.18 60179.445
    hFacW       (k, tile, j, i_g) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    hFacS       (k, tile, j_g, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    maskC       (k, tile, j, i) bool False False False ... False False False
    maskW       (k, tile, j, i_g) bool False False False ... False False False
    maskS       (k, tile, j_g, i) bool False False False ... False False False
    maskCtrlW   (k, tile, j, i_g) bool False False False ... False False False
    maskCtrlS   (k, tile, j_g, i) bool False False False ... False False False
    maskCtrlC   (k, tile, j, i) bool False False False ... False False False
Dimensions without coordinates: nv
Data variables:
    THETA       (time, tile, k, j, i) float32 dask.array<chunksize=(12, 13, 50, 90, 90),␣
↪meta=np.ndarray>
    SSH         (time, tile, j, i) float32 dask.array<chunksize=(12, 13, 90, 90), meta=np.
↪ndarray>
```

1.12.5 Verifying our new NetCDF files

To verify that `to_netcdf()` worked, load them and compare with the originals.

Compare *theta_dataset* with *dataset_1*

[9]:
```
# the first test dataset
dataset_1 = xr.open_dataset(new_filename_1)

# release the file handle (not necessary but generally a good idea)
dataset_1.close()
```

The `np.allclose` method does element-by-element comparison of variables

[10]:
```
# loop through the data variables in dataset_1
for key in dataset_1.keys():
```

(continues on next page)

(continued from previous page)

```
    print ('checking %s ' % key)
    print ('-- identical in dataset_1 and theta_dataset : %s' % \
           np.allclose(dataset_1[key], theta_dataset[key], equal_nan=True))

# note: ``equal_nan`` means nan==nan (default nan != nan)
```

```
checking THETA
-- identical in dataset_1 and theta_dataset : True
```

THETA is the same in both datasets.

Compare *custom_dataset* with *dataset_2*

[11]:
```
# our custom dataset
dataset_2 = xr.open_dataset(new_filename_2)
dataset_2.close()
print ('finished loading')
```

```
finished loading
```

[12]:
```
for key in dataset_2.keys():
    print ('checking %s ' % key)
    print ('-- identical in dataset_2 and custom_dataset : %s'\
           % np.allclose(dataset_2[key], custom_dataset[key], equal_nan=True))
```

```
checking THETA
-- identical in dataset_2 and custom_dataset : True
checking SSH
-- identical in dataset_2 and custom_dataset : True
```

THETA and *SSH* are the same in both datasets.

So nice to hear!

1.12.6 Saving the results of calculations

Calculations in the form of `DataArrays`

Often we would like to store the results of our calculations to disk. If your operations are made at the level of `DataArray` objects (and not the lower `ndarray` level) then you can use these same methods to save your results. All of the coordinates will be preserved (although attributes be lost). Let's demonstrate by making a dummy calculation on SSH

$$SSH_{sq}(i) = SSH(i)^2$$

[13]:
```
SSH_sq = custom_dataset.SSH * custom_dataset.SSH

SSH_sq
```

[13]:
```
<xarray.DataArray 'SSH' (time: 12, tile: 13, j: 90, i: 90)>
dask.array<mul, shape=(12, 13, 90, 90), dtype=float32, chunksize=(12, 13, 90, 90),␣
```

(continues on next page)

(continued from previous page)

```
↪chunktype=numpy.ndarray>
Coordinates:
  * tile     (tile) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
  * j        (j) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * i        (i) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
    XC       (tile, j, i) float32 -111.60647 -111.303 ... -105.58465 -111.86579
    YC       (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
    rA       (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
    iter     (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
  * time     (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
    CS       (tile, j, i) float32 0.06157813 0.06675376 ... -0.9983638
    SN       (tile, j, i) float32 -0.99810225 -0.9977695 ... -0.057182025
    Depth    (tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
```

SSH_sq is itself a `DataArray`.

Before saving, let's give our new *SSH_sq* variable a better name and descriptive attributes.

[14]:
```
SSH_sq.name = 'SSH^2'
SSH_sq.attrs['long_name'] = 'Square of Surface Height Anomaly'
SSH_sq.attrs['units'] = 'm^2'

# Let's see the result
SSH_sq
```

[14]:
```
<xarray.DataArray 'SSH^2' (time: 12, tile: 13, j: 90, i: 90)>
dask.array<mul, shape=(12, 13, 90, 90), dtype=float32, chunksize=(12, 13, 90, 90),␣
↪chunktype=numpy.ndarray>
Coordinates:
  * tile     (tile) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
  * j        (j) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * i        (i) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
    XC       (tile, j, i) float32 -111.60647 -111.303 ... -105.58465 -111.86579
    YC       (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
    rA       (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
    iter     (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
  * time     (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
    CS       (tile, j, i) float32 0.06157813 0.06675376 ... -0.9983638
    SN       (tile, j, i) float32 -0.99810225 -0.9977695 ... -0.057182025
    Depth    (tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
Attributes:
    long_name:  Square of Surface Height Anomaly
    units:      m^2
```

much better! Now we'll save.

[15]:
```
new_filename_3 = './ssh_sq_DataArray.nc'
print ('saving to ', new_filename_3)

SSH_sq.to_netcdf(path=new_filename_3)
print ('finished saving')
```

```
saving to  ./ssh_sq_DataArray.nc
finished saving
```

Calculations in the form of `numpy ndarrays`

If calculations are made at the `ndarray` level then the results will also be `ndarrays`.

[16]:
```
SSH_dummy_ndarray = custom_dataset.SSH.values *  custom_dataset.SSH.values

type(SSH_dummy_ndarray)
```

[16]:
```
numpy.ndarray
```

You'll need to use different methods to save these results to NetCDF files, one of which is described here: http://pyhogs.github.io/intro_netcdf4.html

1.12.7 Summary

Saving `Datasets` and `DataArrays` to disk as NetCDF files is fun and easy with `xarray`!

1.13 Accessing and Subsetting Variables

1.13.1 Objectives

Introduce methods for **accessing** and **subsetting** the variables stored in ECCO `Datasets` and `DataArrays`.

1.13.2 Accessing fields inside `Dataset` and `DataArray` objects

There are two methods for accessing variables stored in `DataArray` and `Dataset` objects, the "dot" method and the "dictionary" method. The syntax of these methods is as follows:

1. The "dot" method: e.g. ,`X.Y`
2. The "dictionary" method: e.g., `Y['Y']`

Both methods work identically to access *Dimensions*, *Coordinates*, and *Data variables*. Accessing *Attribute* variables requires a slightly different approach as we will see.

Before we can demonstrate these methods, first create a `Dataset`:

[1]:
```
import numpy as np
import xarray as xr
import sys
import matplotlib.pyplot as plt
%matplotlib inline
import json

import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
## Import the ecco_v4_py library into Python
## =========================================
## -- If ecco_v4_py is not installed in your local Python library,
##    tell Python where to find it.  For example, if your ecco_v4_py
##    files are in /Users/ifenty/ECCOv4-py/ecco_v4_py, then use:
```

(continues on next page)

(continued from previous page)

```
sys.path.append('/home/ifenty/ECCOv4-py')
import ecco_v4_py as ecco
```

[3]:
```
## Set top-level file directory for the ECCO NetCDF files
## =============================================================
# base_dir = '/home/username/'
base_dir = '/home/ifenty/ECCOv4-release'

## define a high-level directory for ECCO fields
ECCO_dir = base_dir + '/Release3_alt'
```

[4]:
```
## Load the model grid
grid_dir= ECCO_dir + '/nctiles_grid/'

ecco_grid = ecco.load_ecco_grid_nc(grid_dir, 'ECCOv4r3_grid.nc')

## Load one year of SSH and OBP
day_mean_dir= ECCO_dir + '/nctiles_monthly/'

ecco_vars = \
    ecco.recursive_load_ecco_var_from_years_nc(day_mean_dir, \
                                               vars_to_load=['SSH','OBP'], \
                                               years_to_load=2010,\
                                               dask_chunk=False)

## Merge the ecco_grid with the ecco_vars to make the ecco_ds
ecco_ds = xr.merge((ecco_grid , ecco_vars))
```

```
loading files of  SSH
loading files of  OBP
```

[5]:
```
ecco_ds.data_vars
```

[5]:
```
Data variables:
    SSH      (time, tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    OBP      (time, tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
```

[6]:
```
ecco_ds.coords
```

[6]:
```
Coordinates:
  * i          (i) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * i_g        (i_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j          (j) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * j_g        (j_g) int64 0 1 2 3 4 5 6 7 8 9 ... 80 81 82 83 84 85 86 87 88 89
  * k          (k) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_u        (k_u) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_l        (k_l) int64 0 1 2 3 4 5 6 7 8 9 ... 40 41 42 43 44 45 46 47 48 49
  * k_p1       (k_p1) int64 0 1 2 3 4 5 6 7 8 9 ... 42 43 44 45 46 47 48 49 50
  * tile       (tile) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
    XC         (tile, j, i) float32 -111.60647 -111.303 ... -111.86579
    YC         (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
```

(continues on next page)

(continued from previous page)

```
    XG          (tile, j_g, i_g) float32 -115.0 -115.0 ... -102.928925 -108.95171
    YG          (tile, j_g, i_g) float32 -88.17569 -88.31587 ... -88.02409
    CS          (tile, j, i) float32 0.06157813 0.06675376 ... -0.9983638
    SN          (tile, j, i) float32 -0.99810225 -0.9977695 ... -0.057182025
    Z           (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    Zp1         (k_p1) float32 0.0 -10.0 -20.0 -30.0 ... -5244.5 -5678.0 -6134.5
    Zu          (k_u) float32 -10.0 -20.0 -30.0 -40.0 ... -5244.5 -5678.0 -6134.5
    Zl          (k_l) float32 0.0 -10.0 -20.0 -30.0 ... -4834.0 -5244.5 -5678.0
    rA          (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
    dxG         (tile, j_g, i) float32 15584.907 15589.316 ... 23142.107
    dyG         (tile, j, i_g) float32 23210.262 23273.26 ... 15595.26 15583.685
    Depth       (tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    rAz         (tile, j_g, i_g) float32 179944260.0 180486990.0 ... 364150620.0
    dxC         (tile, j, i_g) float32 15583.418 15588.104 ... 23406.256
    dyC         (tile, j_g, i) float32 11563.718 11593.785 ... 15578.138
    rAw         (tile, j, i_g) float32 361699460.0 362790240.0 ... 364760350.0
    rAs         (tile, j_g, i) float32 179944260.0 180486990.0 ... 364150620.0
    drC         (k_p1) float32 5.0 10.0 10.0 10.0 ... 399.0 422.0 445.0 228.25
    drF         (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    PHrefC      (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
    PHrefF      (k_p1) float32 0.0 98.1 196.2 ... 51448.547 55701.18 60179.445
    hFacC       (k, tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    hFacW       (k, tile, j, i_g) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    hFacS       (k, tile, j_g, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    maskC       (k, tile, j, i) bool False False False ... False False False
    maskW       (k, tile, j, i_g) bool False False False ... False False False
    maskS       (k, tile, j_g, i) bool False False False ... False False False
    maskCtrlW   (k, tile, j, i_g) bool False False False ... False False False
    maskCtrlS   (k, tile, j_g, i) bool False False False ... False False False
    maskCtrlC   (k, tile, j, i) bool False False False ... False False False
    time_bnds   (time, nv) datetime64[ns] 2010-01-01 2010-02-01 ... 2011-01-01
    iter        (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
  * time        (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
```

Accessing Data Variables

Now we'll use the two methods to access the SSH DataArray,

[7]:
```
## The Dot Method
ssh_A = ecco_ds.SSH

## The Dictionary Method
ssh_B = ecco_ds['SSH']
```

[8]:
```
print (type(ssh_A))
print (type(ssh_B))
```

```
<class 'xarray.core.dataarray.DataArray'>
<class 'xarray.core.dataarray.DataArray'>
```

We access the numpy arrays stored in these DataArrays with the "dot" method on values.

[9]:
```
ssh_arr = ssh_A.values
print (type(ssh_arr))
```

```
<class 'numpy.ndarray'>
```

The numpy array storing the data

The shape of the numpy array can be found by invoking its .shape

[10]:
```
ssh_arr.shape
```

[10]:
```
(12, 13, 90, 90)
```

The order of these four dimensions is consistent with their ordering in the original DataArray, ~~~ time, tile, j, i ~~~

[11]:
```
ssh_A.dims
```

[11]:
```
('time', 'tile', 'j', 'i')
```

ssh_A and ssh_B are new variables but they are not **copies** of the original SSH DataArray object, they both point to the original numpy array.

We can confirm that ssh_A and ssh_B both refer to the same array in memory using the Python allclose command (and ignoring nans)

[12]:
```
print(np.allclose(ssh_A, ssh_B, equal_nan=True))
```

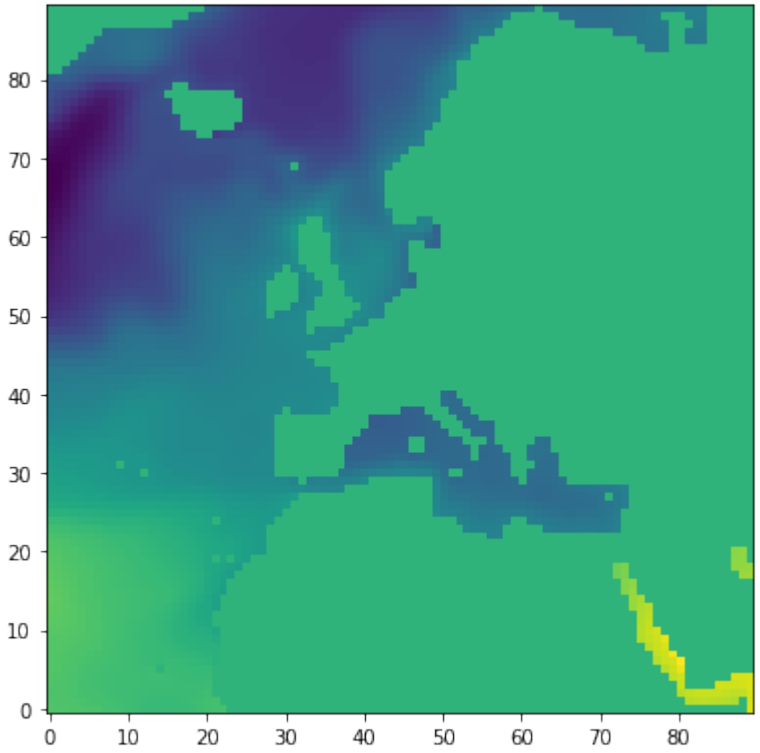
```
True
```

Accessing the numpy array

We have *ssh_arr*, a 4D numpy array (ndarray). Let's take out a single 2D slice of it, the first time record, and the second tile:

[13]:
```
fig=plt.figure(figsize=(8, 6.5))

# plot SSH for time =0, tile = 2 by using ``imshow`` on the ``numpy array``
plt.imshow(ssh_arr[0,2,:,:], origin='lower')
```

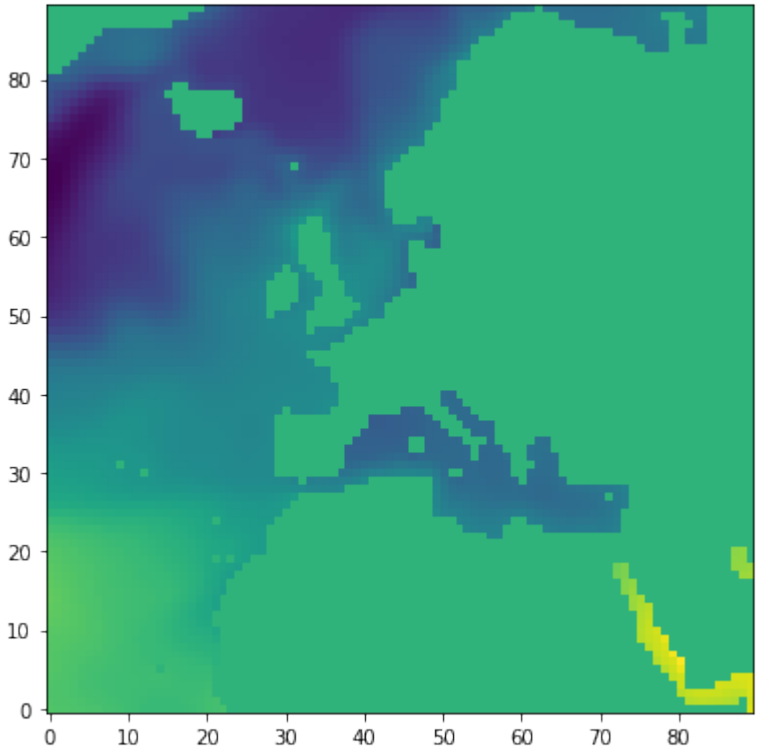
[13]:
```
<matplotlib.image.AxesImage at 0x7f71ae07cac8>
```



We could accessing the array using the "dot" method on *ecco_ds* to first get *SSH* and then use the "dot" method to access the `numpy` array through *values*:

[14]:
```
fig=plt.figure(figsize=(8, 6.5))
plt.imshow(ecco_ds.SSH.values[0,2,:,:], origin='lower')
```

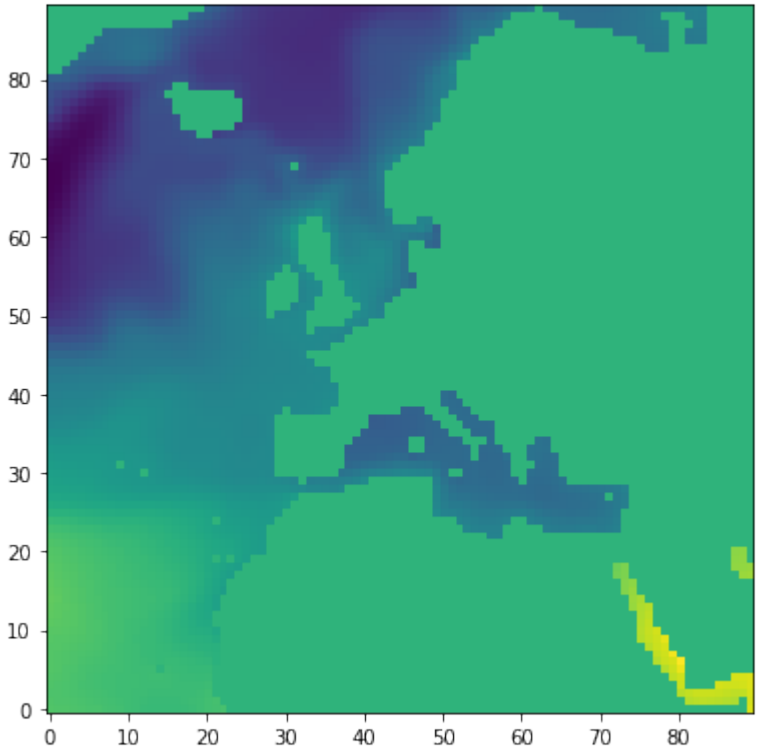
[14]:
```
<matplotlib.image.AxesImage at 0x7f71adfa7438>
```



Or could use the "dictionary" method on *ecco_ds* to get *SSH* then the "dot" method to access the `numpy` array through *values*:

[15]:
```
fig=plt.figure(figsize=(8, 6.5))
plt.imshow(ecco_ds['SSH'].values[0,2,:,:], origin='lower')
```

[15]:
```
<matplotlib.image.AxesImage at 0x7f71adf1e2e8>
```



These are equivalent methods.

Accessing Coordinates

Accessing Coordinates is exactly the same as accessing *Data variables*. Use the "dot" or "dictionary" methods

[16]:
```
xc = ecco_ds['XC']
time = ecco_ds['time']

print(type(xc))
print(type(time))
```

```
<class 'xarray.core.dataarray.DataArray'>
<class 'xarray.core.dataarray.DataArray'>
```

As *xc* is a `DataArray`, we can access the values in its `numpy` array through `.values`

[17]:
```
xc.values
```

[17]:
```
array([[[-111.60647 , -111.303   , -110.94285 , ...,   64.791115,
           64.80521 ,   64.81917 ],
        [-104.8196  , -103.928444, -102.87706 , ...,   64.36745 ,
           64.41012 ,   64.4524  ],
        [ -98.198784,  -96.788055,  -95.14185 , ...,   63.936497,
           64.008224,   64.0793  ],
        ...,
        [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
```

(continues on next page)

(continued from previous page)

```
        [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
         [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ]],

       [[ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
         [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
         [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
         ...,
         [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
         [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
         [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ]],

       [[ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
         [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
         [ -37.5     ,  -36.5     ,  -35.5     , ...,   49.5     ,
           50.5     ,   51.5     ],
         ...,
         [ -37.730072,  -37.17829 ,  -36.597565, ...,   50.597565,
           51.17829 ,   51.730072],
         [ -37.771988,  -37.291943,  -36.764027, ...,   50.764027,
           51.291943,   51.771988],
         [ -37.837925,  -37.44421 ,  -36.968143, ...,   50.968143,
           51.44421 ,   51.837925]],

       ...,

       [[-127.83792 , -127.77199 , -127.73007 , ..., -127.5     ,
          -127.5     , -127.5     ],
         [-127.44421 , -127.29195 , -127.17829 , ..., -126.5     ,
          -126.5     , -126.5     ],
         [-126.96814 , -126.76402 , -126.597565, ..., -125.5     ,
          -125.5     , -125.5     ],
         ...,
         [ -39.031857,  -39.235973,  -39.402435, ...,  -40.5     ,
           -40.5     ,  -40.5     ],
         [ -38.55579 ,  -38.708057,  -38.82171 , ...,  -39.5     ,
           -39.5     ,  -39.5     ],
         [ -38.162075,  -38.228012,  -38.269928, ...,  -38.5     ,
           -38.5     ,  -38.5     ]],

       [[-127.5     , -127.5     , -127.5     , ..., -127.5     ,
          -127.5     , -127.5     ],
         [-126.5     , -126.5     , -126.5     , ..., -126.5     ,
```

(continues on next page)

(continued from previous page)

```
         -126.5     , -126.5     ],
        [-125.5     , -125.5     , -125.5     , ..., -125.5     ,
         -125.5     , -125.5     ],
        ...,
        [ -40.5     ,  -40.5     ,  -40.5     , ...,  -40.5     ,
          -40.5     ,  -40.5     ],
        [ -39.5     ,  -39.5     ,  -39.5     , ...,  -39.5     ,
          -39.5     ,  -39.5     ],
        [ -38.5     ,  -38.5     ,  -38.5     , ...,  -38.5     ,
          -38.5     ,  -38.5     ]],

       [[-127.5     , -127.5     , -127.5     , ..., -115.850204,
         -115.50567 , -115.166985],
        [-126.5     , -126.5     , -126.5     , ..., -115.78025 ,
         -115.464066, -115.153244],
        [-125.5     , -125.5     , -125.5     , ..., -115.71079 ,
         -115.42275 , -115.139595],
        ...,
        [ -40.5     ,  -40.5     ,  -40.5     , ..., -101.42989 ,
         -106.83081 , -112.28605 ],
        [ -39.5     ,  -39.5     ,  -39.5     , ..., -100.48844 ,
         -106.24874 , -112.090065],
        [ -38.5     ,  -38.5     ,  -38.5     , ...,  -99.42048 ,
         -105.58465 , -111.86579 ]]], dtype=float32)
```

The shape of the *xc* is 13 x 90 x 90. Unlike *SSH* there is no time dimension.

[18]:
```
xc.values.shape
```

[18]:
```
(13, 90, 90)
```

Accessing Attributes

To access *Attribute* fields you you can use the dot method directly on the `Dataset` or `DataArray` or the dictionary method on the `attrs` field.

To demonstrate: we access the `units` attribute on *OBP* using both methods

[19]:
```
ecco_ds.OBP.units
```

[19]:
```
'm'
```

[20]:
```
ecco_ds.OBP.attrs['units']
```

[20]:
```
'm'
```

1.13.3 Subsetting variables using the `[]`, `sel`, `isel`, and `where` syntaxes

So far, a considerable amount of attention has been placed on the *Coordinates* of `Dataset` and `DataArray` objects. Why? Labeled coordinates are certainly not necessary for calculations on the basic numerical arrays that store the ECCO state estimate fields. The reason so much attention has been placed on coordinates is because the `xarray` offers several very useful methods for selecting (or indexing) subsets of data.

For more details of these *indexing* methods, please see the excellent `xarray` documentation: http://xarray.pydata.org/en/stable/indexing.html

Subsetting `numpy` arrays using the [ ] syntax

Subsetting `numpy` arrays is simple with the standard Python **[ ]** syntax.

To demonstrate, we'll pull out the `numpy` array of *SSH* and then select the first time record and second tile

> **Note:** `numpy` array indexing starts with 0*

[21]:
```
ssh_arr = ecco_ds.SSH.values
type(ssh_arr)
ssh_arr.shape
```

[21]:
```
(12, 13, 90, 90)
```

We know the order of the dimensions of SSH is [time, tile, j, i], so :

[22]:
```
ssh_time_0_tile_2 = ssh_arr[0,2,:,:]
```

ssh_time_0_tile_2 is also a numpy array, but now it is a 2D array:

[23]:
```
ssh_time_0_tile_2.shape
```

[23]:
```
(90, 90)
```

We know that the *time* coordinate on *ecco_ds* has one dimension, **time**, so we can use the `[]` syntax to find the first element of that array as well:

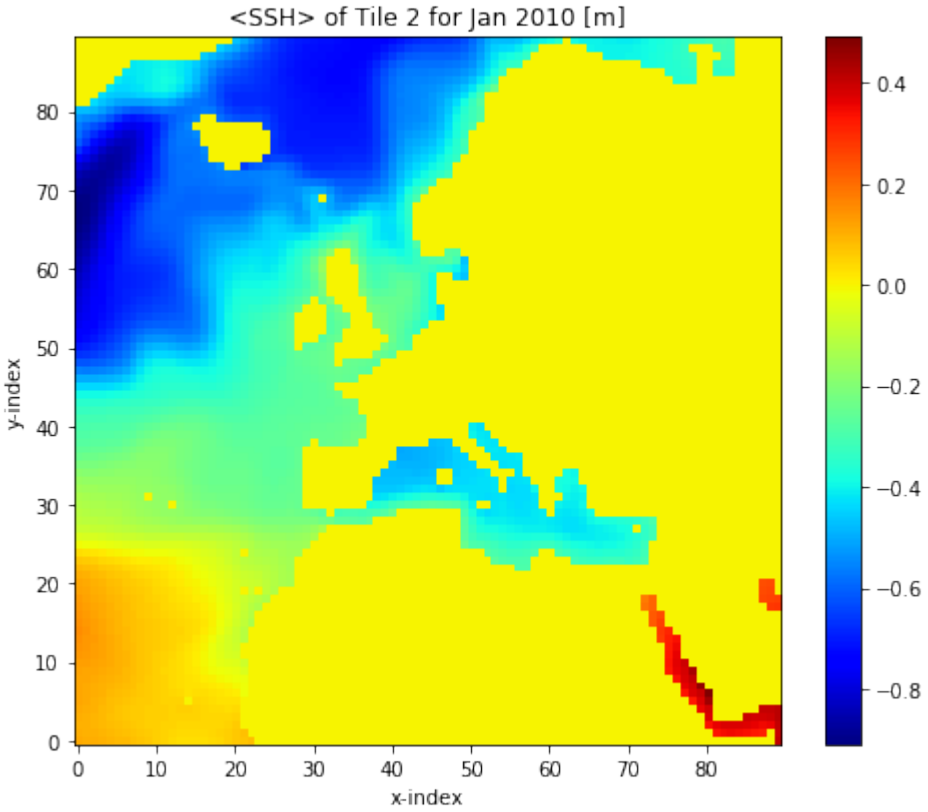
[24]:
```
print(ecco_ds.time.dims)

# the first time record is
print(ecco_ds.time.values[0])
```

```
('time',)
2010-01-16T12:00:00.000000000
```

which confirms that the first time record corresponds with Jan 2010

Now that we have a 2D array in *ssh_time_0_tile_2*, let's plot it

[25]:
```
fig=plt.figure(figsize=(8, 6.5))
plt.imshow(ssh_time_0_tile_2, origin='lower',cmap='jet')
plt.colorbar()
plt.title('<SSH> of Tile 2 for Jan 2010 [m]');
plt.xlabel('x-index');
plt.ylabel('y-index');
```



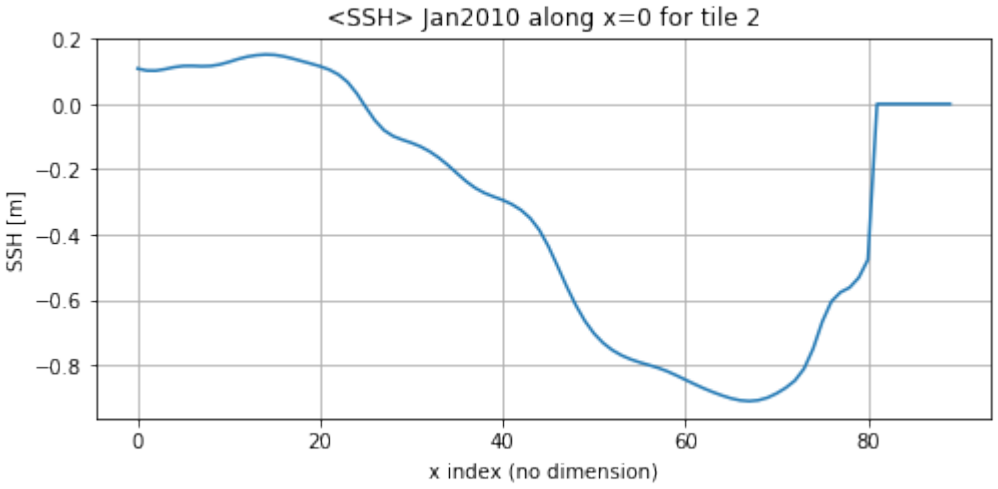
By eye we see large negative values around x=0, y=70 (i=0, j=70). Let's confirm:

[26]:
```
# remember the order of the array is [tile, j, i]
ssh_time_0_tile_2[70,0]
```

[26]:
```
-0.88685966
```

Let's plot *SSH* in this tile from y=0 to y=89 along x=0 (from the subtropical gyre into the subpolar gyre)

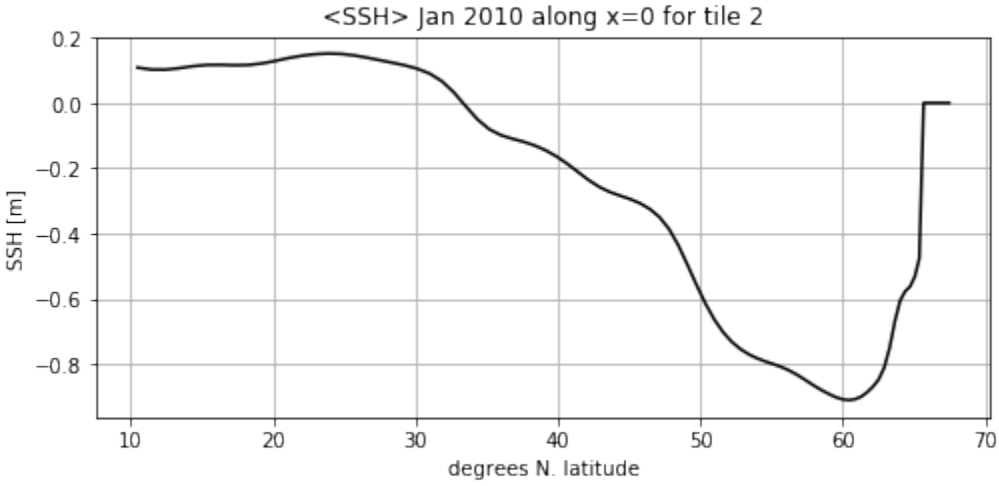
[27]:
```
fig=plt.figure(figsize=(8, 3.5))

plt.plot(ssh_time_0_tile_2[:,0])
plt.title('<SSH> Jan2010 along x=0 for tile 2')
plt.ylabel('SSH [m]');plt.xlabel('x index (no dimension)');
plt.grid()
```



We see that around y=80, SSH=0 because from around y=80 on we are on Greenland.

A more interesting plot might be to have the model latitude on the **x** axis instead of x-index:

[28]:
```
fig=plt.figure(figsize=(8, 3.5))

yc_tile_2 = ecco_ds.YC.values[2,:,:]

plt.plot(yc_tile_2[:,0],ssh_time_0_tile_2[:,0], color='k')
plt.title('<SSH> Jan 2010 along x=0 for tile 2')
plt.ylabel('SSH [m]');plt.xlabel('degrees N. latitude');
plt.grid()
```



We can always use **[ ]** method to subset our `numpy` arrays. It is a simple, direct method for accessing our fields. Just for fun let's plot this SSH subset:

Subsetting DataArrays using the [ ] syntax

An interesting and useful alternative to subsetting numpy arrays with the **[ ]** method is to subset DataArray instead:

[29]:
```
ssh_time_0_tile_2_DA = ecco_ds.SSH[0,2,:,:]
ssh_time_0_tile_2_DA
```

[29]:
```
<xarray.DataArray 'SSH' (j: 90, i: 90)>
array([[0.10849833, 0.10763063, 0.10525029, ..., 0.        , 0.        ,
        0.41118386],
       [0.10236199, 0.1005336 , 0.09722137, ..., 0.31839028, 0.        ,
        0.40181533],
       [0.10205435, 0.09833906, 0.09325342, ..., 0.353757  , 0.35896647,
        0.40689626],
       ...,
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ]], dtype=float32)
Coordinates:
  * i        (i) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * j        (j) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
    tile     int64 2
    XC       (j, i) float32 -37.5 -36.5 -35.5 ... 50.968143 51.44421 51.837925
    YC       (j, i) float32 10.458642 10.458642 10.458642 ... 67.53387 67.47211
    CS       (j, i) float32 1.0 1.0 1.0 1.0 ... 0.8916124 0.9051672 0.9424238
    SN       (j, i) float32 -1.8064405e-15 9.046443e-16 ... -0.33442083
    rA       (j, i) float32 11896091000.0 11896091000.0 ... 212633870.0
    Depth    (j, i) float32 4658.681 4820.5703 5014.177 ... 0.0 0.0 0.0
    iter     int32 158532
    time     datetime64[ns] 2010-01-16T12:00:00
Attributes:
    units:          m
    long_name:      Surface Height Anomaly adjusted with global steric height...
    standard_name:  sea_surface_height
```

The resulting DataArray is a subset of the original DataArray. The subset has two fewer dimensions (**tile** and **time** have been eliminated). The horizontal dimensions **j** and **i** are unchanged.

Even though the **tile** and **time** dimensions have been eliminated, the dimensional and non-dimensional coordinates associated with **time** and **tile** remain. In fact, these coordinates *tell us when in time and which tile our subset comes from*:

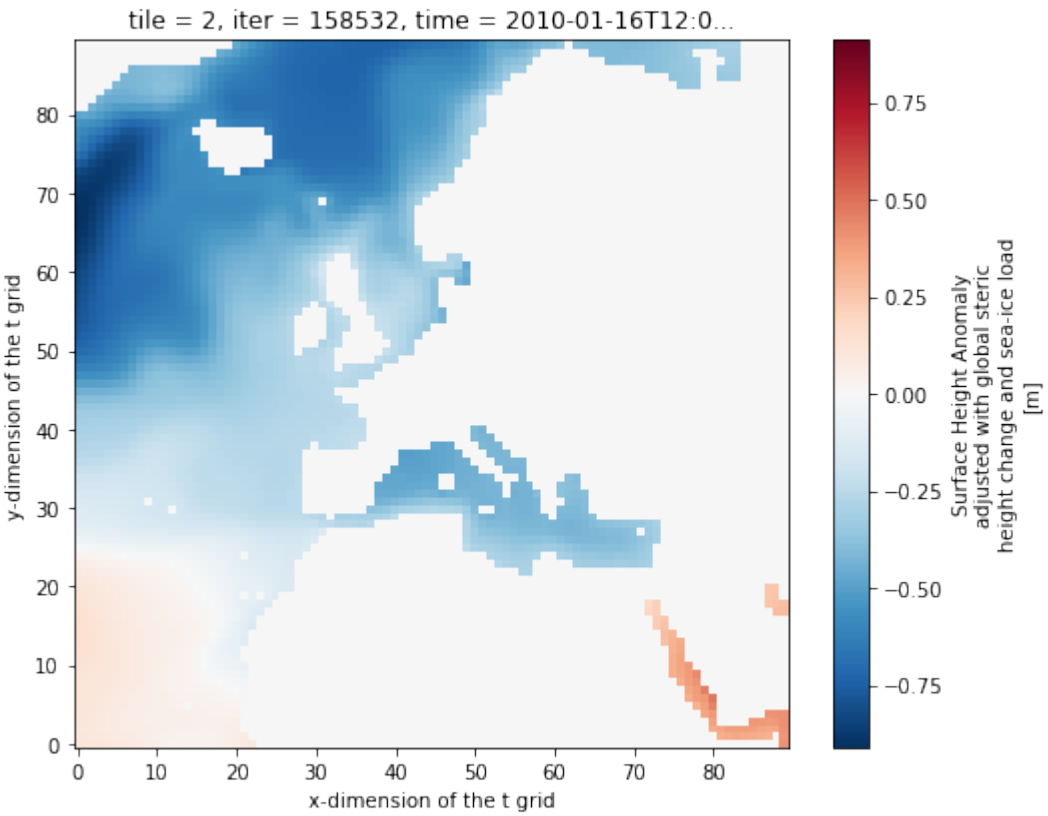
```
Coordinates:
  * j        (j) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * i        (i) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
    tile     int64 2
    XC       (j, i) float32 -37.5 -36.5 -35.5 ... 50.968143 51.44421 51.837925
    YC       (j, i) float32 10.458642 10.458642 10.458642 ... 67.53387 67.47211
    Depth    (j, i) float32 4658.681 4820.5703 5014.177 ... 0.0 0.0 0.0
    rA       (j, i) float32 11896091000.0 11896091000.0 ... 212633870.0
    iter     int32 158532
    time     datetime64[ns] 2010-01-16T12:00:00
```

Notice that the *tile* coordinate is 2 and the *time* coordinate is Jan 16, 2010, the middle of Jan.

As a `DataArray` we can take full advantage of the built-in plotting functionality of `xarray`. This functionality, which we've seen a few times, automatically labels the figure (although sometimes the labels can be a little odd).

[30]:
```
fig=plt.figure(figsize=(8, 6.5))
ssh_time_0_tile_2_DA.plot()
```

[30]:
```
<matplotlib.collections.QuadMesh at 0x7f71add1cf60>
```



Subsetting `DataArrays` using the .sel( ) syntax

Another useful method for subsetting `DataArrays` is the **.sel( )** syntax. The **.sel( )** syntax takes advantage of the fact that coordinates are **labels**. We **sel**ect subsets of the `DataArray` by providing a subset of coordinate labels.

Let's select tile 2 and time 2010-01-16:

[31]:
```
ssh_time_0_tile_2_sel = ecco_ds.SSH.sel(time='2010-01-16', tile=2)
ssh_time_0_tile_2_sel
```

[31]:
```
<xarray.DataArray 'SSH' (time: 1, j: 90, i: 90)>
array([[[0.10849833, 0.10763063, 0.10525029, ..., 0.        ,
         0.        , 0.41118386],
        [0.10236199, 0.1005336 , 0.09722137, ..., 0.31839028,
         0.        , 0.40181533],
        [0.10205435, 0.09833906, 0.09325342, ..., 0.353757  ,
         0.35896647, 0.40689626],
        ...,
```

(continues on next page)

(continued from previous page)

```
        [0.        , 0.        , 0.        , ..., 0.        ,
         0.        , 0.        ],
        [0.        , 0.        , 0.        , ..., 0.        ,
         0.        , 0.        ],
        [0.        , 0.        , 0.        , ..., 0.        ,
         0.        , 0.        ]]], dtype=float32)
Coordinates:
  * i        (i) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * j        (j) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
    tile     int64 2
    XC       (j, i) float32 -37.5 -36.5 -35.5 ... 50.968143 51.44421 51.837925
    YC       (j, i) float32 10.458642 10.458642 10.458642 ... 67.53387 67.47211
    CS       (j, i) float32 1.0 1.0 1.0 1.0 ... 0.8916124 0.9051672 0.9424238
    SN       (j, i) float32 -1.8064405e-15 9.046443e-16 ... -0.33442083
    rA       (j, i) float32 11896091000.0 11896091000.0 ... 212633870.0
    Depth    (j, i) float32 4658.681 4820.5703 5014.177 ... 0.0 0.0 0.0
    iter     (time) int32 158532
  * time     (time) datetime64[ns] 2010-01-16T12:00:00
Attributes:
    units:          m
    long_name:      Surface Height Anomaly adjusted with global steric height...
    standard_name:  sea_surface_height
```

The only difference here is that the resulting array has 'time' as a singleton dimension (dimension of length 1). I don't know why. Just use the `squeeze` command to squeeze it out.

[32]:
```
ssh_time_0_tile_2_sel = ssh_time_0_tile_2_sel.squeeze()
ssh_time_0_tile_2_sel
```

[32]:
```
<xarray.DataArray 'SSH' (j: 90, i: 90)>
array([[0.10849833, 0.10763063, 0.10525029, ..., 0.        , 0.        ,
        0.41118386],
       [0.10236199, 0.1005336 , 0.09722137, ..., 0.31839028, 0.        ,
        0.40181533],
       [0.10205435, 0.09833906, 0.09325342, ..., 0.353757  , 0.35896647,
        0.40689626],
       ...,
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ]], dtype=float32)
Coordinates:
  * i        (i) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * j        (j) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
    tile     int64 2
    XC       (j, i) float32 -37.5 -36.5 -35.5 ... 50.968143 51.44421 51.837925
    YC       (j, i) float32 10.458642 10.458642 10.458642 ... 67.53387 67.47211
    CS       (j, i) float32 1.0 1.0 1.0 1.0 ... 0.8916124 0.9051672 0.9424238
    SN       (j, i) float32 -1.8064405e-15 9.046443e-16 ... -0.33442083
    rA       (j, i) float32 11896091000.0 11896091000.0 ... 212633870.0
    Depth    (j, i) float32 4658.681 4820.5703 5014.177 ... 0.0 0.0 0.0
```

(continues on next page)

(continued from previous page)

```
    iter     int32 158532
    time     datetime64[ns] 2010-01-16T12:00:00
Attributes:
    units:          m
    long_name:      Surface Height Anomaly adjusted with global steric height...
    standard_name:  sea_surface_height
```

Subsetting `DataArrays` using the .isel( ) syntax

The last subsetting method is **.isel( )** syntax. **.isel( )** uses the numerical **index** of coordinates instead of their label. Subsets are extracted by providing a set of coordinate indices.

Because **sel()** uses the coordinate values as LABELS and **isel()** uses the index of the coordinates, they cannot necessarily be used interchangably. It is only because in ECCOv4 NetCDF files we gave the NAMES of some coordinates the same names as the INDICES of those coordinates that you can use: * **sel(tile=2)** gives you the tile with the index NAME 2 * **isel(tile=2)** gives you the tile at index 2

Let's pull out a slice of *SSH* for tile at INDEX POSITION 2 and *time* at INDEX POSITION 0

[33]:
```
ssh_time_0_tile_2_isel = ecco_ds.SSH.isel(tile=2, time=0)
ssh_time_0_tile_2_isel
```

[33]:
```
<xarray.DataArray 'SSH' (j: 90, i: 90)>
array([[0.10849833, 0.10763063, 0.10525029, ..., 0.        , 0.        ,
        0.41118386],
       [0.10236199, 0.1005336 , 0.09722137, ..., 0.31839028, 0.        ,
        0.40181533],
       [0.10205435, 0.09833906, 0.09325342, ..., 0.353757  , 0.35896647,
        0.40689626],
       ...,
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ]], dtype=float32)
Coordinates:
  * i        (i) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * j        (j) int64 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
    tile     int64 2
    XC       (j, i) float32 -37.5 -36.5 -35.5 ... 50.968143 51.44421 51.837925
    YC       (j, i) float32 10.458642 10.458642 10.458642 ... 67.53387 67.47211
    CS       (j, i) float32 1.0 1.0 1.0 1.0 ... 0.8916124 0.9051672 0.9424238
    SN       (j, i) float32 -1.8064405e-15 9.046443e-16 ... -0.33442083
    rA       (j, i) float32 11896091000.0 11896091000.0 ... 212633870.0
    Depth    (j, i) float32 4658.681 4820.5703 5014.177 ... 0.0 0.0 0.0
    iter     int32 158532
    time     datetime64[ns] 2010-01-16T12:00:00
Attributes:
    units:          m
    long_name:      Surface Height Anomaly adjusted with global steric height...
    standard_name:  sea_surface_height
```

More examples of subsetting using the [ ], .sel( ) and .isel( ) syntaxes

In the examples above we only subsetted month (Jan 2010) and a single tile (tile 2). More complex subsetting is possible. Here are some three examples that yield equivalent, more complex, subsets:

> **Note:** Python array indexing goes up to but not including the final number in a range. Because array indexing starts from 0, array index 41 corresponds to the 42nd element.

[34]:
```
ssh_sub_bracket  = ecco_ds.SSH[3, 5, 31:41, 5:22]
ssh_sub_isel     = ecco_ds.SSH.isel(tile=5, time=3, i=range(5,22), j=range(31,41))
ssh_sub_sel      = ecco_ds.SSH.sel(tile=5, time='2010-03-16', i=range(5,22), j=range(31,
↪41)).squeeze()

print('\nssh_sub_bracket')
print('--size %s ' % str(ssh_sub_bracket.shape))
print('--time %s ' % ssh_sub_bracket.time.values)
print('--tile %s ' % ssh_sub_bracket.tile.values)

print('\nssh_sub_isel')
print('--size %s ' % str(ssh_sub_isel.shape))
print('--time %s ' % ssh_sub_isel.time.values)
print('--tile %s ' % ssh_sub_isel.tile.values)


print('\nssh_sub_isel')
print('--size %s ' % str(ssh_sub_sel.shape))
print('--time %s ' % ssh_sub_sel.time.values)
print('--tile %s ' % ssh_sub_sel.tile.values)
```

```
ssh_sub_bracket
--size (10, 17)
--time 2010-04-16T12:00:00.000000000
--tile 5

ssh_sub_isel
--size (10, 17)
--time 2010-04-16T12:00:00.000000000
--tile 5

ssh_sub_isel
--size (10, 17)
--time 2010-03-16T12:00:00.000000000
--tile 5
```

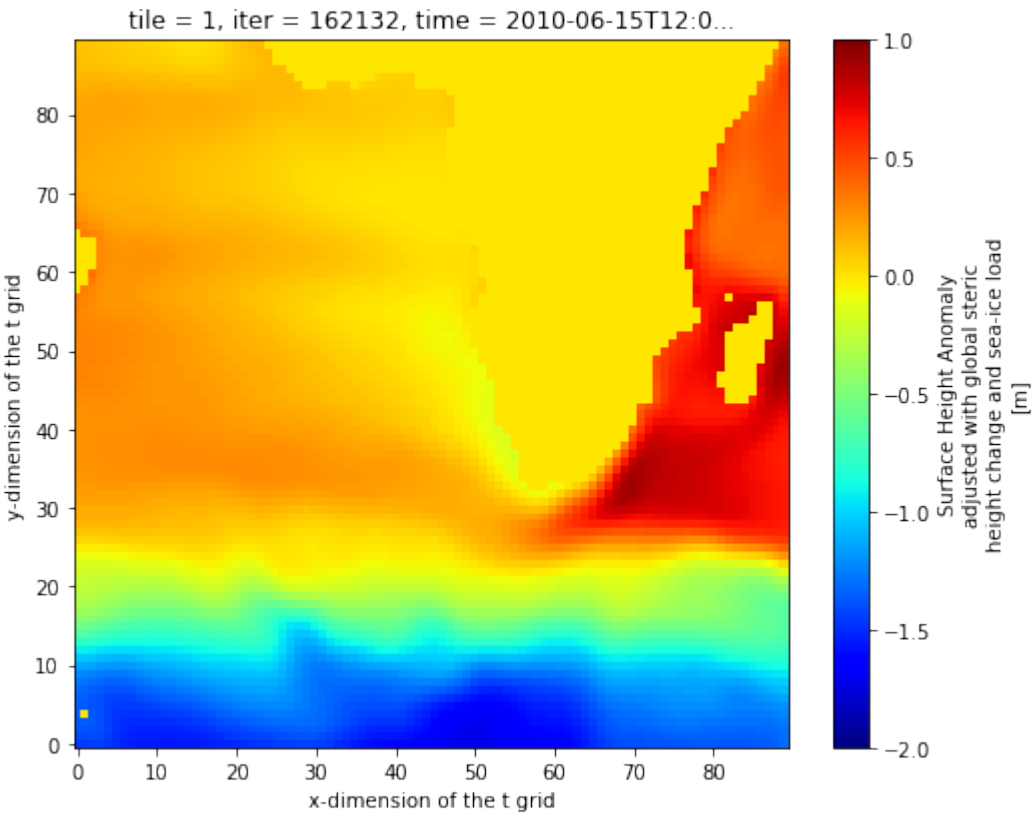
Subsetting `Datasets` using the .sel( ), and .isel( ) syntaxes

Amazingly, we can use the **.sel** and **.isel** methods to simultaneously subset multiple `DataArrays` stored within an single `Dataset`. Let's make an interesting `Dataset` to subset and then test out the **.sel( )** and **.isel( )** subsetting methods.

Let's work on tile 1, time = 5 (June, 2010)

[35]:
```
fig=plt.figure(figsize=(8, 6.5))

ecco_ds.SSH.isel(tile=1,time=5).plot(cmap='jet',vmin=-2, vmax=1)
```

[35]:
```
<matplotlib.collections.QuadMesh at 0x7f71adc03a90>
```



Subset tile 1, j = 50 (a single row through the array), and time = 5 (June 2010)

[36]:
```
output_tile1_time06_j50= ecco_ds.isel(tile=1, time=5, j=50).load()
output_tile1_time06_j50.data_vars
```

[36]:
```
Data variables:
    SSH      (i) float32 0.2910574 0.30933306 0.3110007 ... 0.8554105 0.8880241
    OBP      (i) float32 197.39972 265.14847 298.19925 ... 365.57407 446.80014
```

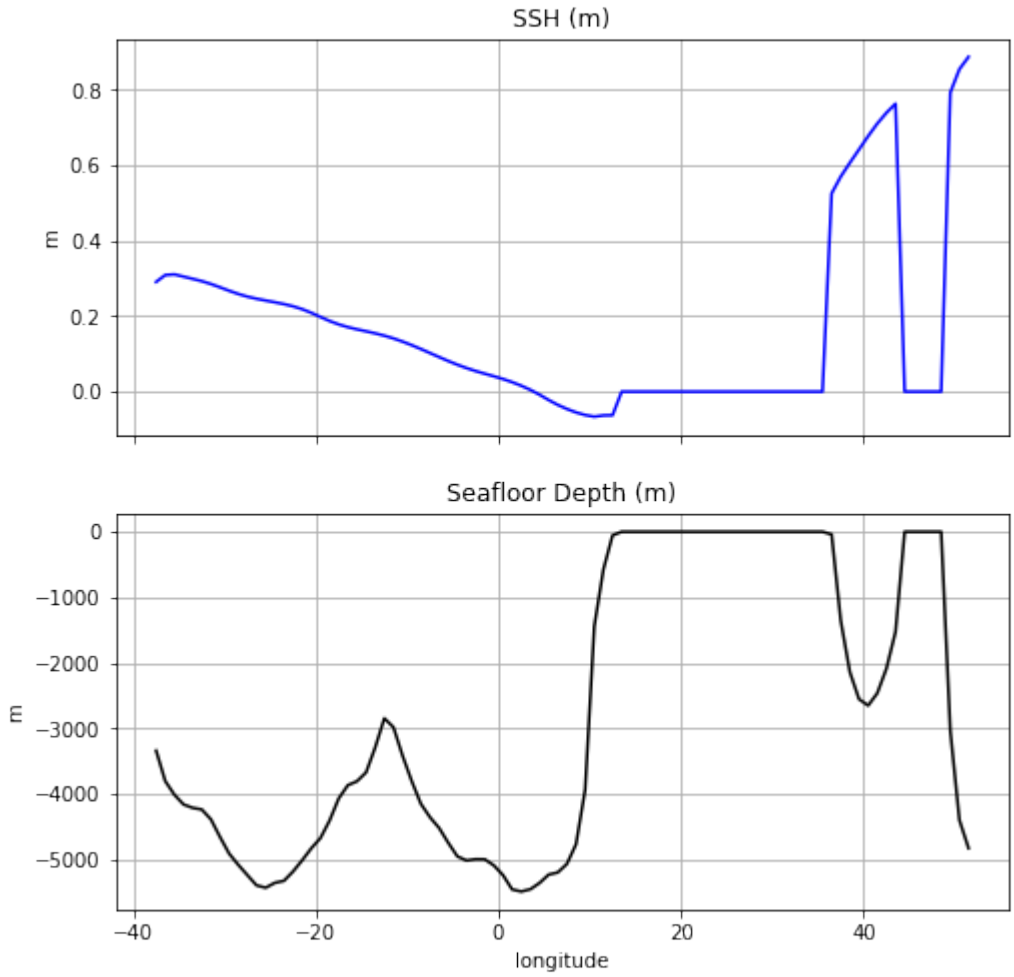
All variables that had **tile, time**, or **j** coordinates have been subset while other variables are unchanged. Let's plot the seafloor depth and sea surface height from west to east along j=50, (see plot at Line 16) which extends across the S. Atlantic, across Africa to Madagascar, and finally into to W. Indian Ocean.

[37]:
```
f, axarr = plt.subplots(2, sharex=True,figsize=(8, 8))
(ax1, ax2) = axarr
ax1.plot(output_tile1_time06_j50.XC, output_tile1_time06_j50.SSH,color='b')
```

(continues on next page)

(continued from previous page)

```
ax1.set_ylabel('m')
ax1.set_title('SSH (m)')
ax1.grid()

ax2.plot(output_tile1_time06_j50.XC, -output_tile1_time06_j50.Depth,color='k')
ax2.set_xlabel('longitude')
ax2.set_ylabel('m')
ax2.set_title('Seafloor Depth (m)')
ax2.grid()
```



Subsetting using the where( ) syntax

The **where( )** method is quite different than other subsetting methods because subsetting is done by masking out values with *nans* that do not meet some specified criteria.

For more infomation about **where( )** see http://xarray.pydata.org/en/stable/indexing.html#masking-with-where

Let's demonstrate **where** by masking out all SSH values that do not fall within a box defined between 20S to 60N and 50W to 10E.

First, we'll extract the `SSH DataArray`

[38]:
```
ssh_da=ecco_ds.SSH
```

Create a matrix that is `True` where latitude is between 20S and 60N and `False` otherwise.

[39]:
```
lat_bounds = np.logical_and(ssh_da.YC  > -20, ssh_da.YC < 60)
```

Create a matrix that is `True` where longitude is between 50W and 10E and `False` otherwise.

[40]:
```
lon_bounds = np.logical_and(ssh_da.XC  > -50, ssh_da.XC < 10)
```

Combine the `lat_bounds` and `lon_bounds` logical matrices:

[41]:
```
lat_lon_bounds = np.logical_and(lat_bounds, lon_bounds)
```

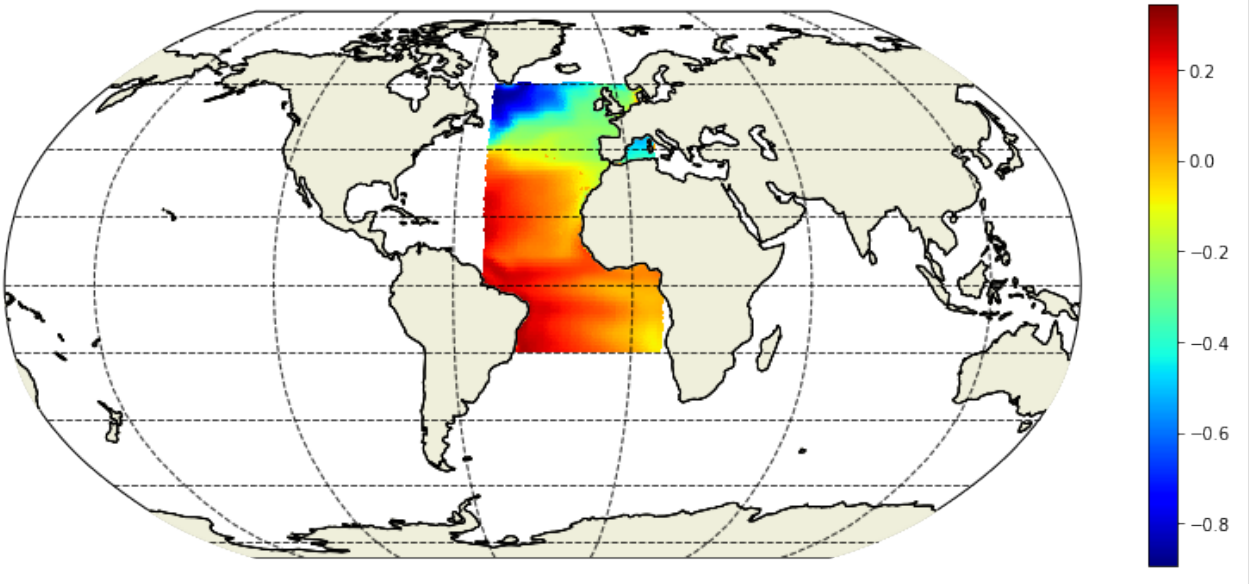
Finally, use **where** to mask out all SSH values that do not fall within our `lat_lon_bounds`

[42]:
```
ssh_da_subset_space = ssh_da.where(lat_lon_bounds, np.nan)
```

To visualize the SSH in our box we'll use one of our ECCO v4 custom plotting routines (which will be the subject of another tutorial).

Notice the use of **.sel( )** to subset a single time slice (time=1) for plotting.

[43]:
```
fig=plt.figure(figsize=(14, 6))

ecco.plot_proj_to_latlon_grid(ecco_ds.XC, ecco_ds.YC, \
                              ssh_da_subset_space.isel(time=6),\
                              dx=.5, dy=.5,user_lon_0 = -30,\
                              show_colorbar=True);
```



1.13.4 Conclusion

You now know several different methods for accessing and subsetting fields in `Dataset` and `DataArray` objects.

To learn a more about indexing/subsetting methods please refer to the `xarray` manual for indexing methods, http://xarray.pydata.org/en/stable/indexing.html.

1.14 Operating on Numpy arrays

1.14.1 Objectives

Introduce numpy's **pass-by-reference** approach handling numpy arrays and methods for avoiding pitfalls when operating on numpy arrays.

1.14.2 Introduction

From: http://span.ece.utah.edu/common-mistakes-in-moving-from-matlab-to-python:

"Whenever you reference an array in Python, the computer will provide the memory address for the thing you are accessing, not the actual value. This is called **pass-by-reference**. This saves memory and makes your programs faster, but it is also harder to keep straight."

From: https://docs.python.org/2/library/copy.html

"Assignment statements in Python do not copy objects, they create bindings [pointers] between a target and an object." "… a copy is sometimes needed so one can change one copy without changing the other. The 'copy' module provides generic … copy operations."

If you are not familiar with the **pass-by-reference** aspect of Python then I strongly suggest you read this short, informative essay on "Python Names and Values": https://nedbatchelder.com/text/names.html

We've briefly touched on this important subject in earlier tutorials. Now we'll go into a bit more detail.

1.14.3 Variable assignments

Unlike some other languages, creating a new variable with an assignment statement in Python such as `x = some_numpy_array`

does not make a copy of `some_numpy_array`. Instead, the assignment statement makes `x` and `some_numpy_array` both point to the same `numpy` array in memory. Because `x` and `some_numpy_array` are both refer (or pointer) to the same `numpy` array in memory, the `numpy` array can be changed by operations on either `x` or `some_numpy_array`. If you aren't aware of this behavior then you may run into very difficult to identify bugs in your calculations!

A simple demonstration

Let's demonstrate this issue with a very simple `numpy` array

[1]:
```
import numpy as np
import xarray as xr
import sys
import matplotlib.pyplot as plt
%matplotlib inline
```

(continues on next page)

(continued from previous page)

```
import json
from copy import deepcopy
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
## Import the ecco_v4_py library into Python
## =========================================

## -- If ecco_v4_py is not installed in your local Python library,
##    tell Python where to find it.  For example, if your ecco_v4_py
##    files are in /Users/ifenty/ECCOv4-py/ecco_v4_py, then use:

sys.path.append('/home/ifenty/ECCOv4-py')
import ecco_v4_py as ecco
```

[3]:
```
## Set top-level file directory for the ECCO NetCDF files
## ======================================================================
# base_dir = '/home/username/'
base_dir = '/home/ifenty/ECCOv4-release'

## define a high-level directory for ECCO fields
ECCO_dir = base_dir + '/Release3_alt'
```

Create a simple numpy array

[4]:
```
a=np.array([1, 2, 3, 4, 5])

# Assign 'b' to point to the same numpy array
b=a

# Test to see if b and a point to the same thing
b is a
```

[4]:
```
True
```

Now change the fourth element of b and print both a and b

[5]:
```
b[3] = 10
print (a)
print (b)
```

```
[ 1  2  3 10  5]
[ 1  2  3 10  5]
```

A fancier demonstration

Let's now demonstrate with a numpy array that stores SSH output.

[6]:
```
## LOAD NETCDF SSH FILE

# directory of the file
data_dir= ECCO_dir + '/nctiles_monthly/SSH/'
# filename
fname = 'SSH_2010.nc'
# load the dataset file
ssh_dataset = xr.open_dataset(data_dir + fname).load()

## Load the model grid
grid_dir= ECCO_dir + '/nctiles_grid/'

ecco_grid = ecco.load_ecco_grid_nc(grid_dir, 'ECCOv4r3_grid.nc')

## Merge SSH and GRID
output_all = xr.merge((ssh_dataset, ecco_grid))
```

Recall the dimensions of our SSH DataArray:

[7]:
```
output_all.SSH.dims
```

[7]:
```
('time', 'tile', 'j', 'i')
```

Show the first four SSH values in **j** and **i** for the fifth month (May 1992) and second tile:

[8]:
```
output_all.SSH[4,1,0:4,0:4].values
```

[8]:
```
array([[-1.4365957, -1.436502 , -1.4301504, -1.434309 ],
       [-1.3933921, -1.3902488, -1.3870735, -1.401771 ],
       [-1.3574185, -1.3534354, -1.3570538, -1.3828031],
       [-1.3392653, -1.3360693, -1.3463365, -1.378602 ]], dtype=float32)
```

Assign the variable ssh_tmp to this *subset* of the numpy array that SSH points to:

[9]:
```
ssh_tmp = output_all.SSH[4,1,0:2,0:2].values
ssh_tmp
```

[9]:
```
array([[-1.4365957, -1.436502 ],
       [-1.3933921, -1.3902488]], dtype=float32)
```

Now change the values of all elements of ssh_tmp to 10

[10]:
```
ssh_tmp[:] = 10
ssh_tmp
```

[10]:
```
array([[10., 10.],
       [10., 10.]], dtype=float32)
```

And see that yes, in fact, this change is reflected in our SSH DataArray:

[11]:
```
output_all.SSH[4,1,0:4,0:4].values
```

[11]:
```
array([[10.       , 10.       , -1.4301504, -1.434309 ],
       [10.       , 10.       , -1.3870735, -1.401771 ],
       [-1.3574185, -1.3534354, -1.3570538, -1.3828031],
       [-1.3392653, -1.3360693, -1.3463365, -1.378602 ]], dtype=float32)
```

1.14.4 Dealing with *pass-by-reference*: right hand side operations

One way to have a new variable assignment not point to the original variable is to *perform an operation on the right hand side of the assignment statement*.

"Python evaluates expressions from left to right. Notice that while evaluating an assignment, the right-hand side is evaluated before the left-hand side." https://docs.python.org/2/reference/expressions.html#evaluation-order

Performing an operation on the right hand side creates new values in memory. The new variable assignment will then point to these new values, leaving the original untouched.

Simple demonstration 1

Operate on a by adding 1 before the assigment statement

[12]:
```
# Create a simple numpy array
a=np.array([1, 2, 3, 4, 5])

b = a + 1

print (a)
print (b)
```

```
[1 2 3 4 5]
[2 3 4 5 6]
```

Now change the fourth element of b and print both a and b

[13]:
```
b[3] = 10
print (a)
print (b)
```

```
[1 2 3 4 5]
[ 2  3  4 10  6]
```

a and b do indeed point to different values in memory.

Simple demonstration 2

Operate on a by adding 0 before the assigment statement. This is a kind of dummy operation.

[14]:
```
# Create a simple numpy array
a=np.array([1, 2, 3, 4, 5])

# Add 0 to `a`:
b = a + 0
```

(continues on next page)

(continued from previous page)

```
print (a)
print (b)
```

```
[1 2 3 4 5]
[1 2 3 4 5]
```

[15]:
```
# Test to see if b and a point to the same thing
b is a
```

[15]:
```
False
```

Now change the fourth element of b and print both a and b

[16]:
```
b[3] = 10
print (a)
print (b)
```

```
[1 2 3 4 5]
[ 1  2  3 10  5]
```

Once again we see that a and b do indeed point to different values in memory.

A fancier demonstration

Let's now demonstrate with a numpy array that stores SSH output.

[17]:
```
output_all.SSH[4,1,5:9,5:9].values
```

[17]:
```
array([[-1.4406753, -1.4481026, -1.4435667, -1.4330329],
       [-1.4137946, -1.4145803, -1.4049884, -1.3896267],
       [-1.3741281, -1.3701942, -1.3568683, -1.3380871],
       [-1.3235712, -1.316149 , -1.2999685, -1.2789109]], dtype=float32)
```

[18]:
```
ssh_tmp = output_all.SSH[4,1,5:9,5:9].values * output_all.rA[1,5:9,5:9].values
ssh_tmp[:] = 10
ssh_tmp
```

[18]:
```
array([[10., 10., 10., 10.],
       [10., 10., 10., 10.],
       [10., 10., 10., 10.],
       [10., 10., 10., 10.]], dtype=float32)
```

[19]:
```
output_all.SSH[4,1,5:9,5:9].values
```

[19]:
```
array([[-1.4406753, -1.4481026, -1.4435667, -1.4330329],
       [-1.4137946, -1.4145803, -1.4049884, -1.3896267],
       [-1.3741281, -1.3701942, -1.3568683, -1.3380871],
       [-1.3235712, -1.316149 , -1.2999685, -1.2789109]], dtype=float32)
```

Operating on the right hand side of the assignment does indeed new arrays in memory leaving the original SSH numpy array untouched.

1.14.5 Dealing with *pass-by-reference*: copy and deepcopy

A second way to have a new variable assignment not point to the original variable is to *use the copy or deepcopy command.*

Simple demonstration

Use the `numpy` command.

[20]:
```
# Create a simple numpy array
a=np.array([1, 2, 3, 4, 5])
b=np.copy(a)

print (a)
print (b)
```

```
[1 2 3 4 5]
[1 2 3 4 5]
```

Now change the fourth element of `b` and print both `a` and `b`

[21]:
```
b[3] = 10
print (a)
print (b)
```

```
[1 2 3 4 5]
[ 1  2  3 10  5]
```

[22]:
```
output_all.SSH
```

[22]:
```
<xarray.DataArray 'SSH' (time: 12, tile: 13, j: 90, i: 90)>
array([[[[ 0.        ,  0.        ,  0.        , ...,  0.        ,
           0.        ,  0.        ],
         [ 0.        ,  0.        ,  0.        , ...,  0.        ,
           0.        ,  0.        ],
         [ 0.        ,  0.        ,  0.        , ...,  0.        ,
           0.        ,  0.        ],
         ...,
         [-1.4836141 , -1.4890583 , -1.4887266 , ..., -1.3676528 ,
          -1.3607242 , -1.3533412 ],
         [-1.4628036 , -1.468267  , -1.4667836 , ..., -1.3557527 ,
          -1.3482903 , -1.3397818 ],
         [-1.436315  , -1.4399437 , -1.4370856 , ..., -1.3461887 ,
          -1.3382714 , -1.3282658 ]],

        [[-1.4029788 , -1.4037812 , -1.4008827 , ..., -1.3376493 ,
          -1.3292958 , -1.3175498 ],
         [-1.3666646 , -1.365299  , -1.3648341 , ..., -1.3276558 ,
          -1.3189425 , -1.3056219 ],
         [-1.33584   , -1.334351  , -1.3388703 , ..., -1.3141139 ,
          -1.3052138 , -1.2908865 ],
         ...,
         [ 0.14946148,  0.1455971 ,  0.14140229, ...,  0.        ,
           0.4259282 ,  0.40249246],
```

(continues on next page)

(continued from previous page)

```
        [ 0.13342337,  0.1310458 ,  0.12794277, ...,  0.        ,
          0.42839238,  0.41154262],
        [ 0.11906417,  0.11801734,  0.11567361, ...,  0.        ,
          0.        ,  0.41444954]],

       [[ 0.10849833,  0.10763063,  0.10525029, ...,  0.        ,
          0.        ,  0.41118386],
        [ 0.10236199,  0.1005336 ,  0.09722137, ...,  0.31839028,
          0.        ,  0.40181533],
        [ 0.10205435,  0.09833906,  0.09325342, ...,  0.353757  ,
          0.35896647,  0.40689626],
        ...,
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ]],

       ...,

       [[ 0.        ,  0.        ,  0.        , ...,  0.35200754,
          0.326652  ,  0.30624378],
        [ 0.        ,  0.        ,  0.        , ...,  0.35387242,
          0.32897007,  0.3091959 ],
        [ 0.        ,  0.        ,  0.        , ...,  0.3596493 ,
          0.33368325,  0.31345782],
        ...,
        [ 0.        ,  0.        ,  0.        , ...,  0.11090944,
          0.10653148,  0.1093755 ],
        [ 0.        ,  0.        ,  0.        , ...,  0.10750817,
          0.1044946 ,  0.10859182],
        [ 0.        ,  0.        ,  0.        , ...,  0.10480525,
          0.10333875,  0.10848776]],

       [[ 0.30412942,  0.33163443,  0.388738  , ..., -0.35260823,
         -0.42698237, -0.50772387],
        [ 0.30684927,  0.3322026 ,  0.386021  , ..., -0.31668594,
         -0.38891688, -0.4718595 ],
        [ 0.3105247 ,  0.33349368,  0.38373435, ..., -0.3030887 ,
         -0.37445134, -0.45894527],
        ...,
        [ 0.11961964,  0.13626505,  0.15554132, ..., -1.2845815 ,
         -1.3033888 , -1.3298517 ],
        [ 0.11943392,  0.13602602,  0.15484709, ..., -1.3105577 ,
         -1.3325728 , -1.3604729 ],
        [ 0.11928844,  0.1349795 ,  0.15260401, ..., -1.3288026 ,
         -1.3561577 , -1.3883135 ]],

       [[-0.5964835 , -0.69236034, -0.79338306, ...,  0.        ,
          0.        ,  0.        ],
        [-0.5660535 , -0.6684363 , -0.7742219 , ...,  0.        ,
```

(continues on next page)

(continued from previous page)

```
          0.        ,  0.        ],
        [-0.5563829 , -0.66219753, -0.7694144 , ...,  0.        ,
          0.        ,  0.        ],
        ...,
        [-1.361862  , -1.3975031 , -1.4352301 , ...,  0.        ,
          0.        ,  0.        ],
        [-1.3903762 , -1.4201684 , -1.4501717 , ...,  0.        ,
          0.        ,  0.        ],
        [-1.4191432 , -1.4458475 , -1.4696381 , ...,  0.        ,
          0.        ,  0.        ]]],


      [[[ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        ...,
        [-1.489962  , -1.49527   , -1.4950125 , ..., -1.324543  ,
         -1.31546   , -1.3052927 ],
        [-1.4686763 , -1.474007  , -1.4730073 , ..., -1.3154641 ,
         -1.3059864 , -1.2944871 ],
        [-1.4415836 , -1.445527  , -1.4437983 , ..., -1.3081948 ,
         -1.2985847 , -1.2857456 ]],

       [[-1.4074324 , -1.4091369 , -1.4077874 , ..., -1.3005854 ,
         -1.2910386 , -1.2770216 ],
        [-1.3706107 , -1.3701998 , -1.3714415 , ..., -1.2901691 ,
         -1.2808572 , -1.2660264 ],
        [-1.3400941 , -1.3389751 , -1.3451121 , ..., -1.2752724 ,
         -1.2662243 , -1.2510723 ],
        ...,
        [ 0.14600152,  0.14037795,  0.13405259, ...,  0.        ,
          0.45676908,  0.46960458],
        [ 0.13392949,  0.12819704,  0.12193992, ...,  0.        ,
          0.43185055,  0.44771263],
        [ 0.12269467,  0.11731683,  0.11124787, ...,  0.        ,
          0.        ,  0.42728487]],

       [[ 0.11415915,  0.10866896,  0.10236249, ...,  0.        ,
          0.        ,  0.41325122],
        [ 0.1091438 ,  0.10260367,  0.09539554, ...,  0.34004778,
          0.        ,  0.40250835],
        [ 0.10801908,  0.10005993,  0.09191936, ...,  0.370763  ,
          0.37222847,  0.40804937],
        ...,
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
```

(continues on next page)

(continued from previous page)

```
          0.        ,  0.        ]],

       ...,

       [[ 0.        ,  0.        ,  0.        , ...,  0.39924735,
          0.38143918,  0.36281937],
        [ 0.        ,  0.        ,  0.        , ...,  0.4045941 ,
          0.38406068,  0.36293516],
        [ 0.        ,  0.        ,  0.        , ...,  0.40966666,
          0.38610974,  0.36248463],
        ...,
        [ 0.        ,  0.        ,  0.        , ...,  0.12320995,
          0.12149823,  0.12537551],
        [ 0.        ,  0.        ,  0.        , ...,  0.11993366,
          0.11874654,  0.12275659],
        [ 0.        ,  0.        ,  0.        , ...,  0.11488416,
          0.11462326,  0.11893921]],

       [[ 0.35339177,  0.36148992,  0.39120036, ..., -0.3515566 ,
         -0.4434473 , -0.5372167 ],
        [ 0.35098886,  0.35647574,  0.384084  , ..., -0.31312683,
         -0.40090024, -0.49447078],
        [ 0.34828657,  0.35177138,  0.37818742, ..., -0.29459885,
         -0.3796233 , -0.4731894 ],
        ...,
        [ 0.13400438,  0.14537305,  0.15664023, ..., -1.2893282 ,
         -1.3100575 , -1.3397995 ],
        [ 0.13128757,  0.14274582,  0.15433277, ..., -1.3145573 ,
         -1.3370266 , -1.3671248 ],
        [ 0.1274146 ,  0.1388315 ,  0.15071633, ..., -1.3328918 ,
         -1.359795  , -1.3930436 ]],

       [[-0.63250536, -0.72816867, -0.82351893, ...,  0.        ,
          0.        ,  0.        ],
        [-0.59260696, -0.6927064 , -0.79215497, ...,  0.        ,
          0.        ,  0.        ],
        [-0.5730935 , -0.6754043 , -0.7760804 , ...,  0.        ,
          0.        ,  0.        ],
        ...,
        [-1.3741646 , -1.4095769 , -1.4447788 , ...,  0.        ,
          0.        ,  0.        ],
        [-1.3989064 , -1.4287863 , -1.457388  , ...,  0.        ,
          0.        ,  0.        ],
        [-1.4253014 , -1.4526149 , -1.4762621 , ...,  0.        ,
          0.        ,  0.        ]]],


      [[[ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
```

(continues on next page)

(continued from previous page)

```
          0.        ,  0.        ],
         ...,
         [-1.5110403 , -1.5177205 , -1.5174176 , ..., -1.3357865 ,
          -1.3278884 , -1.318876  ],
         [-1.4881549 , -1.4948115 , -1.4935229 , ..., -1.3228539 ,
          -1.314229  , -1.3036441 ],
         [-1.4580069 , -1.4625804 , -1.4603127 , ..., -1.3126371 ,
          -1.303386  , -1.290982  ]],

        [[-1.4192245 , -1.4211298 , -1.4195518 , ..., -1.3029585 ,
          -1.2933172 , -1.2792544 ],
         [-1.3780817 , -1.3780969 , -1.3797989 , ..., -1.2915645 ,
          -1.2817683 , -1.2664136 ],
         [-1.3462648 , -1.3455915 , -1.3522567 , ..., -1.2767069 ,
          -1.2668498 , -1.2507555 ],
         ...,
         [ 0.1566297 ,  0.14574713,  0.1340149 , ...,  0.        ,
           0.5034744 ,  0.52975076],
         [ 0.14160103,  0.13072166,  0.11933897, ...,  0.        ,
           0.46551415,  0.49194607],
         [ 0.12809302,  0.1176656 ,  0.10695522, ...,  0.        ,
           0.        ,  0.45891845]],

        [[ 0.11827811,  0.10802748,  0.09787346, ...,  0.        ,
           0.        ,  0.4385822 ],
         [ 0.11309812,  0.10250971,  0.09244011, ...,  0.3657641 ,
           0.        ,  0.42507946],
         [ 0.11333717,  0.10238108,  0.09199497, ...,  0.3817734 ,
           0.38925597,  0.42275748],
         ...,
         [ 0.        ,  0.        ,  0.        , ...,  0.        ,
           0.        ,  0.        ],
         [ 0.        ,  0.        ,  0.        , ...,  0.        ,
           0.        ,  0.        ],
         [ 0.        ,  0.        ,  0.        , ...,  0.        ,
           0.        ,  0.        ]],

        ...,

        [[ 0.        ,  0.        ,  0.        , ...,  0.42629334,
           0.40751386,  0.38404158],
         [ 0.        ,  0.        ,  0.        , ...,  0.42555302,
           0.40451616,  0.37930268],
         [ 0.        ,  0.        ,  0.        , ...,  0.42385182,
           0.40150982,  0.37547565],
         ...,
         [ 0.        ,  0.        ,  0.        , ...,  0.14473848,
           0.14240266,  0.14556926],
         [ 0.        ,  0.        ,  0.        , ...,  0.13558038,
           0.13387772,  0.13747904],
         [ 0.        ,  0.        ,  0.        , ...,  0.12472015,
           0.12382509,  0.12815464]],
```

(continues on next page)

(continued from previous page)

```
 [[ 0.36552784,  0.35883728,  0.36747   , ..., -0.37859654,
   -0.4690925 , -0.5647183 ],
  [ 0.35887176,  0.35017943,  0.35735488, ..., -0.34685484,
   -0.4315016 , -0.5234984 ],
  [ 0.35353845,  0.34268492,  0.34815198, ..., -0.33039528,
   -0.41046697, -0.4991011 ],
  ...,
  [ 0.15426582,  0.16672038,  0.17900392, ..., -1.2917107 ,
   -1.3121684 , -1.3426142 ],
  [ 0.1466583 ,  0.15988627,  0.17363326, ..., -1.3188969 ,
   -1.3416951 , -1.3739729 ],
  [ 0.13771701,  0.1512237 ,  0.16593452, ..., -1.3384157 ,
   -1.3662696 , -1.4031518 ]],

 [[-0.66363436, -0.76205665, -0.85687906, ...,  0.        ,
    0.        ,  0.        ],
  [-0.6209822 , -0.71981865, -0.8161234 , ...,  0.        ,
    0.        ,  0.        ],
  [-0.59452295, -0.692481  , -0.7885612 , ...,  0.        ,
    0.        ,  0.        ],
  ...,
  [-1.3775544 , -1.4144093 , -1.4529849 , ...,  0.        ,
    0.        ,  0.        ],
  [-1.4076394 , -1.439422  , -1.4712203 , ...,  0.        ,
    0.        ,  0.        ],
  [-1.4387896 , -1.4683706 , -1.4941783 , ...,  0.        ,
    0.        ,  0.        ]]],


...,


[[[ 0.        ,  0.        ,  0.        , ...,  0.        ,
    0.        ,  0.        ],
  [ 0.        ,  0.        ,  0.        , ...,  0.        ,
    0.        ,  0.        ],
  [ 0.        ,  0.        ,  0.        , ...,  0.        ,
    0.        ,  0.        ],
  ...,
  [-1.5169188 , -1.5242634 , -1.5220859 , ..., -1.36742   ,
   -1.3529748 , -1.3388332 ],
  [-1.4893768 , -1.4971659 , -1.4928439 , ..., -1.3538916 ,
   -1.3381528 , -1.3220502 ],
  [-1.4557267 , -1.4612318 , -1.4543641 , ..., -1.3420252 ,
   -1.3259686 , -1.308566  ]],

 [[-1.4152532 , -1.4169811 , -1.409366  , ..., -1.3315804 ,
   -1.3167756 , -1.2991067 ],
  [-1.3728914 , -1.3715063 , -1.3669327 , ..., -1.3211186 ,
   -1.3085409 , -1.2908047 ],
  [-1.3378468 , -1.3357688 , -1.3373411 , ..., -1.3085197 ,
```

(continues on next page)

(continued from previous page)

```
         -1.298021  , -1.280042  ],
        ...,
        [ 0.14488927,  0.14486471,  0.1455704 , ...,  0.        ,
          0.41786316,  0.48173904],
        [ 0.11153719,  0.11205091,  0.11443915, ...,  0.        ,
          0.37641078,  0.44489366],
        [ 0.08118004,  0.08179028,  0.08559205, ...,  0.        ,
          0.        ,  0.40048105]],

       [[ 0.05805106,  0.05808871,  0.0618963 , ...,  0.        ,
          0.        ,  0.36893418],
        [ 0.04563557,  0.0444993 ,  0.04670291, ...,  0.23532906,
          0.        ,  0.33999464],
        [ 0.04870504,  0.04574053,  0.04486019, ...,  0.24327128,
          0.26102594,  0.32868192],
        ...,
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ]],

       ...,

       [[ 0.        ,  0.        ,  0.        , ...,  0.31560326,
          0.28781885,  0.30457547],
        [ 0.        ,  0.        ,  0.        , ...,  0.30183744,
          0.27608654,  0.29699475],
        [ 0.        ,  0.        ,  0.        , ...,  0.2909881 ,
          0.26693258,  0.29118684],
        ...,
        [ 0.        ,  0.        ,  0.        , ...,  0.07493891,
          0.07432082,  0.08609825],
        [ 0.        ,  0.        ,  0.        , ...,  0.06333499,
          0.061132  ,  0.07336642],
        [ 0.        ,  0.        ,  0.        , ...,  0.05451504,
          0.05124369,  0.06337053]],

       [[ 0.36829612,  0.45668772,  0.5346397 , ..., -0.5123356 ,
         -0.5917844 , -0.6731994 ],
        [ 0.36443853,  0.4552048 ,  0.5336968 , ..., -0.4702206 ,
         -0.54875636, -0.6336393 ],
        [ 0.36134198,  0.45349684,  0.53227144, ..., -0.4475117 ,
         -0.5257021 , -0.6136234 ],
        ...,
        [ 0.10326117,  0.12333801,  0.14608134, ..., -1.28159   ,
         -1.2933701 , -1.3159804 ],
        [ 0.09370542,  0.11881482,  0.1466319 , ..., -1.308702  ,
         -1.3279408 , -1.3554627 ],
        [ 0.08548156,  0.11409506,  0.14582734, ..., -1.3291799 ,
         -1.3578951 , -1.3934467 ]],
```

(continues on next page)

(continued from previous page)

```
       [[-0.75963545, -0.8508026 , -0.9435896 , ...,  0.        ,
          0.        ,  0.        ],
        [-0.72582525, -0.8225674 , -0.9187593 , ...,  0.        ,
          0.        ,  0.        ],
        [-0.7104533 , -0.8112295 , -0.9089868 , ...,  0.        ,
          0.        ,  0.        ],
        ...,
        [-1.3510643 , -1.3976614 , -1.4507179 , ...,  0.        ,
          0.        ,  0.        ],
        [-1.3892317 , -1.4282138 , -1.4713119 , ...,  0.        ,
          0.        ,  0.        ],
        [-1.4299054 , -1.4643455 , -1.4974834 , ...,  0.        ,
          0.        ,  0.        ]]],


      [[[ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
          0.        ,  0.        ],
        ...,
        [-1.4967054 , -1.502654  , -1.5001634 , ..., -1.3340458 ,
         -1.3227609 , -1.31188   ],
        [-1.4694204 , -1.4754978 , -1.4709007 , ..., -1.3241014 ,
         -1.311319  , -1.2982643 ],
        [-1.4372219 , -1.440807  , -1.4335835 , ..., -1.3163857 ,
         -1.3029233 , -1.2881821 ]],

       [[-1.3995142 , -1.399057  , -1.3908552 , ..., -1.3096857 ,
         -1.2967411 , -1.2810549 ],
        [-1.359279  , -1.3553827 , -1.3499216 , ..., -1.3022889 ,
         -1.290798  , -1.274405  ],
        [-1.323668  , -1.3191642 , -1.3201401 , ..., -1.2925091 ,
         -1.2827281 , -1.2657163 ],
        ...,
        [ 0.11687395,  0.11706704,  0.11802986, ...,  0.        ,
          0.4334885 ,  0.44428313],
        [ 0.09929278,  0.10102028,  0.10379542, ...,  0.        ,
          0.41442624,  0.43558455],
        [ 0.08507469,  0.08816329,  0.0925502 , ...,  0.        ,
          0.        ,  0.4132457 ]],

       [[ 0.07387409,  0.077453  ,  0.08254188, ...,  0.        ,
          0.        ,  0.39164448],
        [ 0.06708062,  0.06982116,  0.07399624, ...,  0.24350804,
          0.        ,  0.3668178 ],
        [ 0.0685271 ,  0.06928899,  0.07084791, ...,  0.28005257,
          0.29600853,  0.36135092],
        ...,
        [ 0.        ,  0.        ,  0.        , ...,  0.        ,
```

(continues on next page)

(continued from previous page)

```
          0.        ,  0.        ],
         [ 0.        ,  0.        ,  0.        , ...,  0.        ,
           0.        ,  0.        ],
         [ 0.        ,  0.        ,  0.        , ...,  0.        ,
           0.        ,  0.        ]],

        ...,

        [[ 0.        ,  0.        ,  0.        , ...,  0.26727462,
           0.22990613,  0.23907676],
         [ 0.        ,  0.        ,  0.        , ...,  0.25806925,
           0.22213893,  0.23322569],
         [ 0.        ,  0.        ,  0.        , ...,  0.25158796,
           0.21656525,  0.22877966],
         ...,
         [ 0.        ,  0.        ,  0.        , ...,  0.08118027,
           0.07804877,  0.08201623],
         [ 0.        ,  0.        ,  0.        , ...,  0.07406381,
           0.07104729,  0.07628159],
         [ 0.        ,  0.        ,  0.        , ...,  0.06990036,
           0.06730542,  0.07331178]],

        [[ 0.29907313,  0.3912239 ,  0.48033908, ..., -0.47317785,
          -0.56365776, -0.65234643],
         [ 0.2947312 ,  0.3865718 ,  0.47382972, ..., -0.42516652,
          -0.5157564 , -0.61019945],
         [ 0.29063997,  0.38139656,  0.46688396, ..., -0.39321777,
          -0.4851443 , -0.58585995],
         ...,
         [ 0.08940418,  0.10047803,  0.11612792, ..., -1.2786794 ,
          -1.2925618 , -1.3149059 ],
         [ 0.08617538,  0.09981632,  0.11717496, ..., -1.301788  ,
          -1.3228261 , -1.3493527 ],
         [ 0.08436081,  0.09902733,  0.11718916, ..., -1.3183063 ,
          -1.3480992 , -1.381798  ]],

        [[-0.74166995, -0.83347535, -0.92762154, ...,  0.        ,
           0.        ,  0.        ],
         [-0.707929  , -0.8075284 , -0.906459  , ...,  0.        ,
           0.        ,  0.        ],
         [-0.69183034, -0.7981904 , -0.8999608 , ...,  0.        ,
           0.        ,  0.        ],
         ...,
         [-1.348128  , -1.3918104 , -1.4414979 , ...,  0.        ,
           0.        ,  0.        ],
         [-1.3810068 , -1.4177865 , -1.4585305 , ...,  0.        ,
           0.        ,  0.        ],
         [-1.4156406 , -1.4483571 , -1.4804294 , ...,  0.        ,
           0.        ,  0.        ]]],


       [[[ 0.        ,  0.        ,  0.        , ...,  0.        ,
```

(continues on next page)

(continued from previous page)

```
          0.        ,  0.        ],
         [ 0.        ,  0.        ,  0.        , ...,  0.        ,
           0.        ,  0.        ],
         [ 0.        ,  0.        ,  0.        , ...,  0.        ,
           0.        ,  0.        ],
         ...,
         [-1.463372  , -1.466484  , -1.4635477 , ..., -1.318022  ,
          -1.3103173 , -1.3024466 ],
         [-1.4383494 , -1.4415286 , -1.4376326 , ..., -1.3078314 ,
          -1.2983781 , -1.2879825 ],
         [-1.4094868 , -1.4109806 , -1.4058659 , ..., -1.3008653 ,
          -1.2901821 , -1.277352  ]],

        [[-1.3759882 , -1.3743817 , -1.3691081 , ..., -1.2955976 ,
          -1.2844023 , -1.2695681 ],
         [-1.3394302 , -1.3350021 , -1.3323509 , ..., -1.2904404 ,
          -1.2794557 , -1.2629302 ],
         [-1.3061111 , -1.301164  , -1.3042493 , ..., -1.2837274 ,
          -1.2732459 , -1.2554117 ],
         ...,
         [ 0.12598938,  0.1267241 ,  0.12829028, ...,  0.        ,
           0.3990867 ,  0.37388888],
         [ 0.1142365 ,  0.1159412 ,  0.11839783, ...,  0.        ,
           0.39991677,  0.3842097 ],
         [ 0.1103783 ,  0.11341098,  0.11685228, ...,  0.        ,
           0.        ,  0.3827097 ]],

        [[ 0.11015419,  0.11427843,  0.11837347, ...,  0.        ,
           0.        ,  0.37261513],
         [ 0.11017552,  0.11442809,  0.11824088, ...,  0.27746603,
           0.        ,  0.3570833 ],
         [ 0.11064535,  0.11360037,  0.11599879, ...,  0.32267517,
           0.3240342 ,  0.36032936],
         ...,
         [ 0.        ,  0.        ,  0.        , ...,  0.        ,
           0.        ,  0.        ],
         [ 0.        ,  0.        ,  0.        , ...,  0.        ,
           0.        ,  0.        ],
         [ 0.        ,  0.        ,  0.        , ...,  0.        ,
           0.        ,  0.        ]],

        ...,

        [[ 0.        ,  0.        ,  0.        , ...,  0.25312427,
           0.20679894,  0.19787078],
         [ 0.        ,  0.        ,  0.        , ...,  0.24408737,
           0.19740611,  0.18982588],
         [ 0.        ,  0.        ,  0.        , ...,  0.23627043,
           0.19088954,  0.18501939],
         ...,
         [ 0.        ,  0.        ,  0.        , ...,  0.11057193,
           0.10666864,  0.10647956],
```

(continues on next page)

(continued from previous page)

```
         [ 0.        ,  0.        ,  0.        , ...,  0.10842457,
           0.10574125,  0.10616441],
         [ 0.        ,  0.        ,  0.        , ...,  0.10844468,
           0.10687705,  0.10721517]],

        [[ 0.23266444,  0.30498484,  0.39119166, ..., -0.45387354,
          -0.54551655, -0.63072217],
         [ 0.22718684,  0.30146563,  0.3875697 , ..., -0.39773405,
          -0.48800388, -0.5802953 ],
         [ 0.22384512,  0.2981879 ,  0.38286942, ..., -0.35758346,
          -0.4479277 , -0.547731  ],
         ...,
         [ 0.10817628,  0.11316824,  0.12408888, ..., -1.2703364 ,
          -1.2884878 , -1.312592  ],
         [ 0.10809258,  0.1135454 ,  0.1254746 , ..., -1.290234  ,
          -1.3133153 , -1.3399168 ],
         [ 0.10853176,  0.11351557,  0.12581852, ..., -1.3032938 ,
          -1.3325437 , -1.3641984 ]],

        [[-0.7145314 , -0.80162513, -0.89393973, ...,  0.        ,
           0.        ,  0.        ],
         [-0.67591274, -0.77460396, -0.87444943, ...,  0.        ,
           0.        ,  0.        ],
         [-0.6548358 , -0.76413405, -0.869508  , ...,  0.        ,
           0.        ,  0.        ],
         ...,
         [-1.343692  , -1.3808477 , -1.4207497 , ...,  0.        ,
           0.        ,  0.        ],
         [-1.3690091 , -1.4003408 , -1.43325   , ...,  0.        ,
           0.        ,  0.        ],
         [-1.394759  , -1.4236512 , -1.4513302 , ...,  0.        ,
           0.        ,  0.        ]]]], dtype=float32)
Coordinates:
  * j        (j) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * i        (i) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
    XC       (tile, j, i) float32 -111.60647 -111.303 ... -105.58465 -111.86579
    YC       (tile, j, i) float32 -88.24259 -88.382515 ... -88.07871 -88.10267
    rA       (tile, j, i) float32 362256450.0 363300960.0 ... 361119100.0
  * tile     (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
    iter     (time) int32 158532 159204 159948 160668 ... 165084 165804 166548
  * time     (time) datetime64[ns] 2010-01-16T12:00:00 ... 2010-12-16T12:00:00
    CS       (tile, j, i) float32 0.06157813 0.06675376 ... -0.9983638
    SN       (tile, j, i) float32 -0.99810225 -0.9977695 ... -0.057182025
    Depth    (tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
Attributes:
    units:          m
    long_name:      Surface Height Anomaly adjusted with global steric height...
    standard_name:  sea_surface_height
```

Fancier demonstration

`Dataset` and `DataArray` objects are too complicated for `numpy`'s `copy` command. For complex objects such as these use the `deepcopy` command.

[23]:
```
ssh_tmp = deepcopy(output_all.SSH)
ssh_tmp[:] = 10
ssh_tmp[4,1,5:9,5:9].values
```

[23]:
```
array([[10., 10., 10., 10.],
       [10., 10., 10., 10.],
       [10., 10., 10., 10.],
       [10., 10., 10., 10.]], dtype=float32)
```

[24]:
```
output_all.SSH[4,1,5:9,5:9].values
```

[24]:
```
array([[-1.4406753, -1.4481026, -1.4435667, -1.4330329],
       [-1.4137946, -1.4145803, -1.4049884, -1.3896267],
       [-1.3741281, -1.3701942, -1.3568683, -1.3380871],
       [-1.3235712, -1.316149 , -1.2999685, -1.2789109]], dtype=float32)
```

Using `deepcopy` gives us an entirely new array in memory. Operations on `ssh_tmp` do not affect the original fields that we found in the `output_all_SSH DataArray`.

alternative to deepcopy

`xarray` give us another way to deepcopy `DataArrays` and `Datasets`:

```
ssh_tmp = output_all.copy(deep=True)
```

1.14.6 Conclusion

You now know about the possible pitfalls for dealing with Python's **pass-by-reference** way of handling assignment statements and different methods for making copies of `numpy` arrays and `Datasets` and `DataArrays`.

1.15 Plotting Tiles

1.15.1 Objectives

Introduce several different methods for plotting ECCO v4 fields that are stored as tiles in `Datasets` or `DataArrays`. Emphasis is placed on fields stored on the ECCO v4 native llc90 grid and loaded from NetCDF tile files.

1.15.2 Introduction

"Over the years many different plotting modules and packages have been developed for Python. For most of that time there was no clear favorite package, but recently matplotlib has become the most widely used. Nevertheless, many of the others are still available and may suit your tastes or needs better. Some of these are interfaces to existing plotting libraries while others are Python-centered new implementations. – from : https://wiki.python.org/moin/NumericAndScientific/Plotting

The link above profiles a long list of Python tools for plotting. In this tutorial we use just two libraries, *matplotlib* and *Cartopy*.

matplotlib

"Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shell, the jupyter notebook, web application servers, and four graphical user interface toolkits."

"For simple plotting the pyplot module provides a MATLAB-like interface, particularly when combined with [Juypter Notebooks]. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users." – from https://matplotlib.org/index.html

Matplotlib and pyplot even have a tutorial: https://matplotlib.org/users/pyplot_tutorial.html

Cartopy

"Cartopy is a Python package designed for geospatial data processing in order to produce maps and other geospatial data analyses."

Cartopy makes use of the powerful PROJ.4, NumPy and Shapely libraries and includes a programmatic interface built on top of Matplotlib for the creation of publication quality maps.

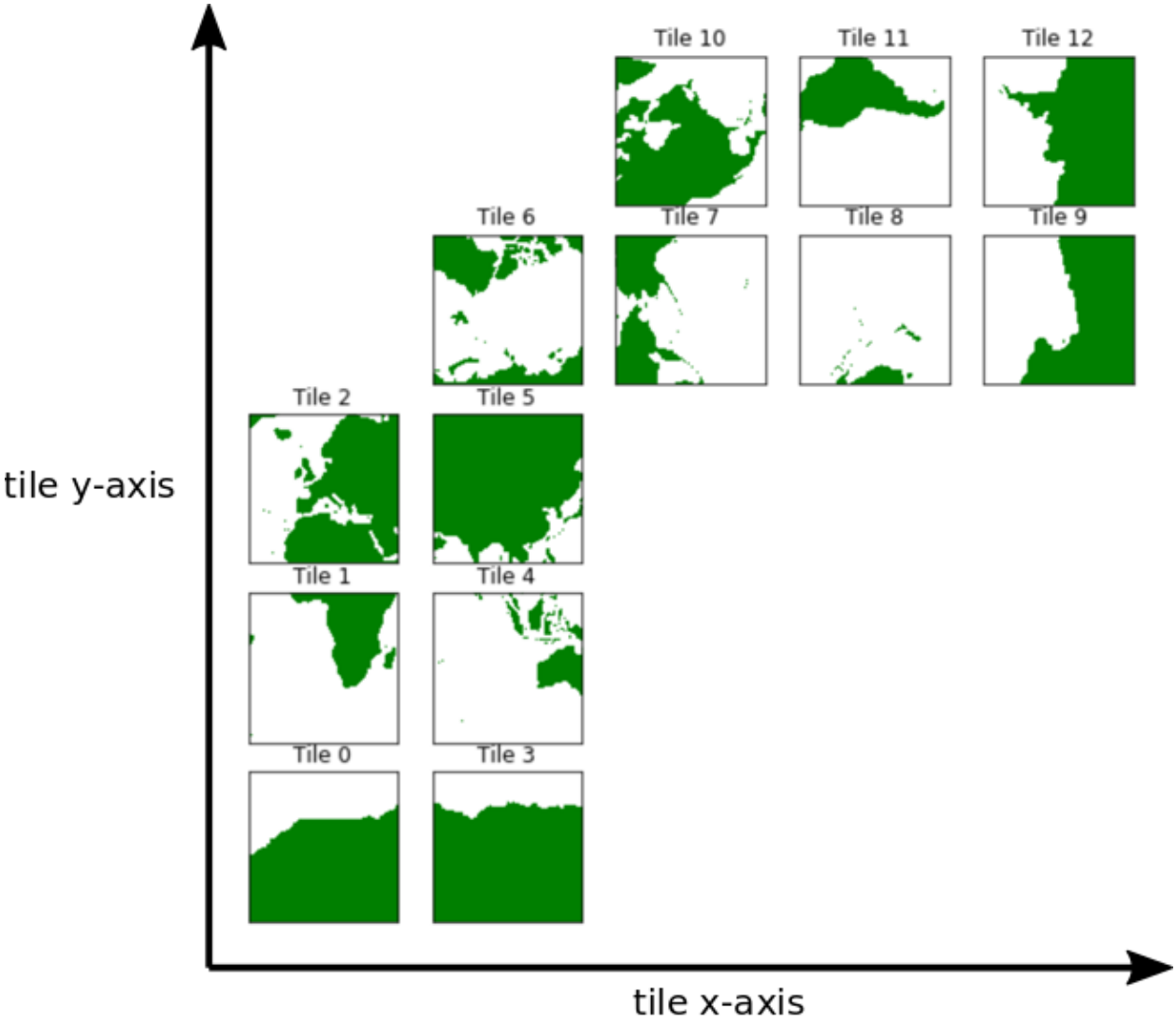
Key features of cartopy are its object oriented projection definitions, and its ability to transform points, lines, vectors, polygons and images between those projections.

You will find cartopy especially useful for large area / small scale data, where Cartesian assumptions of spherical data traditionally break down. If you've ever experienced a singularity at the pole or a cut-off at the dateline, it is likely you will appreciate cartopy's unique features!"*

– from https://scitools.org.uk/cartopy/docs/latest/

1.15.3 The default orientation of the lat-lon-cap tile fields

Before we begin plotting ECCOv4 fields on the native llc90 model grid we are reminded how how the 13 tiles are oriented with respect to their "local" **x** and **y** and with respect to each other.



Tiles 7-12 are rotated 90 degrees counter-clockwise relative to tiles 0-5.

> **Note:** *The rotated orientation of tiles 7-12 presents some complications but don't panic! The good news is that you don't need to reorient tiles to plot them.*

1.15.4 Plotting single tiles using `imshow`, `pcolormesh`, and `contourf`

First, let's load the all 13 tiles for sea surface height and the model grid parameters.

[1]:
```
import numpy as np
import sys
import xarray as xr
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
# load some useful cartopy routines
from cartopy import config
import cartopy.crs as ccrs
import cartopy.feature as cfeature

# and a new matplotlib routine
import matplotlib.path as mpath
```

[3]:
```
## Import the ecco_v4_py library into Python
## =========================================

## -- If ecco_v4_py is not installed in your local Python library,
##    tell Python where to find it.  For example, if your ecco_v4_py
##    files are in /Users/ifenty/ECCOv4-py/ecco_v4_py, then use:

sys.path.append('/home/ifenty/ECCOv4-py')
import ecco_v4_py as ecco
```

[4]:
```
## Set top-level file directory for the ECCO NetCDF files
## ======================================================
# base_dir = homehome/username/'
base_dir = '/home/ifenty/ECCOv4-release'

## define a high-level directory for ECCO fields
ECCO_dir = base_dir + '/Release3_alt'
```

[5]:
```
## Load the model grid
grid_dir= ECCO_dir + '/nctiles_grid/'

ecco_grid = ecco.load_ecco_grid_nc(grid_dir, 'ECCOv4r3_grid.nc')

## Load one year of 2D daily data, SSH, SST, and SSS
day_mean_dir= ECCO_dir + '/nctiles_daily/'

ecco_vars = ecco.recursive_load_ecco_var_from_years_nc(day_mean_dir, \
                                            vars_to_load=['SSH','THETA', 'SALT'], \
                                            years_to_load=2000)

## Merge the ecco_grid with the ecco_vars to make the ecco_ds
ecco_ds = xr.merge((ecco_grid , ecco_vars)).load()
```

```
loading files of  SALT
loading files of  SSH
loading files of  THETA
```

Plotting a single tile with `imshow`

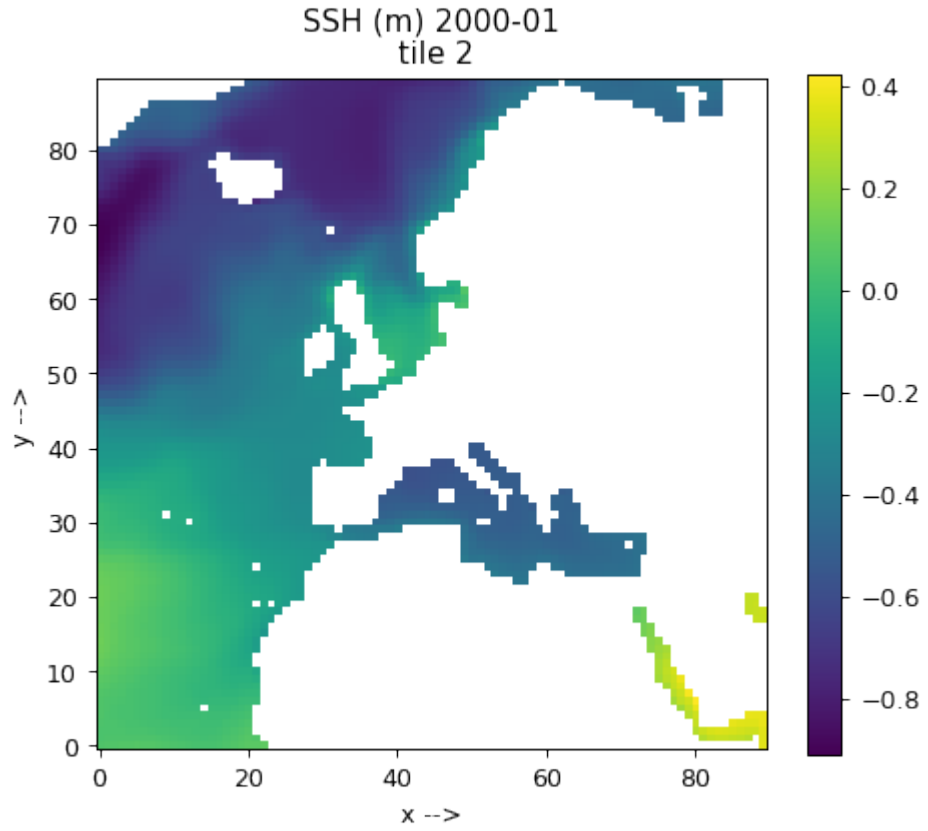
First we'll plot the average SSH for the first month (Jan 1992) on tiles 3, 7, and 8 using the basic `imshow` routine from *pyplot*. We are plotting these three different tiles to show that these lat-lon-cap tiles all have a different orientation in x and y.

> **Note:** *The* **origin='lower'** *argument to ``imshow`` is required to make the :math:`y` origin at the bottom of the plot.*

Tile 2 (Northeast Atlantic)

[6]:
```
plt.figure(figsize=(6,5), dpi= 90)

# Step 1, select the tile to plot using the **.isel( )** syntax.
tile_to_plot = ecco_ds.SSH.isel(tile=2, time=0)
tile_to_plot= tile_to_plot.where(ecco_ds.hFacC.isel(tile=2,k=0) !=0, np.nan)

# Step 2, use plt.imshow()
plt.imshow(tile_to_plot, origin='lower');

# Step 3, add colorbar, title, and x and y axis labels
plt.colorbar()
plt.title('SSH (m) ' + str(ecco_ds.time[0].values)[0:7] + '\n tile 2')
plt.xlabel('x -->')
plt.ylabel('y -->')
```

[6]:
```
Text(0, 0.5, 'y -->')
```

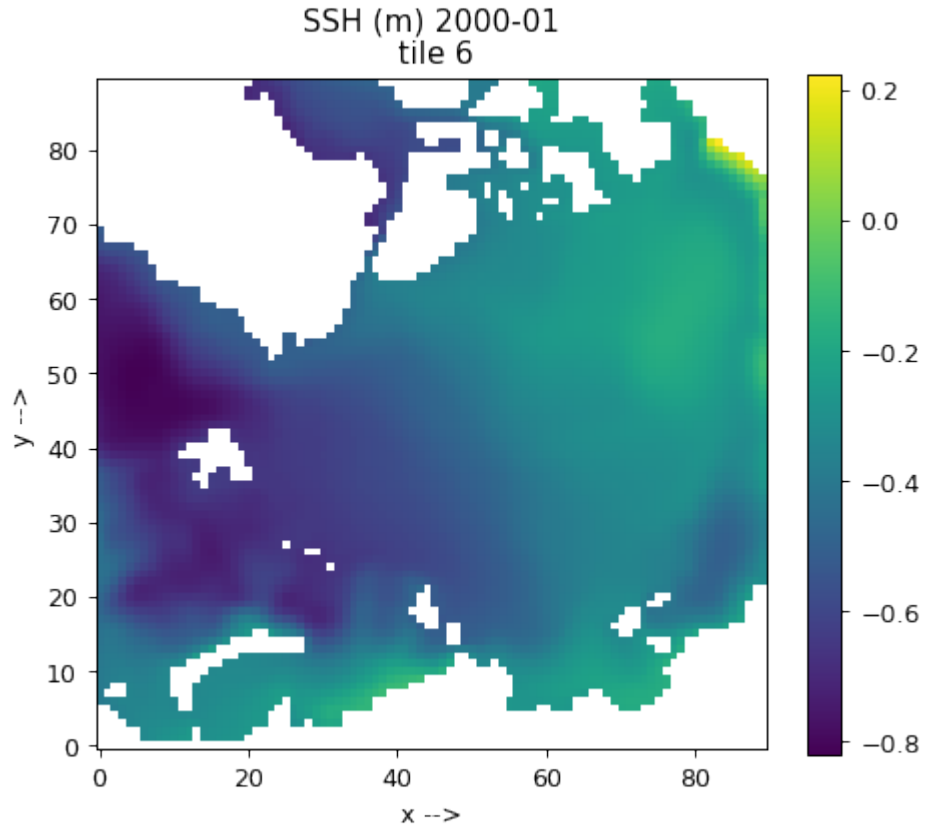


Tiles 0-5 are by default in a quasi-lat-lon orientation. $+x$ is to the east and $+y$ is to the north.

Tile 6 (the Arctic cap)

This time we'll plot the Arctic cap tile 6. Notice the layout of the Arctic cap tile in x and y. We'll follow the same procedure for plotting except we'll use LaTeX to add arrows in the x and y axis labels (for fun).

[7]:
```
plt.figure(figsize=(6,5), dpi= 90)

# Step 1, select the tile to plot using the **.isel( )** syntax.
tile_to_plot = ecco_ds.SSH.isel(tile=6, time=0)
tile_to_plot= tile_to_plot.where(ecco_ds.hFacC.isel(tile=6,k=0) !=0, np.nan)

# Step 2, use plt.imshow()
plt.imshow(tile_to_plot, origin='lower');

# Step 3, add colorbar, title, and x and y axis labels
plt.colorbar()
plt.title('SSH (m) ' + str(ecco_ds.time[0].values)[0:7] + '\n tile 6')
plt.xlabel('x -->');
plt.ylabel('y -->');
```

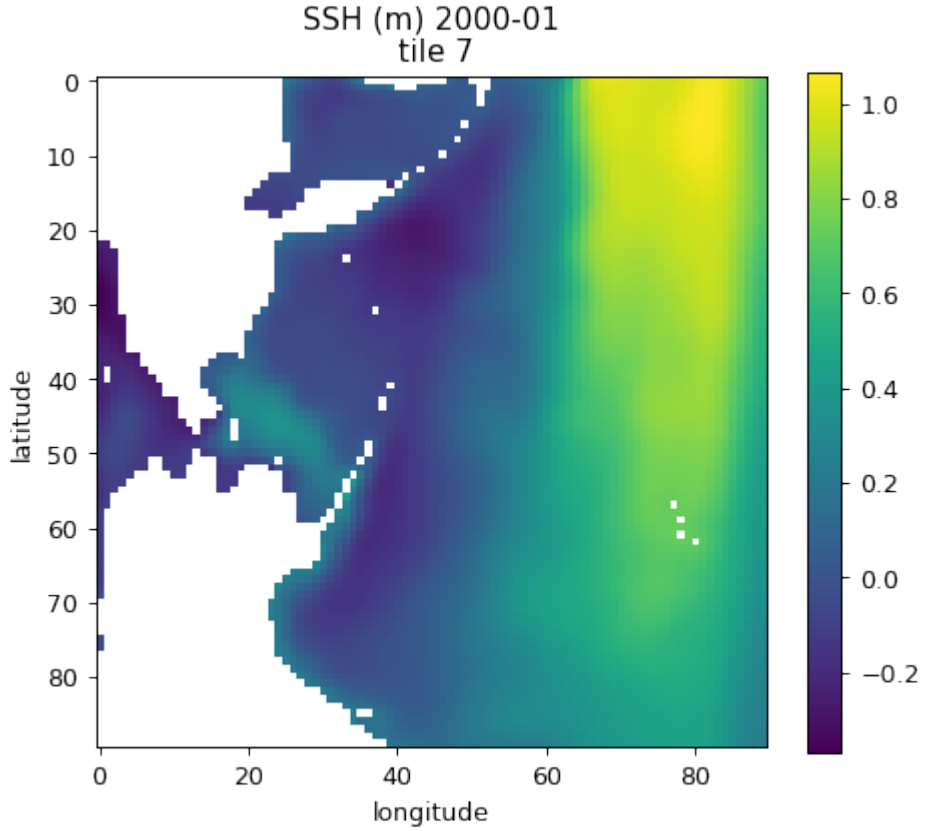


Because tile 6 is the Arctic cap, x and y do not map to east and west throughout the domain.

Tile 7 (N. Pacific / Bering Sea / Chukchi Sea)

For tiles 7-12 , positive x is southwards and positive y is eastwards.

[8]:
```
plt.figure(figsize=(6,5), dpi= 90)

# pull out lats and lons
tile_num=7
tile_to_plot = ecco_ds.SSH.isel(tile=tile_num, time=1)
tile_to_plot= tile_to_plot.where(ecco_ds.hFacC.isel(tile=tile_num,k=0) !=0, np.nan)

plt.imshow(tile_to_plot)
plt.colorbar()
plt.title('SSH (m) ' + str(ecco_ds.time[1].values)[0:7] + '\n tile ' + str(tile_num))
plt.xlabel('longitude');
plt.ylabel('latitude');
```



Tiles 7-12 are are also in a quasi-lat-lon orientation except that $+x$ is roughly south and $+y$ is roughy east.

Plotting a single tile with `pcolor` and `contourf`

The `pcolor` and `contourf` routines allows us to add latitude and longitude to the figure. Because SSH is a 'c' point variable, its lat/lon coordinates are *YC* and *XC*

We can't plot the Arctic cap tile with pcolor and contourf using latitude and longitude for the plot x and y axes because of the singularity at the pole and the 360 wrapping in longitude.

Instead, we will demonstrate `pcolor` and `contourf` for tile 2.

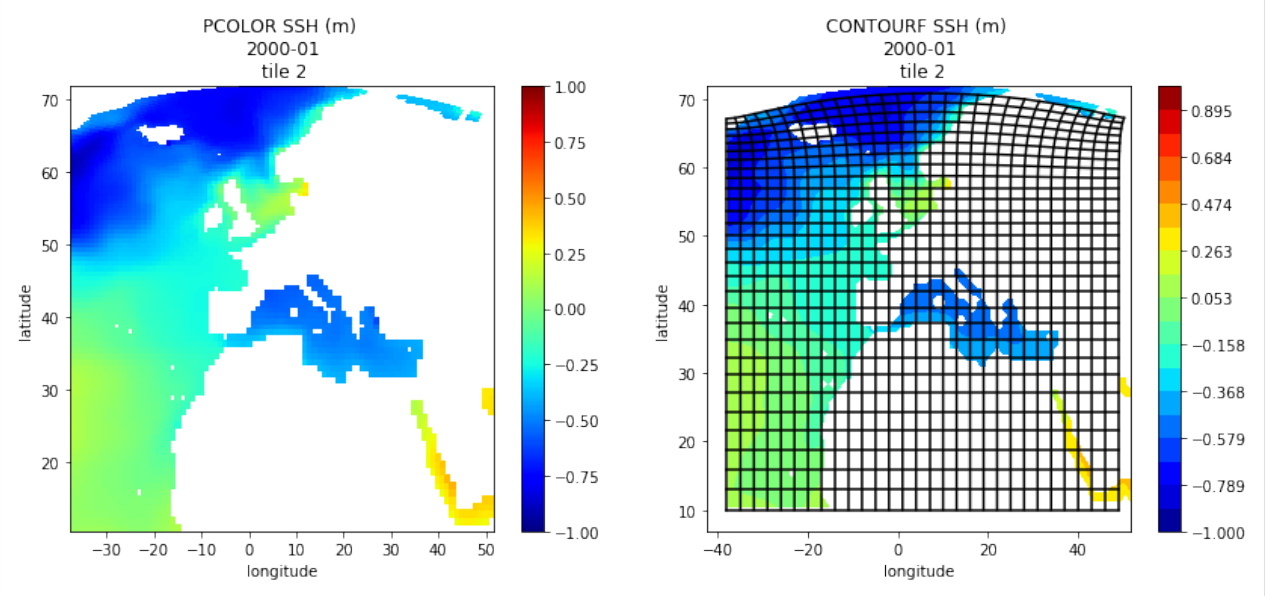
Tile 2 (Northeast N. Atlantic)

[9]:
```
fig=plt.figure(figsize=(10, 10))

tile_num=2

# pull out lats and lons
lons = ecco_ds.XC.sel(tile=tile_num)
lats = ecco_ds.YC.sel(tile=tile_num)
tile_to_plot = ecco_ds.SSH.isel(tile=tile_num, time=1)

# mask to NaN where hFacC is == 0
```

(continues on next page)

(continued from previous page)

```
# syntax is actually "keep where hFacC is not equal to zero"
tile_to_plot= tile_to_plot.where(ecco_ds.hFacC.isel(tile=tile_num,k=0) !=0, np.nan)

# create subplot for pcolor
fig = plt.subplot(221)

# use pcolor with 'lons' and 'lats' for the plot x and y axes
plt.pcolor(lons, lats, tile_to_plot, vmin=-1, vmax=1, cmap='jet')
plt.colorbar()
plt.title('PCOLOR SSH (m) \n' + str(ecco_ds.time[1].values)[0:7] + '\n tile ' + str(tile_
↪num))
plt.xlabel('longitude')
plt.ylabel('latitude')

# create subplot for contourf
fig=plt.subplot(222)

# use contourf with 'lons' and 'lats' for the plot x and y axes
plt.contourf(lons, lats, tile_to_plot, np.linspace(-1,1, 20,endpoint=True), cmap='jet',␣
↪vmin=-1, vmax=1)
plt.title('CONTOURF SSH (m) \n' + str(ecco_ds.time[1].values)[0:7] + '\n tile ' +␣
↪str(tile_num))
plt.xlabel('longitude')
plt.ylabel('latitude')
plt.colorbar()

# plot every 3rd model grid line to show how tile 3 is 'warped' above around 60N
plt.plot(ecco_ds.XG.isel(tile=tile_num)[::3,::3], ecco_ds.YG.isel(tile=tile_num)[::3,::
↪3],'k-')
plt.plot(ecco_ds.XG.isel(tile=tile_num)[::3,::3].T, ecco_ds.YG.isel(tile=tile_num)[::3,::
↪3].T,'k-')

# push the subplots away from each other a bit
plt.subplots_adjust(bottom=0, right=1.2, top=.9)
```



Tile 7 (N. Pacific / Bering Sea / Chukchi Sea)

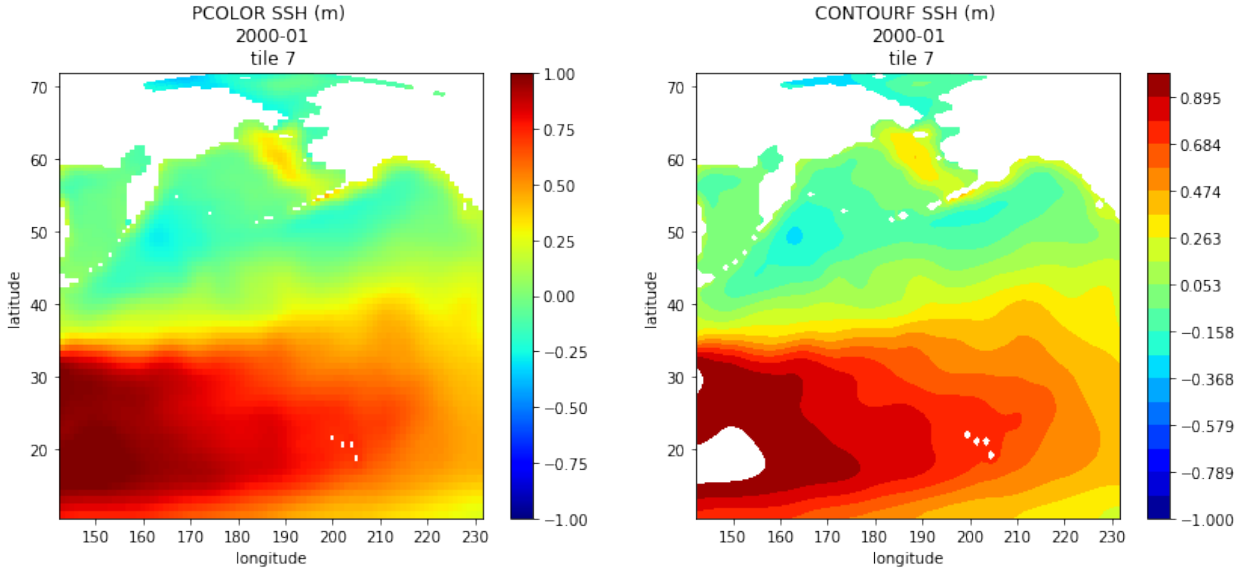
If longitude and latitude are passed as the 'x' and 'y' arguments to `pcolor` and `contourf` then the fields will be oriented geographically.

[10]:
```
fig=plt.figure(figsize=(10, 10))

tile_num=7

# pull out lats and lons
lons = np.copy(ecco_ds.XC.sel(tile=tile_num))

# we must convert the longitude coordinates from
# [-180 to 180] to [0 to 360]
# because of the crossing of the international date line.
lons[lons < 0] = lons[lons < 0]+360

lats = ecco_ds.YC.sel(tile=tile_num)
tile_to_plot = ecco_ds.SSH.isel(tile=tile_num, time=1)

# mask to NaN where hFacC is == 0
# syntax is actually "keep where hFacC is not equal to zero"
tile_to_plot= tile_to_plot.where(ecco_ds.hFacC.isel(tile=tile_num,k=0) !=0, np.nan)

# create subplot for pcolor
fig = plt.subplot(221)

# use pcolor with 'lons' and 'lats' for the plot x and y axes
plt.pcolor(lons, lats, tile_to_plot, vmin=-1, vmax=1, cmap='jet')
plt.colorbar()
plt.title('PCOLOR SSH (m) \n' + str(ecco_ds.time[1].values)[0:7] + '\n tile ' + str(tile_
↪num))
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('longitude')
plt.ylabel('latitude')

# create subplot for contourf
fig=plt.subplot(222)

# use contourf with 'lons' and 'lats' for the plot x and y axes
plt.contourf(lons, lats, tile_to_plot, np.linspace(-1,1, 20,endpoint=True), cmap='jet',␣
↪vmin=-1, vmax=1)
plt.title('CONTOURF SSH (m) \n' + str(ecco_ds.time[1].values)[0:7] + '\n tile ' +␣
↪str(tile_num))
plt.xlabel('longitude')
plt.ylabel('latitude')

plt.colorbar()

# push the subplots away from each other a bit
plt.subplots_adjust(bottom=0, right=1.2, top=.9)
```



1.15.5 Plotting fields from one tile using `Cartopy`

The `Cartopy` package provides routines to make plots using different geographic projections. We'll demonstrate plotting these three tiles again using *Cartopy*.

To see a list of `Cartopy` projections, see http://pelson.github.io/cartopy/crs/projections.html

Geographic Projections (AKA: plate carrée)

Cartopy works by transforming geographic coordintes (lat/lon) to new x,y coordinates associated with different projections. The most familiar projection is the so-called geographic projection (aka plate carree). When we plotted tiles using pcolor and contourf we were de-factor using the plate carree projection longitude and latitude were the 'x' and the 'y' of the plot.

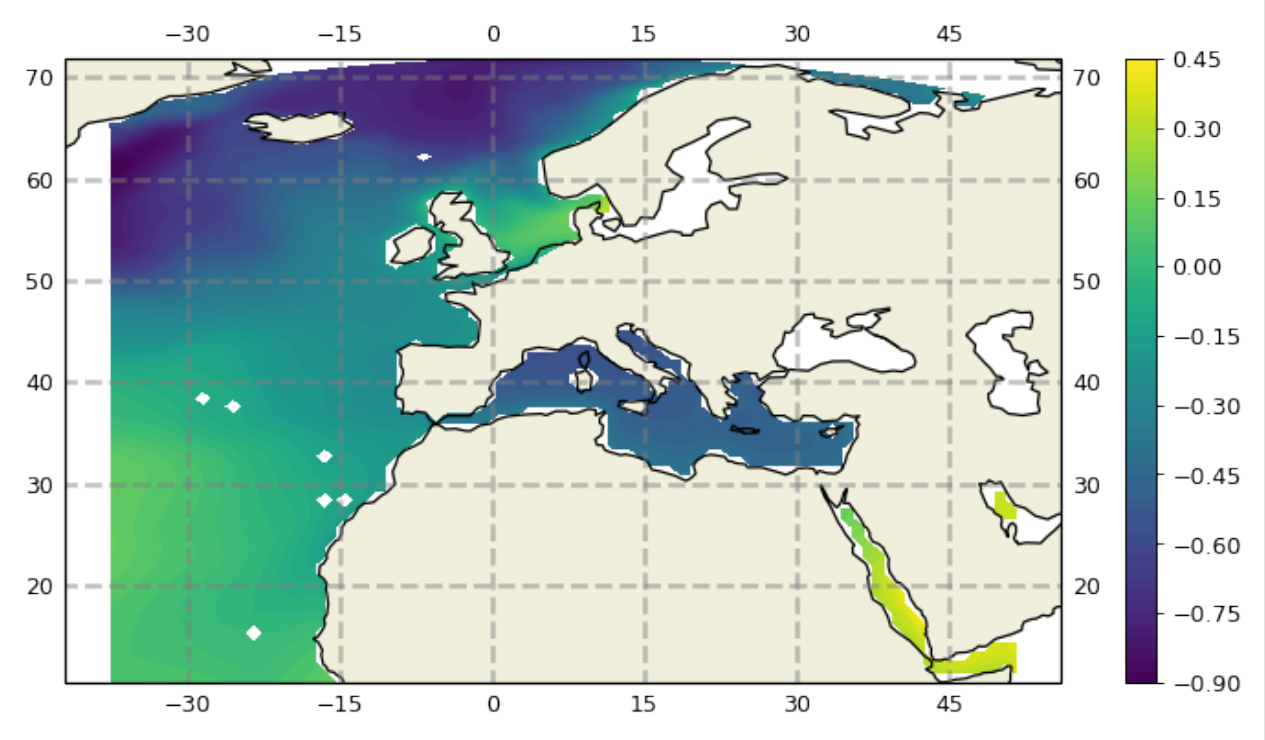
With Cartopy we can make similar plots in the plate carree projection system and also apply some cool extra details, like land masks.

We'll demonstrate on tiles 2 and 7 (again skipping tile 6 (Arctic cap) because we cannot use geographic coordinates as x and y when there is a polar singularity and 360 degrees of longitude.

Tile 2 with plate carree

[11]:
```
tile_num=2

lons = ecco_ds.XC.isel(tile=tile_num)
lats = ecco_ds.YC.isel(tile=tile_num)

tile_to_plot = ecco_ds.SSH.isel(tile=tile_num, time=1)
# mask to NaN where hFacC is == 0
# syntax is actually "keep where hFacC is not equal to zero"
tile_to_plot= tile_to_plot.where(ecco_ds.hFacC.isel(tile=tile_num,k=0) !=0, np.nan)

plt.figure(figsize=(10,5), dpi= 90)

# here is where you specify what projection you want to use
ax = plt.axes(projection=ccrs.PlateCarree())

# here is here you tell Cartopy that the projection
# of your 'x' and 'y' are geographic (lons and lats)
# and that you want to transform those lats and lons
# into 'x' and 'y' in the projection
plt.contourf(lons, lats, tile_to_plot, 60,
             transform=ccrs.PlateCarree());

gl = ax.gridlines(crs=ccrs.PlateCarree(), draw_labels=True,
                  linewidth=2, color='gray', alpha=0.5, linestyle='--');
ax.coastlines()
ax.add_feature(cfeature.LAND)

plt.colorbar()
```

[11]:
```
<matplotlib.colorbar.Colorbar at 0x7f6074546a20>
```



Other features we could have added include:

```
cartopy.feature.BORDERS
    Country boundaries.

cartopy.feature.COASTLINE
    Coastline, including major islands.

cartopy.feature.LAKES
    Natural and artificial lakes.

cartopy.feature.LAND
    Land polygons, including major islands.

cartopy.feature.OCEAN
    Ocean polygons.

cartopy.feature.RIVERS
    Single-line drainages, including lake centerlines.
```

Let's add geographic borders just to demonstrate how extra features can be added to a `Cartopy` map

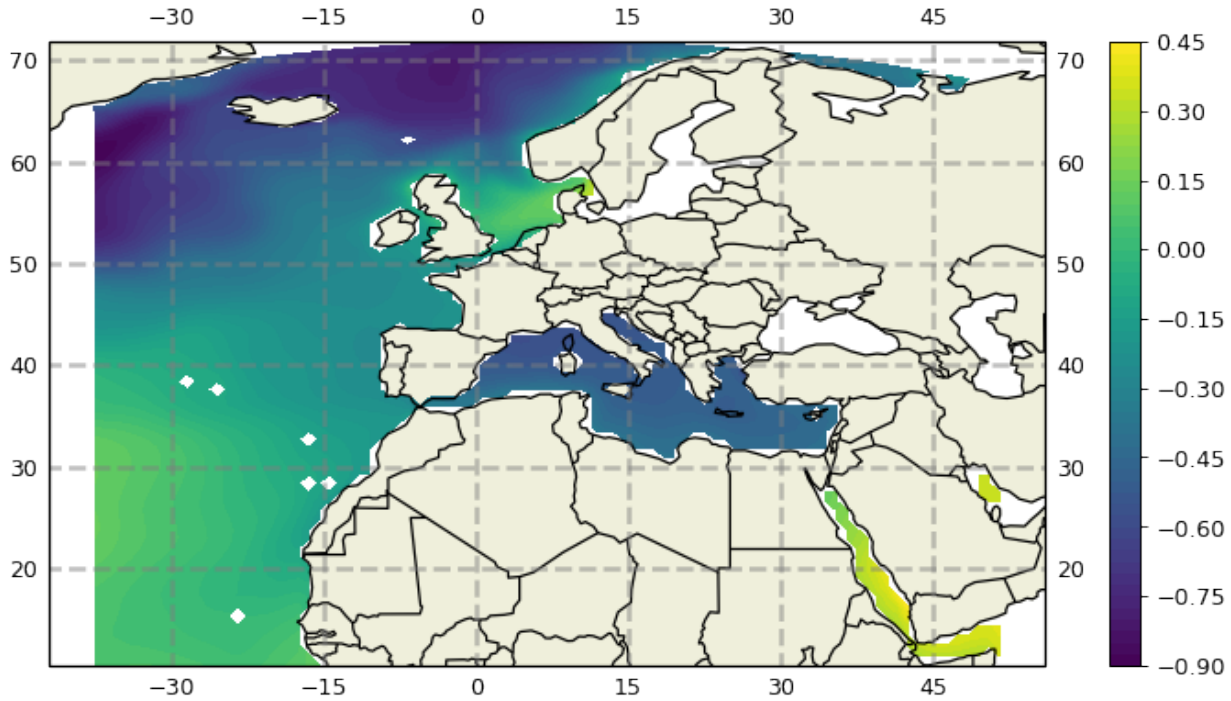
[12]:
```
plt.figure(figsize=(10,5), dpi= 90)

# here is where you specify what projection you want to use
ax = plt.axes(projection=ccrs.PlateCarree())

# here is here you tell Cartopy that the projection
# of your 'x' and 'y' are geographic (lons and lats)
```

(continues on next page)

(continued from previous page)

```
# and that you want to transform those lats and lons
# into 'x' and 'y' in the projection
plt.contourf(lons, lats, tile_to_plot, 60,
             transform=ccrs.PlateCarree());

gl = ax.gridlines(crs=ccrs.PlateCarree(), \
                  draw_labels=True,
                  linewidth=2, color='gray', \
                  alpha=0.5, linestyle='--');
ax.coastlines()
ax.add_feature(cfeature.LAND)
ax.add_feature(cfeature.BORDERS)
plt.colorbar()
```

[12]:
```
<matplotlib.colorbar.Colorbar at 0x7f6074626e10>
```



Tile 7 with plate carree

To use the plate carree projection across the international date line specify the *central_longitude=-180* argument when defining the projection and for creating the gridlines. (see https://stackoverflow.com/questions/13856123/setting-up-a-map-which-crosses-the-dateline-in-cartopy)

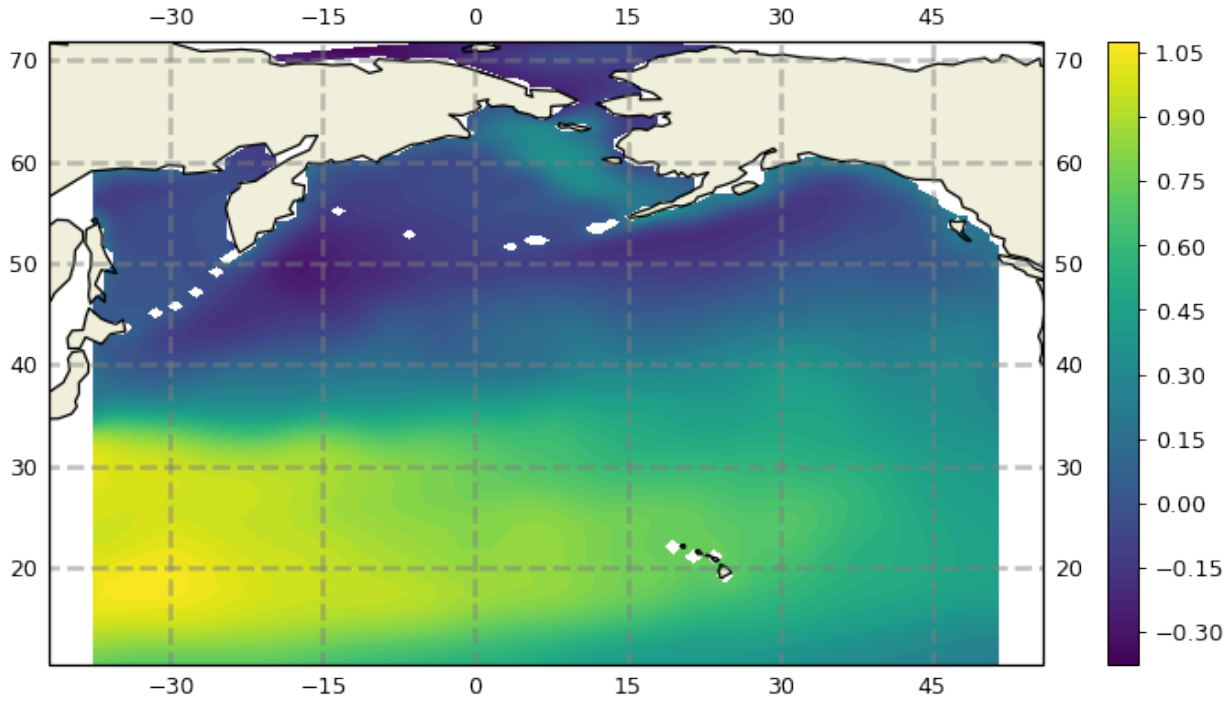
[13]:
```
tile_num=7

# pull out lats and lons
lons = np.copy(ecco_ds.XC.sel(tile=tile_num))

# we must convert the longitude coordinates from
```

(continues on next page)

(continued from previous page)

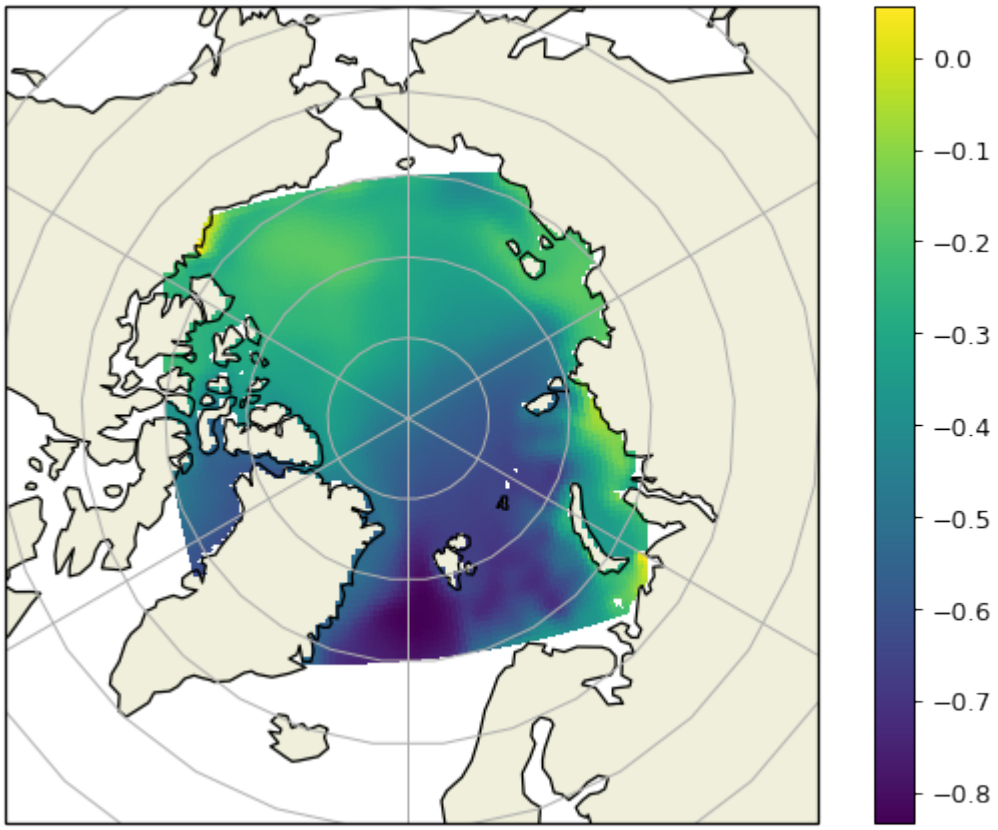
```
# [-180 to 180] to [0 to 360]
# because of the crossing of the international date line.
lons[lons < 0] = lons[lons < 0]+360
lats = ecco_ds.YC.isel(tile=tile_num)

tile_to_plot = ecco_ds.SSH.isel(tile=tile_num, time=1)
# mask to NaN where hFacC is == 0
# syntax is actually "keep where hFacC is not equal to zero"
tile_to_plot= tile_to_plot.where(ecco_ds.hFacC.isel(tile=tile_num,k=0) !=0, np.nan)

plt.figure(figsize=(10,5), dpi= 90)

# here is where you specify what projection you want to use
ax = plt.axes(projection=ccrs.PlateCarree(central_longitude=-180))

# here is here you tell Cartopy that the projection of your 'x' and 'y' are geographic␣
↪(lons and lats)
# and that you want to transform those lats and lons into 'x' and 'y' in the projection
plt.contourf(lons, lats, tile_to_plot, 60,
             transform=ccrs.PlateCarree())

gl = ax.gridlines(crs=ccrs.PlateCarree(central_longitude=-180), draw_labels=True,
                  linewidth=2, color='gray', alpha=0.5, linestyle='--');
ax.coastlines()
ax.add_feature(cfeature.LAND)
plt.colorbar()
```

[13]: <matplotlib.colorbar.Colorbar at 0x7f607c07bb70>



Polar Stereographic Projection

[14]:
```
tile_num=6

lons = ecco_ds.XC.isel(tile=tile_num)
lats = ecco_ds.YC.isel(tile=tile_num)


tile_to_plot = ecco_ds.SSH.isel(tile=tile_num, time=1)

# mask to NaN where hFacC is == 0
# syntax is actually "keep where hFacC is not equal to zero"
tile_to_plot = tile_to_plot.where(ecco_ds.hFacC.isel(tile=tile_num,k=0) !=0, \
                      np.nan)

plt.figure(figsize=(8,6), dpi= 90)


# Make a new projection, time of class "NorthPolarStereo"
ax = plt.axes(projection=ccrs.NorthPolarStereo(true_scale_latitude=70))

# here is here you tell Cartopy that the projection
# of your 'x' and 'y' are geographic (lons and lats)
# and that you want to transform those lats and lons
#into 'x' and 'y' in the projection
plt.pcolormesh(lons, lats, tile_to_plot,
               transform=ccrs.PlateCarree());

# plot land
ax.add_feature(cfeature.LAND)
ax.gridlines()
ax.coastlines()
plt.colorbar()

# Limit the map to -60 degrees latitude and below.
ax.set_extent([-180, 180, 90, 60], ccrs.PlateCarree())

# Compute a circle in axes coordinates, which we can use as a boundary
# for the map. We can pan/zoom as much as we like - the boundary will be
# permanently circular.
theta = np.linspace(0, 2*np.pi, 100)
center, radius = [0.5, 0.5], 0.5
verts = np.vstack([np.sin(theta), np.cos(theta)]).T
circle = mpath.Path(verts * radius + center)

#ax.set_boundary(circle, transform=ax.transAxes)
```



1.15.6 Plotting all 13 tiles simultaneously: No Projection

Plotting all 13 tiles with `plot_tiles`

The `plot_tiles` routine in the `ecco_v4_py` package makes plots of all 13 tiles of a field. By default the routine will plot all of the tiles in the lat-lon-cap layout shown earlier.

This routine will accept `numpy arrays` of dimension 13x90x90 or 2D slices of DataArrays with the same 13x90x90 dimension.

There are several additional arguments which we can access using the `help` command. Take a second to familiarize yourself with some of them.

[15]:
```
help(ecco.plot_tiles)
```

```
Help on function plot_tiles in module ecco_v4_py.tile_plot:

plot_tiles(tiles, cmap='jet', layout='llc', rotate_to_latlon=False, Arctic_cap_tile_
↪location=2, show_colorbar=False, show_cbar_label=False, show_tile_labels=True, cbar_
↪label='', fig_size=9, **kwargs)
    Plots the 13 tiles of the lat-lon-cap (LLC) grid

    Parameters
    ----------
    tiles : numpy.ndarray or dask.array.core.Array or xarray.core.dataarray.DataArray
```

(continues on next page)

(continued from previous page)

```
        an array of n=1..13 tiles of dimension n x llc x llc

            - If *xarray DataArray* or *dask Array* tiles are accessed via *tiles.
↪sel(tile=n)*
            - If *numpy ndarray* tiles are acceed via [tile,:,:] and thus n must be 13.

    cmap : matplotlib.colors.Colormap, optional, default: jet
        a colormap for the figure

    layout : string, optional, default 'llc'
        a code indicating the layout of the tiles

        :llc:    situates tiles in a fan-like manner which conveys how the tiles
                 are oriented in the model in terms of x an y

        :latlon: situates tiles in a more geographically recognizable manner.
                 Note, this does not rotate tiles 7..12, it just places tiles
                 7..12 adjacent to tiles 0..5.  To rotate tiles 7..12
                 specifiy *rotate_to_latlon* as True

    rotate_to_latlon : boolean, default False
        rotate tiles 7..12 so that columns correspond with
        longitude and rows correspond to latitude.  Note, this rotates
        vector fields (vectors positive in x in tiles 7..12 will be -y
        after rotation).

    Arctic_cap_tile_location : int, default 2
        integer, which lat-lon tile to place the Arctic tile over. can be
        2, 5, 7 or 10.

    show_colorbar : boolean, optional, default False
        add a colorbar

    show_cbar_label : boolean, optional, default False
        add a label on the colorbar

    show_tile_labels : boolean, optional, default True
        show tiles numbers in subplot titles

    cbar_label : str, optional, default '' (empty string)
        the label to use for the colorbar

    less_output : boolean, optional, default False
        A debugging flag.  False = less debugging output

    cmin/cmax : floats, optional, default calculate using the min/max of the data
        the minimum and maximum values to use for the colormap

    fig_size : float, optional, default 9 inches
        size of the figure in inches

    fig_num : int, optional, default none
```

(continues on next page)

(continued from previous page)

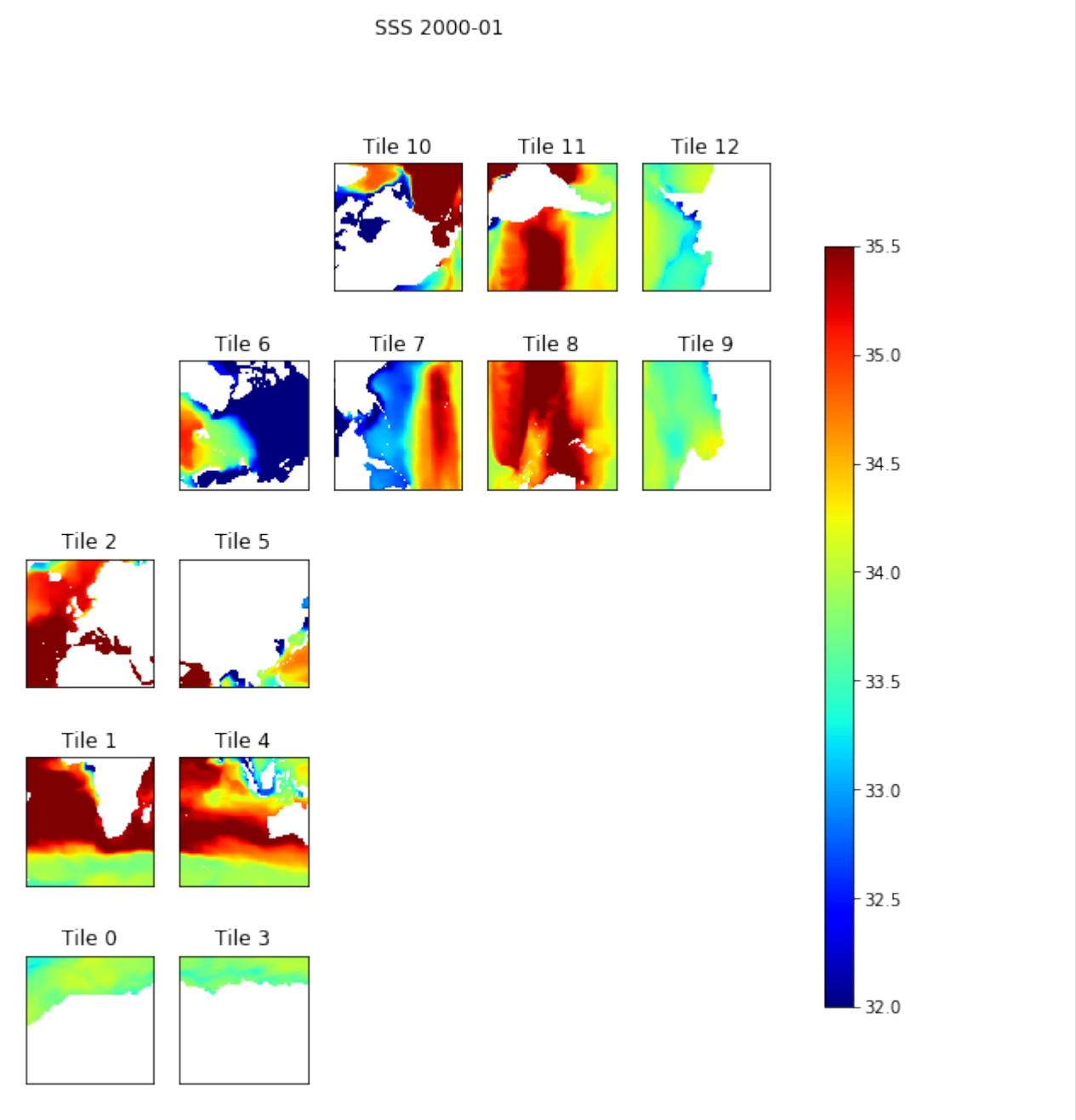
```
        the figure number to make the plot in.  By default make a new figure.

    Returns
    -------
    f : Figure
```

We've seen this routine used in a few earlier routines. We'll provide some additional examples below:

Default 'native grid' layout

[16]:
```
# optional arguments:
#   cbar       - show the colorbar
#   cmin, cmax - color range min and max
#   fsize      - figure size in inches

# pull out surface salinity
tmp_plt = ecco_ds.SALT.isel(time=3)

# mask to NaN where hFacC is == 0
# syntax is actually "keep where hFacC is not equal to zero"
tmp_plt = tmp_plt.where(ecco_ds.hFacC.isel(k=0) != 0, np.nan)
```

[17]:
```
ecco.plot_tiles(tmp_plt, \
                cmin=32, \
                cmax=35.5, \
                show_colorbar=True);

# use `suptitle` (super title) to make a title over subplots.
plt.suptitle('SSS ' + str(ecco_ds.time[3].values)[0:7]);
```



lat-lon layout

Another option of `plot_tiles` is to show tiles 7-12 rotated and lined up tiles 0-5

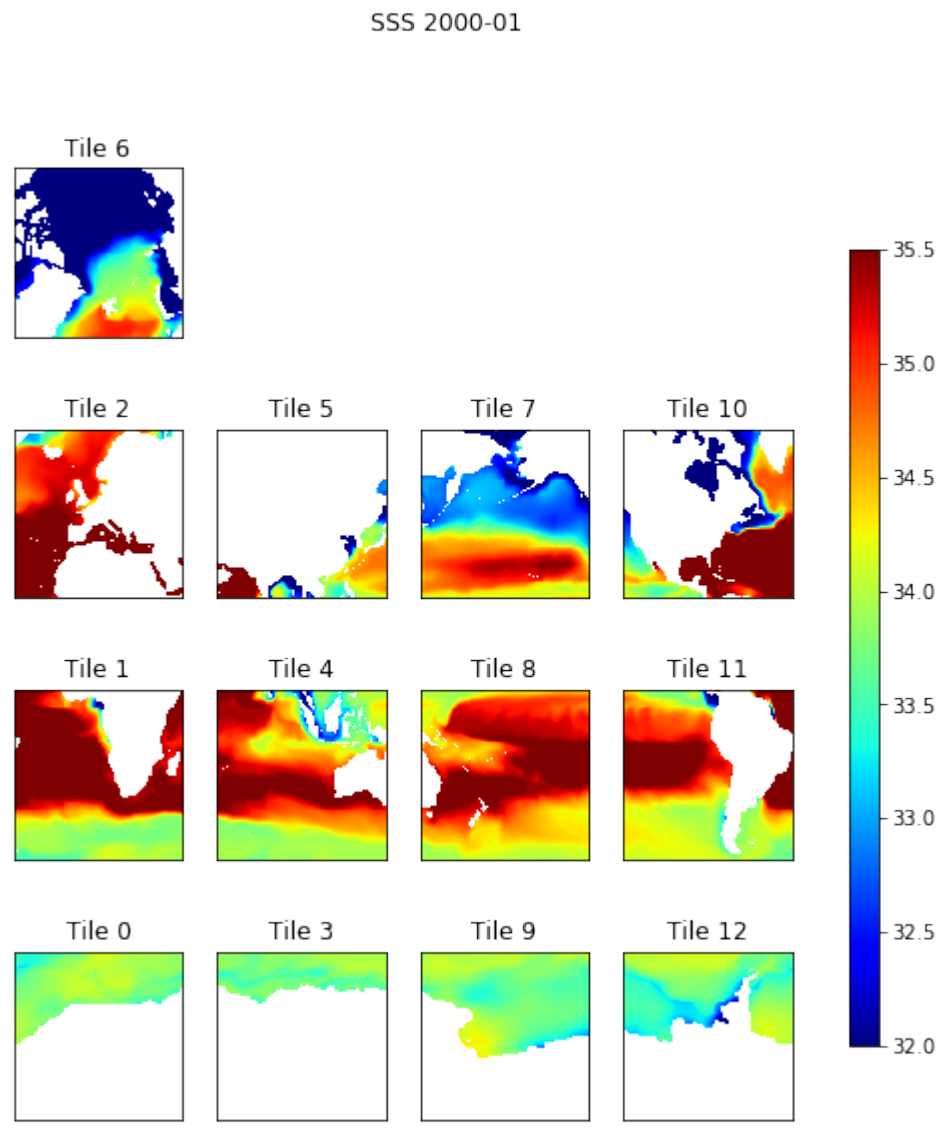
> **Note:** *Rotation of tiles 7-13 is only for* **plotting**. *These arrays are not rotated using this routine. We'll show to how actually rotate these tiles in a later tutorial.*

[18]:
```
# optional arguments:
#   cbar       - show the colorbar
#   cmin, cmax - color range min and max
```

(continues on next page)

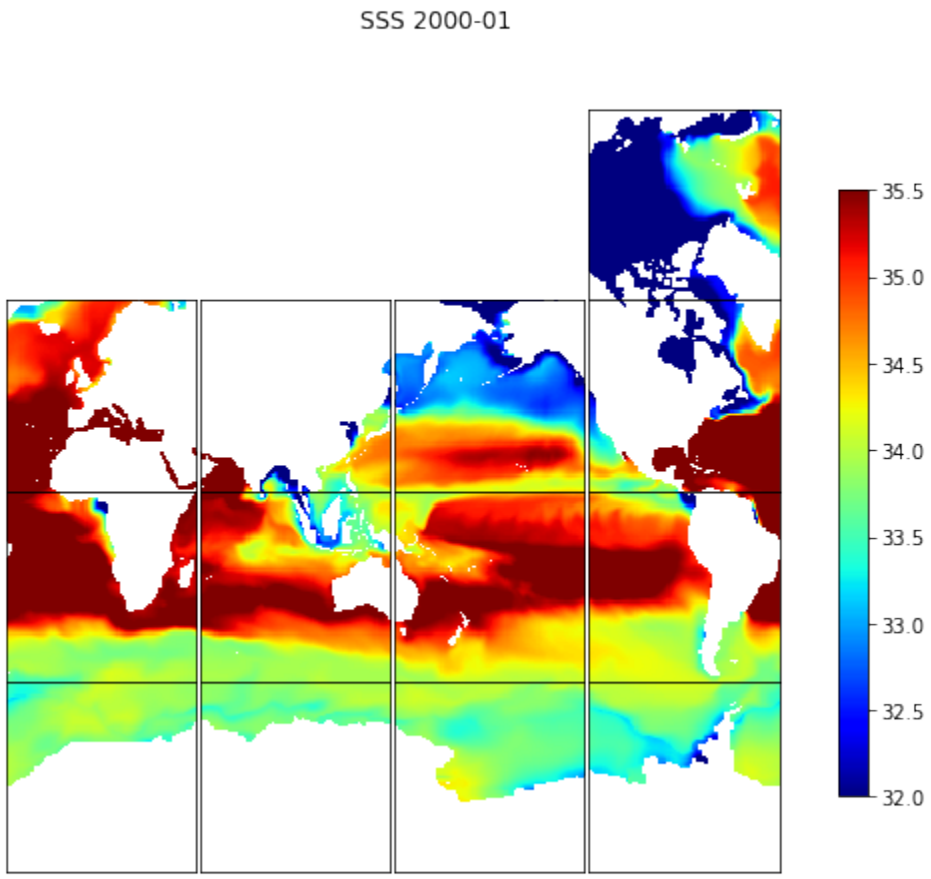
(continued from previous page)

```
#   fsize      - figure size in inches

tmp_plt = ecco_ds.SALT.isel(time=3)
tmp_plt = tmp_plt.where(ecco_ds.hFacC.isel(k=0) != 0, np.nan)
ecco.plot_tiles(tmp_plt, \
                cmin=32, cmax=35.5, \
                show_colorbar=True, fig_size=8,\
                layout='latlon', \
                rotate_to_latlon=True);

# use `suptitle` (super title) to make a title over subplots.
plt.suptitle('SSS ' + str(ecco_ds.time[3].values)[0:7]);
```



The version of `plot_tiles` is to remove the tile labels and put the titles together in a tight formation and sticks the Arctic tile over tile 10

[19]:
```
# optional arguments:
#   cbar       - show the colorbar
#   cmin, cmax - color range min and max
#   fsize      - figure size in inches

tmp_plt = ecco_ds.SALT.isel(time=3)
tmp_plt = tmp_plt.where(ecco_ds.hFacC.isel(k=0) != 0, np.nan)
ecco.plot_tiles(tmp_plt, cmin=32, cmax=35.5, \
                show_colorbar=True, fig_size=8,\
               layout='latlon',rotate_to_latlon=True,\
               show_tile_labels=False, \
               Arctic_cap_tile_location=10)

# use `suptitle` (super title) to make a title over subplots.
plt.suptitle('SSS ' + str(ecco_ds.time[3].values)[0:7]);
```



Almost ready for the hyperwall!

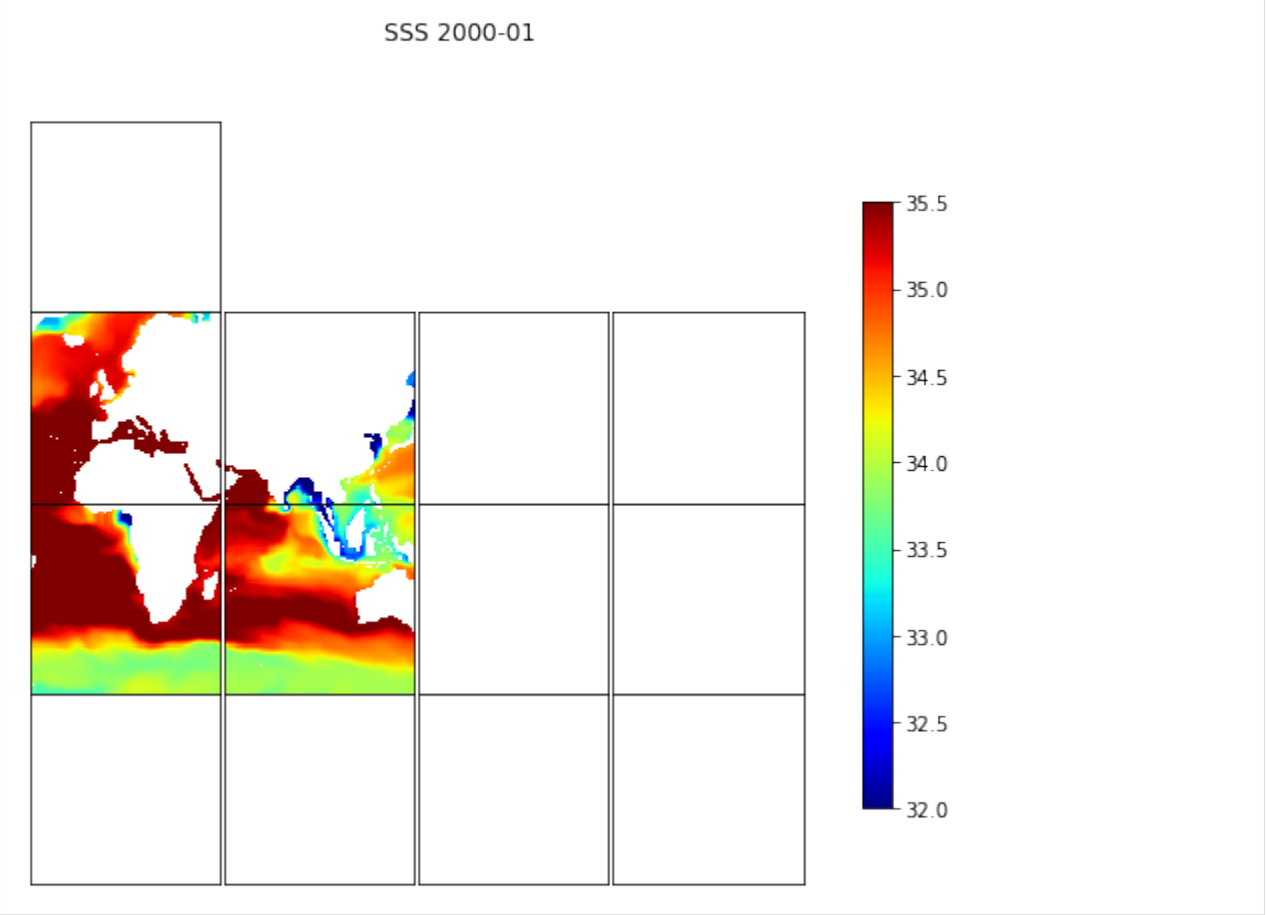
You can even pass `plot_tiles` a subset tiles from DataArray objects provided that the DataArrays have a 'tile' dimension.

[20]:
```
# optional arguments:
#   cbar       - show the colorbar
#   cmin, cmax - color range min and max
#   fsize      - figure size in inches
```

(continues on next page)

(continued from previous page)

```
tmp_plt = ecco_ds.SALT.isel(time=3)
tmp_plt = tmp_plt.where(ecco_ds.hFacC.isel(k=0) != 0, np.nan)

# select a subset of tiles
ecco.plot_tiles(tmp_plt.isel(tile=[1,2,4,5]), cmin=32, cmax=35.5, \
                show_colorbar=True, fig_size=8,\
                layout='latlon',rotate_to_latlon=True,\
                show_tile_labels=False)

# use `suptitle` (super title) to make a title over subplots.
plt.suptitle('SSS ' + str(ecco_ds.time[3].values)[0:7]);
```



1.15.7 Plotting all 13 tiles with `plot_proj_to_latlon_grid`

Our routine `plot_proj_to_latlon_grid` takes `numpy` arrays or `DataArrays` with 13 tiles and creates global plots with one of three types of projections (passed as arguments to the function): ~~~ projection_type : string, optional denote the type of projection, options include 'robin' - Robinson 'PlateCaree' - flat 2D projection 'Mercator' 'cyl' - Lambert Cylindrical 'ortho' - Orthographic 'stereo' - polar stereographic projection, see lat_lim for choosing 'InterruptedGoodeHomolosi ~~~

Before plotting this routine interpolates the the filed onto a lat-lon grid (default resoution 0.25 degree) to conform with `Cartopy's` requirement that the fields to be transformed be on regular square grid.

There are only three argument required of `plot_proj_to_latlon_grid`, an array of longitudes, an array of latitudes, and an array of the field you wish to plot. The arrays can be either numpy arrays or `DataArrays`.

Let's again spend a second to look at the optional arguments available to us in this routine:

[21]:
```
help(ecco.plot_proj_to_latlon_grid)
```

```
Help on function plot_proj_to_latlon_grid in module ecco_v4_py.tile_plot_proj:

plot_proj_to_latlon_grid(lons, lats, data, projection_type='robin', plot_type='pcolormesh
↪', user_lon_0=0, lat_lim=50, levels=20, cmap='jet', dx=0.25, dy=0.25, show_
↪colorbar=False, show_grid_lines=True, show_grid_labels=True, grid_linewidth=1, grid_
↪linestyle='--', subplot_grid=None, less_output=True, custom_background=False,␣
↪background_name=[], background_resolution=[], radius_of_influence=100000, **kwargs)
    Generate a plot of llc data, resampled to lat/lon grid, on specified
    projection.

    Parameters
    ----------
    lons, lats, data : xarray DataArray    :
        give the longitude, latitude values of the grid, and the 2D field to
        be plotted
    projection_type : string, optional
        denote the type of projection, see Cartopy docs.
        options include
            'robin' - Robinson
            'PlateCaree' - flat 2D projection
            'Mercator'
            'EqualEarth'
            'AlbersEqualArea'
            'cyl' - Lambert Cylindrical
            'ortho' - Orthographic
            'stereo' - polar stereographic projection, see lat_lim for choosing
            'InterruptedGoodeHomolosine'
                North or South
    user_lon_0 : float, optional, default 0 degrees
        denote central longitude
    lat_lim : int, optional
        for stereographic projection, denote the Southern (Northern) bounds for
        North (South) polar projection
    levels : int, optional
        number of contours to plot
    cmap : string or colormap object, optional
        denotes to colormap
```

(continues on next page)

(continued from previous page)

```
    dx, dy : float, optional
        latitude, longitude spacing for grid resampling
    show_colorbar : logical, optional, default False
        show a colorbar or not,
    show_grid_lines : logical, optional
        True only possible for Mercator or PlateCarree projections
    grid_linewidth : float, optional, default 1.0
        width of grid lines
    grid_linestyle : string, optional, default = '--'
        pattern of grid lines,
    cmin, cmax : float, optional
        minimum and maximum values for colorbar, default is min/max of data
    subplot_grid : dict or list, optional
        specifying placement on subplot as
            dict:
                {'nrows': rows_val, 'ncols': cols_val, 'index': index_val}

            list:
                [nrows_val, ncols_val, index_val]

            equates to

                matplotlib.pyplot.subplot(
                    row=nrows_val, col=ncols_val,index=index_val)
    less_output : string, optional
        debugging flag, don't print if True
    raidus_of_influence : float, optional.  Default 100000 m
        the radius of the circle within which the input data is search for
        when mapping to the new grid
```

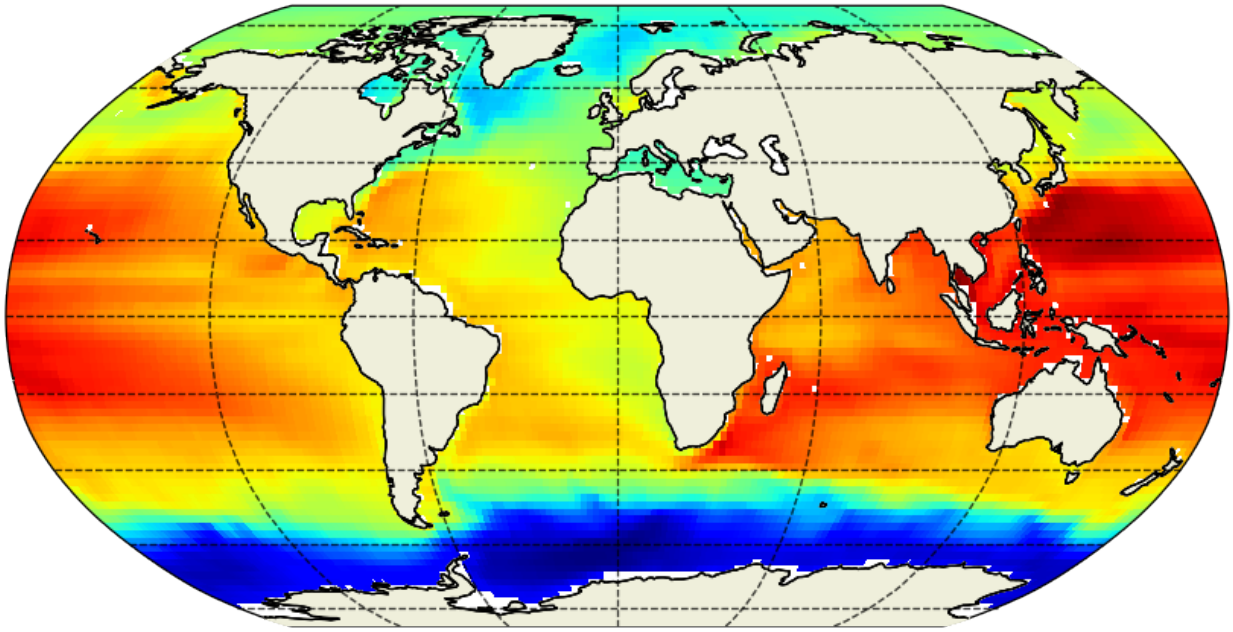
plot_proj_to_latlon_grid(lons, lats, data, projection_type='robin', plot_type='pcolormesh', user_lon_0=-66, lat_lim=50, levels=20, cmap='jet', dx=0.25, dy=0.25, show_colorbar=False, show_grid_lines=True, show_grid_labels=True, subplot_grid=None, less_output=True, **kwargs)

Robinson projection

First we'll demonstrate the Robinson projection interpolated to a 2x2 degree grid

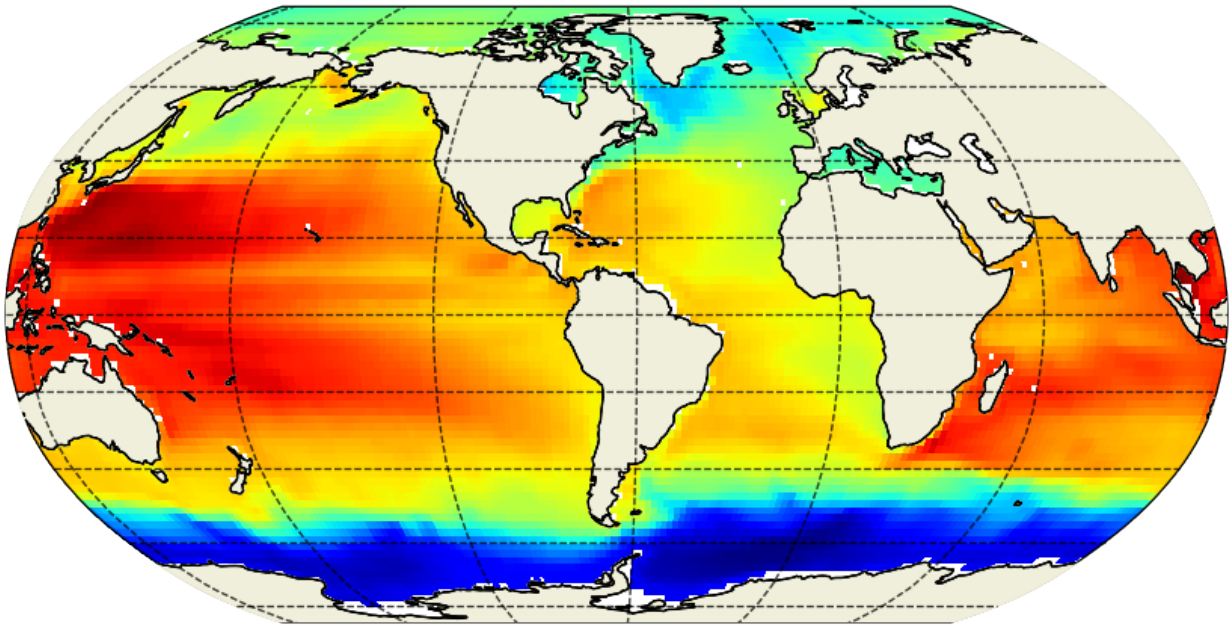
[22]:
```
plt.figure(figsize=(12,6), dpi= 90)

tmp_plt = ecco_ds.SSH.isel(time=1)
tmp_plt = tmp_plt.where(ecco_ds.hFacC.isel(k=0) !=0)

ecco.plot_proj_to_latlon_grid(ecco_ds.XC, \
                              ecco_ds.YC, \
                              tmp_plt, \
                              plot_type = 'pcolormesh', \
                              dx=2,\
                              dy=2, \
                              projection_type = 'robin',\
                              less_output = False);
```

```
-180 < user_lon_0 < 180
Projection type:  robin
len(lon_tmp_d):  2
```



Setting *lon_0* = 110 or -66 yield a global centering that is is more usesful for plotting ocean basins.

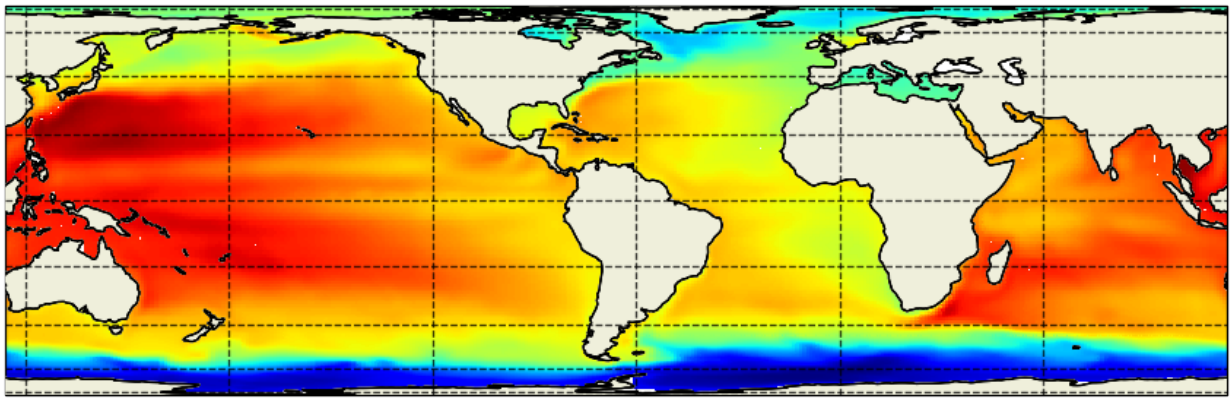
[23]:
```
plt.figure(figsize=(12,6), dpi= 90)

tmp_plt = ecco_ds.SSH.isel(time=1)
tmp_plt = tmp_plt.where(ecco_ds.hFacC.isel(k=0) !=0)

ecco.plot_proj_to_latlon_grid(ecco_ds.XC,
                 ecco_ds.YC,
                 tmp_plt,user_lon_0=-66,
                 plot_type = 'pcolormesh', dx=2,dy=2);
```



Cylindrical projection

Try the Cylindrical Projection with an interpolated lat-lon resolution of 0.25 degrees and pcolormesh.

[24]:
```
plt.figure(figsize=(12,6), dpi= 90)

tmp_plt = ecco_ds.SSH.isel(time=1)
tmp_plt = tmp_plt.where(ecco_ds.hFacC.isel(k=0) !=0)

ecco.plot_proj_to_latlon_grid(ecco_ds.XC, ecco_ds.YC, \
                              tmp_plt, \
                              user_lon_0=-66,\
                              projection_type='cyl',\
                              plot_type = 'pcolormesh', \
                              dx=.25,dy=.25);
```

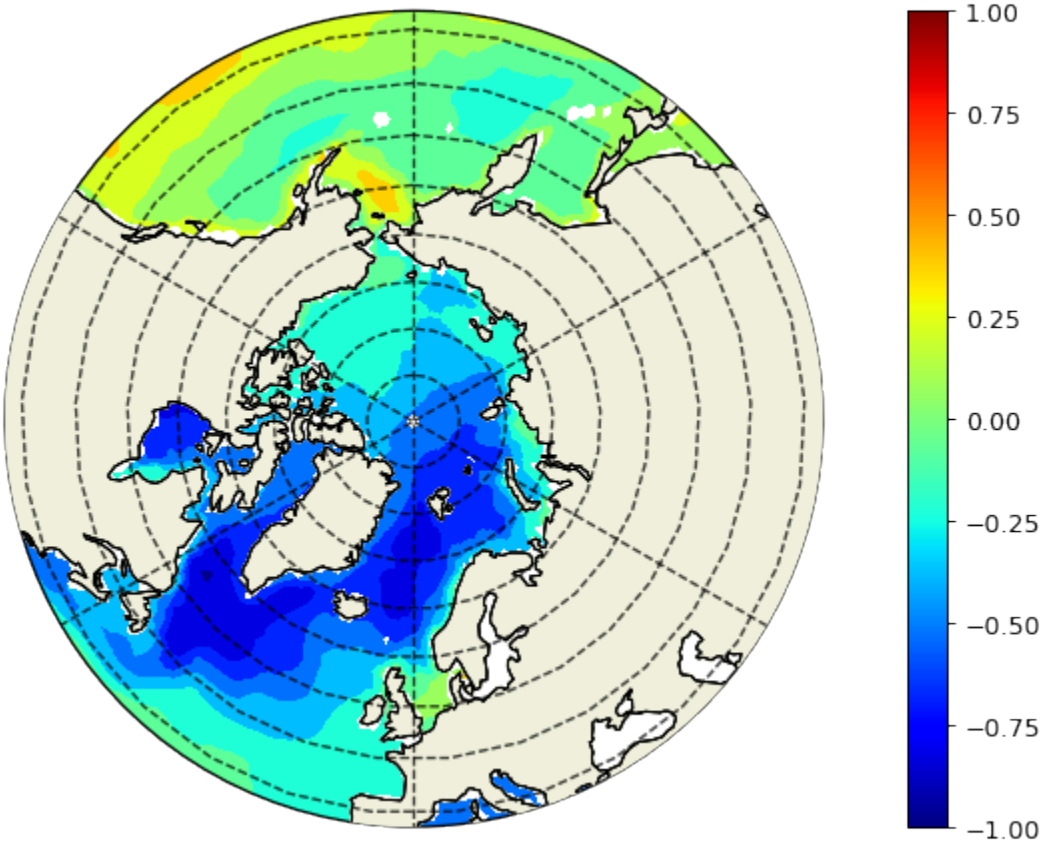


Polar stereographic projection

Another isefi; projection built into `plot_proj_to_latlon_grid` is polar stereographic. The argument *lat_lim* determines the limit of this type of projection. If *lat_lim* is postive, the projection is centered around the north pole and vice versa.

Northern Hemisphere

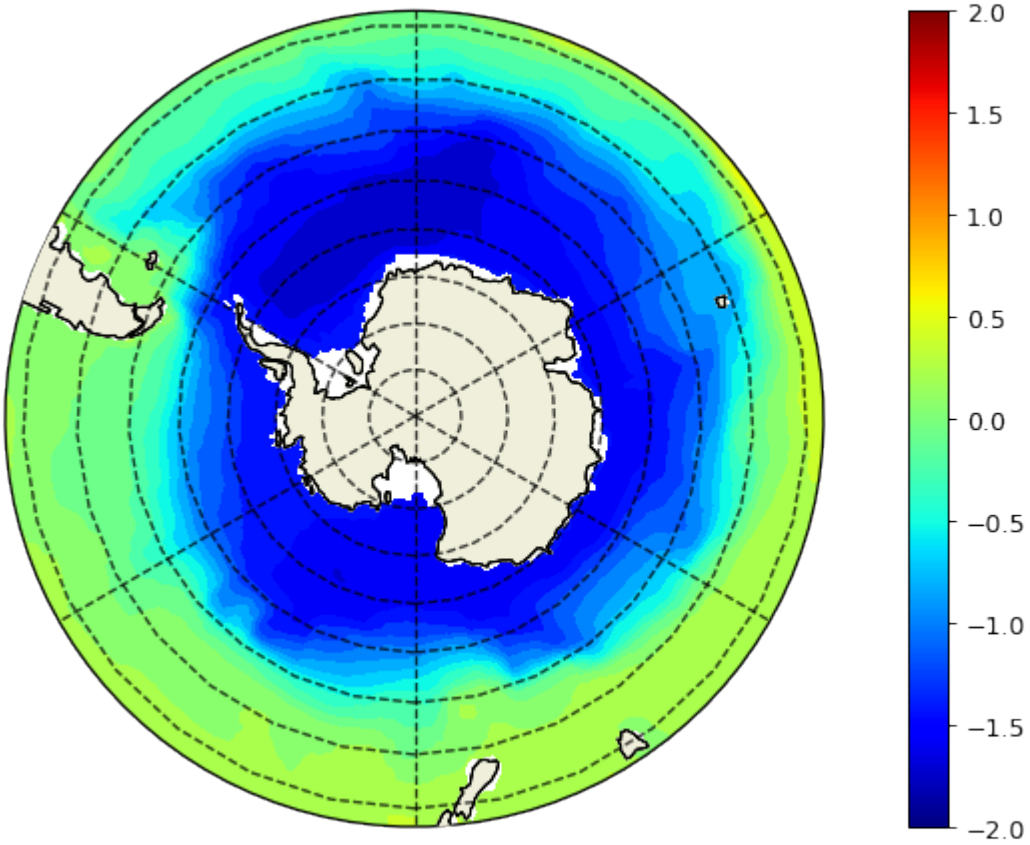
[25]:
```
plt.figure(figsize=(12,6), dpi= 90)

tmp_plt = ecco_ds.SSH.isel(time=1)
tmp_plt = tmp_plt.where(ecco_ds.hFacC.isel(k=0) !=0)

ecco.plot_proj_to_latlon_grid(ecco_ds.XC, ecco_ds.YC, \
                              tmp_plt, \
                              projection_type='stereo',\
                              plot_type = 'contourf', \
                              show_colorbar=True,
                              dx=1, dy=1,cmin=-1, cmax=1,\
                              lat_lim=40);
```



Southern Hemisphere

The final example is a south-pole centered plot. Note that *lat_lim* is now negative.

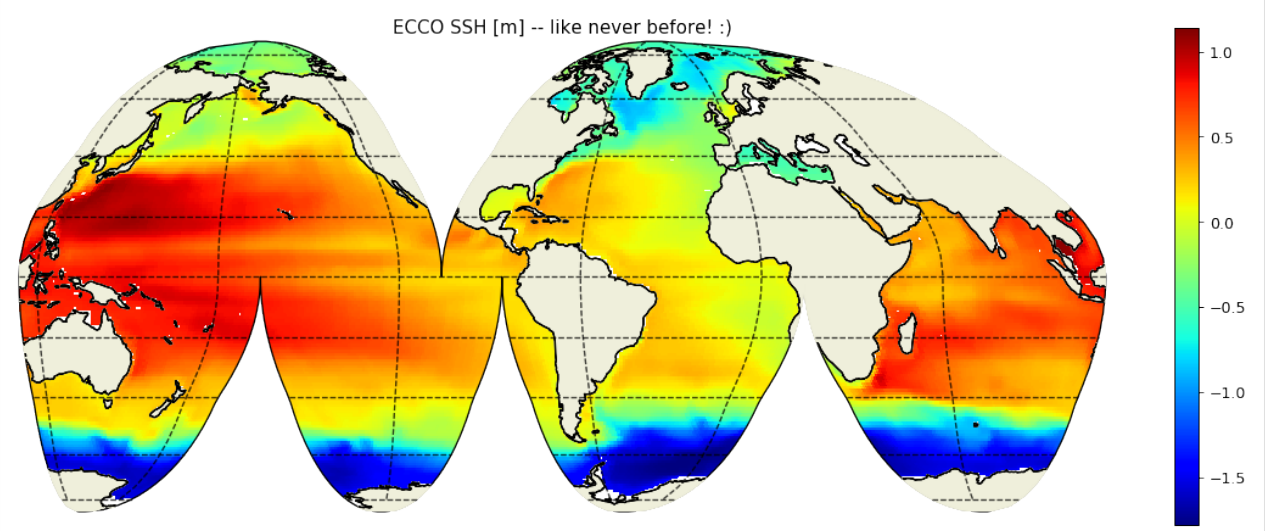
[26]:
```
plt.figure(figsize=(12,6), dpi= 90)

tmp_plt = ecco_ds.SSH.isel(time=1)
tmp_plt = tmp_plt.where(ecco_ds.hFacC.isel(k=0) !=0)

ecco.plot_proj_to_latlon_grid(ecco_ds.XC, ecco_ds.YC, \
                              tmp_plt, \
                              projection_type='stereo',\
                              plot_type = 'contourf', \
                              show_colorbar=True,
                              dx=1, dy=1,\
                              lat_lim=-40,cmin=-2,cmax=2);
```



1.15.8 Conclusion

You now know several ways of plotting ECCO state estimate fields. There is a lot more to explore with `Cartopy` - dive in and start making your own cool plots!

[27]:
```
plt.figure(figsize=(16,6), dpi=90)

tmp_plt = ecco_ds.SSH.isel(time=1)
tmp_plt = tmp_plt.where(ecco_ds.hFacC.isel(k=0) !=0)

ecco.plot_proj_to_latlon_grid(ecco_ds.XC, ecco_ds.YC, \
                              tmp_plt, \
                              user_lon_0=-66,\
                              projection_type='InterruptedGoodeHomolosine',\
                              plot_type = 'pcolormesh', \
                              show_colorbar=True,
                              dx=1, dy=1);

plt.title('ECCO SSH [m] -- like never before! :)');
```



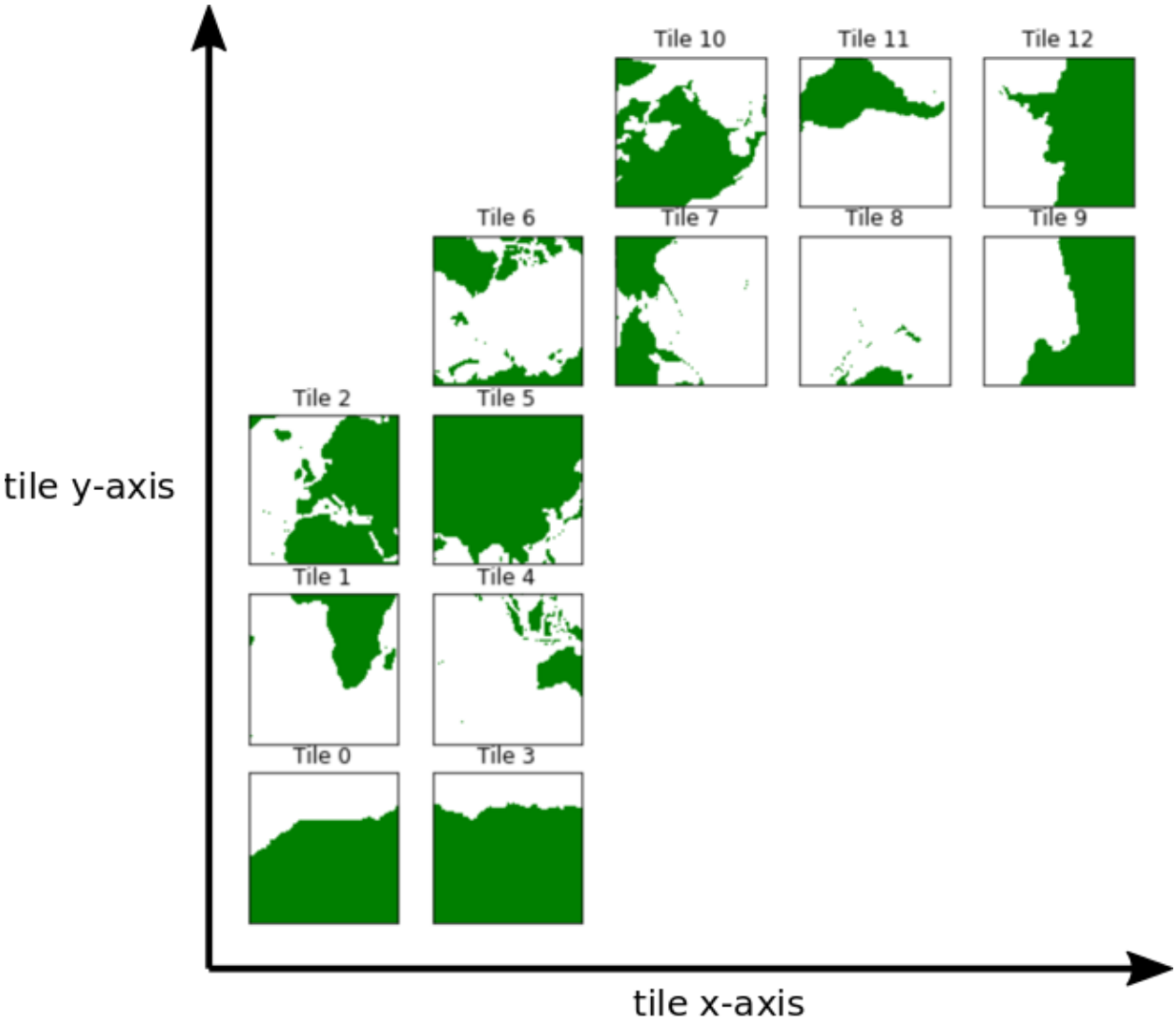
1.16 Interpolating fields from the model llc grid to a regular lat lon grid

1.16.1 Objectives

1. Learn how to interpolate scalar and vector fields from ECCOv4's lat-lon-cap 90 (llc90) model grid to the more common regular latitude-longitude grid.
2. Learn how to save these interpolated fields as netCDF for later analysis

1.16.2 Introduction

Recall the orientations of the 13 tiles of the ECCOv4 native llc90 model grid.



Tiles 7-12 are rotated 90 degrees counter-clockwise relative to tiles 0-5.

In this tutorial we demonstrate two methods for mapping scalar and vector fields from the llc90 model grid to "regular" latitude-longitude grids of arbitrary resolution.

> **Note:** *There are many methods on can use to map between the grids (e.g., nearest neighbor, bilinear interpolation, bin-averaging, etc.), each with its own advantages.)*

1.16.3 How to interpolate scalar ECCO fields to a lat-lon grid

Scalar fields are the most straightforward fields to interpolate.

Preliminaries: Load fields

First, let's load the all 13 tiles for sea surface height and the model grid parameters.

[1]:
```
import numpy as np
import sys
import xarray as xr
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')
from pprint import pprint
import importlib
```

[2]:
```
## Import the ecco_v4_py library into Python
## =========================================

## -- If ecco_v4_py is not installed in your local Python library,
##    tell Python where to find it.  For example, if your ecco_v4_py
##    files are in /Users/ifenty/ECCOv4-py/ecco_v4_py, then use:

from pathlib import Path
sys.path.append(str(Path('c:/Users/Ian/ECCOv4-py')))
import ecco_v4_py as ecco
```

[3]:
```
## Set top-level file directory for the ECCO NetCDF files
## ======================================================================
# base_dir = homehome/username/'
ECCO_dir = Path('E:/inSync Share/Projects/ECCOv4/Release4/')
```

[4]:
```
## Load the model grid
grid_dir= ECCO_dir / 'nctiles_grid/'
```

[5]:
```
## Load the model grid
ecco_grid = ecco.load_ecco_grid_nc(grid_dir, 'ECCO-GRID.nc')
```

[6]:
```
## Load one year of 2D daily data, SSH, SST, and SSS
day_mean_dir= ECCO_dir / 'nctiles_monthly/'

ecco_vars = ecco.recursive_load_ecco_var_from_years_nc(day_mean_dir, \
                                        vars_to_load=['SSH'], \
                                        years_to_load=2000,less_output=True)

ecco_ds = []

## Merge the ecco_grid with the ecco_vars to make the ecco_ds
ecco_ds = xr.merge((ecco_grid , ecco_vars)).load()
```

(continues on next page)

(continued from previous page)

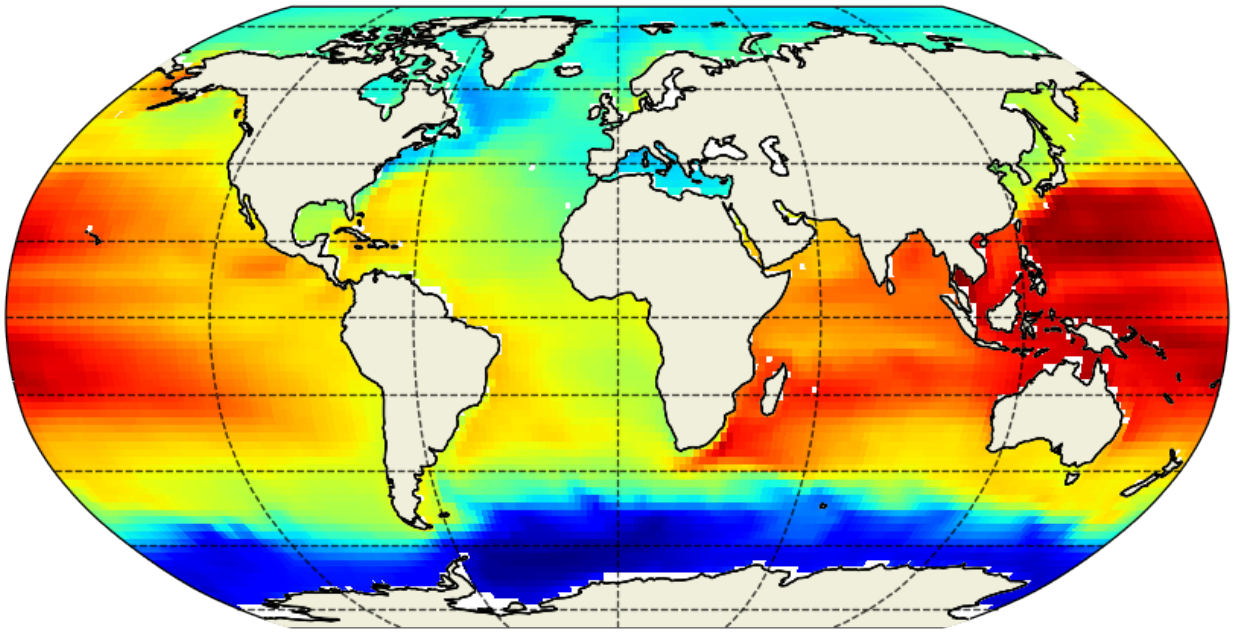
```
pprint(ecco_ds.data_vars)
```

```
loading files of  SSH
Data variables:
    SSH      (time, tile, j, i) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
```

Plotting the dataset

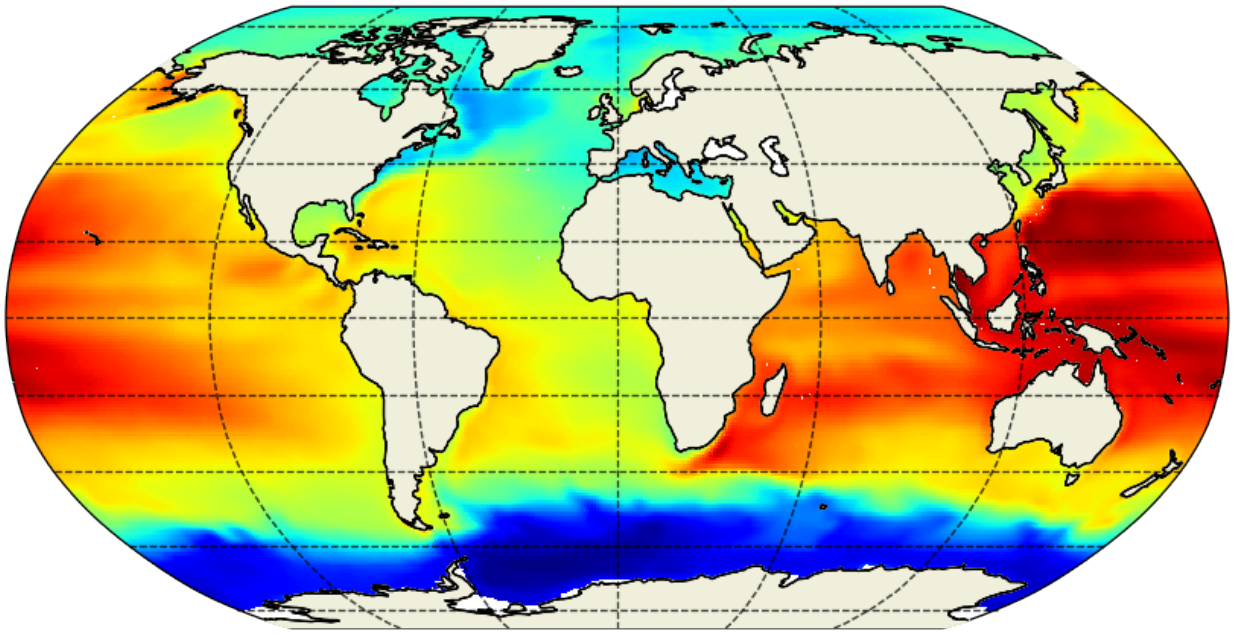
Plotting the ECCOv4 fields was covered in an earlier tutorial. Before demonstrating interpolation, we will first plot one of our SSH fields. Take a closer look at the arguments of `plot_proj_to_latlon_grid`, `dx=2` and `dy=2`.

[7]:
```
plt.figure(figsize=(12,6), dpi= 90)

tmp_plt = ecco_ds.SSH.isel(time=1)
tmp_plt = tmp_plt.where(ecco_ds.hFacC.isel(k=0) !=0)

ecco.plot_proj_to_latlon_grid(ecco_ds.XC, \
                              ecco_ds.YC, \
                              tmp_plt, \
                              plot_type = 'pcolormesh', \
                              dx=2,\
                              dy=2, \
                              projection_type = 'robin',\
                              less_output = True);
```



These `dx` and `dy` arguments tell the plotting function to interpolate the native grid tiles onto a lat-lon grid with spacing of `dx` degrees longitude and `dy` degrees latitude. If we reduced `dx` and `dy` the resulting map would have finer features: Compare with the map when `dx=dy=0.25` degrees:

[8]:
```
plt.figure(figsize=(12,6), dpi= 90)

tmp_plt = ecco_ds.SSH.isel(time=1)
tmp_plt = tmp_plt.where(ecco_ds.hFacC.isel(k=0) !=0)

ecco.plot_proj_to_latlon_grid(ecco_ds.XC, \
                              ecco_ds.YC, \
                              tmp_plt, \
                              plot_type = 'pcolormesh', \
                              dx=0.25,\
                              dy=0.25, \
                              projection_type = 'robin',\
                              less_output = True);
```



Of course you can interpolate to arbitrarily high resolution lat-lon grids, the model resolution will ultimately determine the smallest resolvable features.

Under the hood of `plot_proj_to_lat_longrid` is a call to the very useful routine `resample_to_latlon` which is the now described in more detail

1.16.4 resample_to_latlon

`resample_to_latlon` takes a field with a cooresponding set of lat lon coordinates (the *source* grid) and interpolates to a new lat-lon *target* grid. The arrangement of coordinates in the *source* grid is arbitrary. One also provides the *bounds* of the new lat lon grid. The example shown above uses -90..90N and -180..180E by default. In addition, one specifies which interpolation scheme to use (*mapping method*) and the *radius of influence*, the radius around the *target* grid cells within which to search for values from the *source* grid.

[9]:
```
help(ecco.resample_to_latlon)
```

```
Help on function resample_to_latlon in module ecco_v4_py.resample_to_latlon:

resample_to_latlon(orig_lons, orig_lats, orig_field, new_grid_min_lat, new_grid_max_lat,␣
↪new_grid_delta_lat, new_grid_min_lon, new_grid_max_lon, new_grid_delta_lon, radius_of_
↪influence=120000, fill_value=None, mapping_method='bin_average')
    Take a field from a source grid and interpolate to a target grid.

    Parameters
    ----------
    orig_lons, orig_lats, orig_field : xarray DataArray or numpy array  :
        the lons, lats, and field from the source grid

        new_grid_min_lat, new_grid_max_lat : float
                latitude limits of new lat-lon grid

    new_grid_delta_lat : float
        latitudinal extent of new lat-lon grid cells in degrees (-90..90)

    new_grid_min_lon, new_grid_max_lon : float
                longitude limits of new lat-lon grid (-180..180)

    new_grid_delta_llon : float
         longitudinal extent of new lat-lon grid cells in degrees

    radius_of_influence : float, optional.  Default 120000 m
        the radius of the circle within which the input data is search for
        when mapping to the new grid

    fill_value : float, optional. Default None
                value to use in the new lat-lon grid if there are no valid values
                from the source grid

        mapping_method : string, optional. Default 'bin_average'
        denote the type of interpolation method to use.
        options include
            'nearest_neighbor' - Take the nearest value from the source grid
                                                to the target grid
            'bin_average'      - Use the average value from the source grid
                                                                to the target grid

    RETURNS:
    new_grid_lon, new_grid_lat : ndarrays
        2D arrays with the lon and lat values of the new grid

    data_latlon_projection:
        the source field interpolated to the new grid
```

1.16.5 Demonstrations of `resample_to_latlon`

Global

First we will map to a global lat-lon grid at 1degree using nearest neighbor

[10]:
```
new_grid_delta_lat = 1
new_grid_delta_lon = 1

new_grid_min_lat = -90+new_grid_delta_lat/2
new_grid_max_lat = 90-new_grid_delta_lat/2

new_grid_min_lon = -180+new_grid_delta_lon/2
new_grid_max_lon = 180-new_grid_delta_lon/2

new_grid_lon, new_grid_lat, field_nearest_1deg =\
        ecco.resample_to_latlon(ecco_ds.XC, \
                                ecco_ds.YC, \
                                ecco_ds.SSH.isel(time=0),\
                                new_grid_min_lat, new_grid_max_lat, new_grid_delta_lat,\
                                new_grid_min_lon, new_grid_max_lon, new_grid_delta_lon,\
                                fill_value = np.NaN, \
                                mapping_method = 'nearest_neighbor',
                                radius_of_influence = 120000)
```

[11]:
```
# Dimensions of the new grid:
pprint(new_grid_lat.shape)
pprint(new_grid_lon.shape)
```

```
(180, 360)
(180, 360)
```

[12]:
```
pprint(new_grid_lon)

# the second dimension of new_grid_lon has the center longitude of the new grid cells
pprint(new_grid_lon[0,0:10])
```

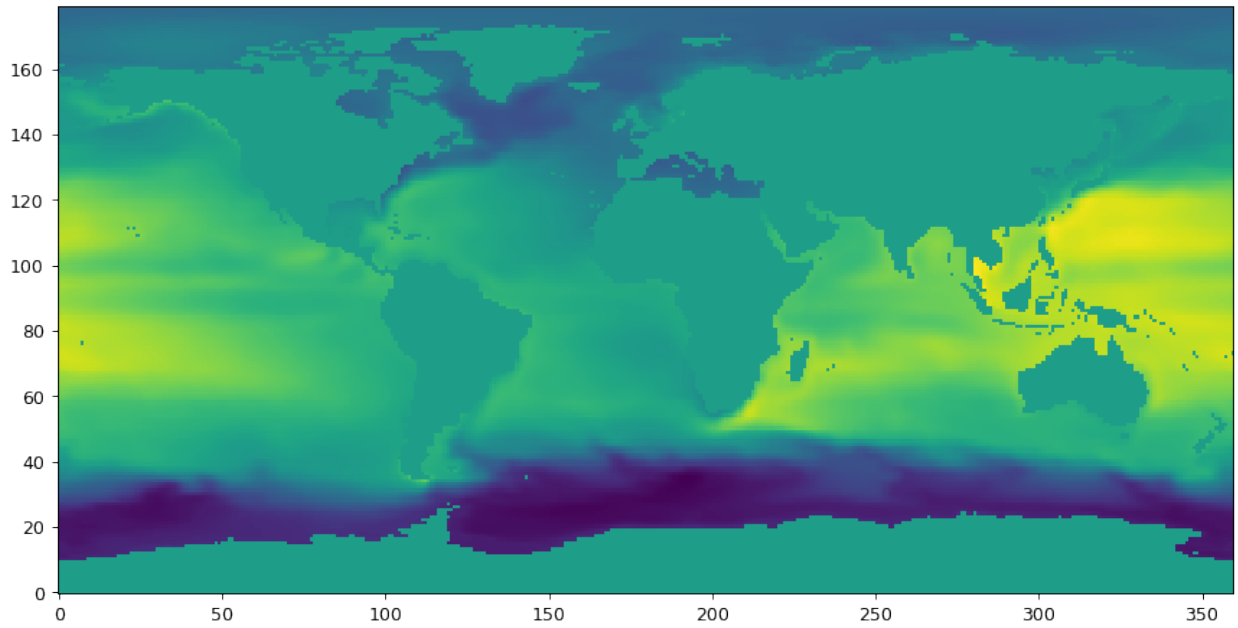
```
array([[-179.5, -178.5, -177.5, ...,  177.5,  178.5,  179.5],
       [-179.5, -178.5, -177.5, ...,  177.5,  178.5,  179.5],
       [-179.5, -178.5, -177.5, ...,  177.5,  178.5,  179.5],
       ...,
       [-179.5, -178.5, -177.5, ...,  177.5,  178.5,  179.5],
       [-179.5, -178.5, -177.5, ...,  177.5,  178.5,  179.5],
       [-179.5, -178.5, -177.5, ...,  177.5,  178.5,  179.5]])
array([-179.5, -178.5, -177.5, -176.5, -175.5, -174.5, -173.5, -172.5,
       -171.5, -170.5])
```

[13]:
```
# The set of lat points
pprint(new_grid_lat)

# or as a 1D vector
# the first dimension of new_grid_lat has the center latitude of the new grid cells
pprint(new_grid_lat[0:10,0])
```

```
array([[-89.5, -89.5, -89.5, ..., -89.5, -89.5, -89.5],
       [-88.5, -88.5, -88.5, ..., -88.5, -88.5, -88.5],
       [-87.5, -87.5, -87.5, ..., -87.5, -87.5, -87.5],
       ...,
       [ 87.5,  87.5,  87.5, ...,  87.5,  87.5,  87.5],
       [ 88.5,  88.5,  88.5, ...,  88.5,  88.5,  88.5],
       [ 89.5,  89.5,  89.5, ...,  89.5,  89.5,  89.5]])
array([-89.5, -88.5, -87.5, -86.5, -85.5, -84.5, -83.5, -82.5, -81.5,
       -80.5])
```

[14]:
```
# plot the whole field
plt.figure(figsize=(12,6), dpi= 90)
plt.imshow(field_nearest_1deg,origin='lower')
```

[14]:
```
<matplotlib.image.AxesImage at 0x1ff70b5f9a0>
```



Notice that although we specified a fill_value of np.nan, the values on land are zero. This is because we did not first mask out original field with nans over dry points. Let's nan out the land points first and interpolate again:

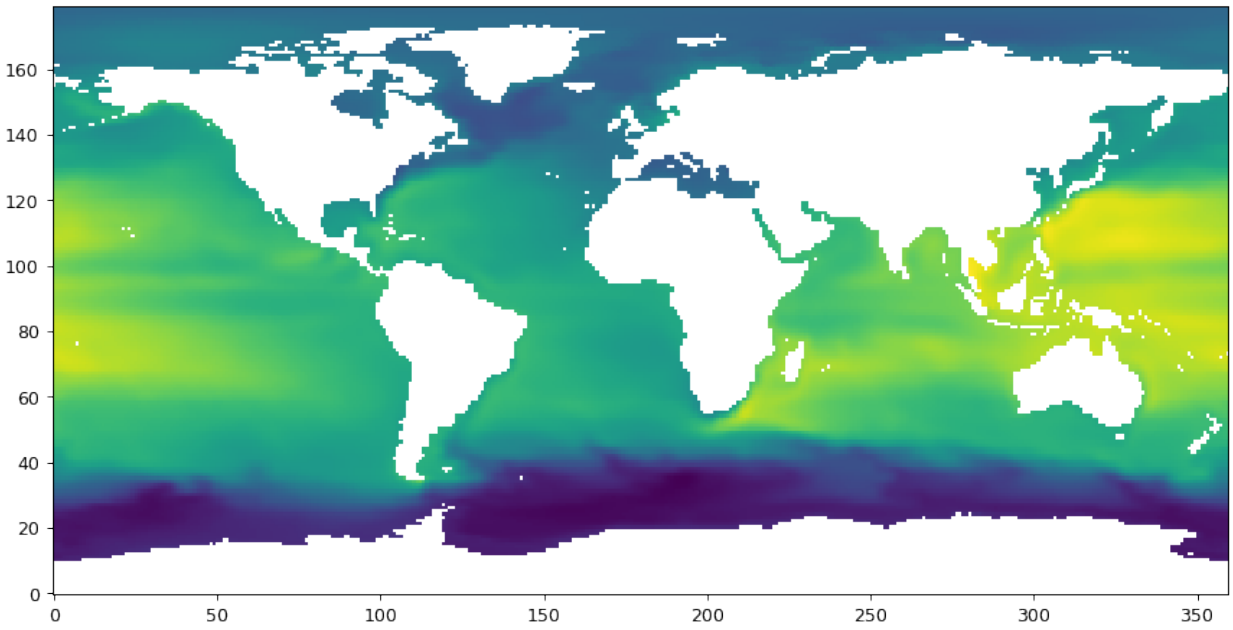
[15]:
```
original_field_with_land_mask= np.where(ecco_ds.maskC.isel(k=0)>0, ecco_ds.SSH.
↪isel(time=0), np.nan)

new_grid_delta_lat = 1
new_grid_delta_lon = 1

new_grid_min_lat = -90+new_grid_delta_lat/2
new_grid_max_lat = 90-new_grid_delta_lat/2

new_grid_min_lon = -180+new_grid_delta_lon/2
new_grid_max_lon = 180-new_grid_delta_lon/2
new_grid_lon, new_grid_lat, field_nearest_1deg =\
        ecco.resample_to_latlon(ecco_ds.XC, \
```

(continues on next page)

(continued from previous page)

```
                                ecco_ds.YC, \
                                original_field_with_land_mask,\
                                new_grid_min_lat, new_grid_max_lat, new_grid_delta_lat,\
                                new_grid_min_lon, new_grid_max_lon, new_grid_delta_lon,\
                                fill_value = np.NaN, \
                                mapping_method = 'nearest_neighbor',
                                radius_of_influence = 120000)

# plot the whole field, this time land values are nans.
plt.figure(figsize=(12,6), dpi= 90)
plt.imshow(field_nearest_1deg,origin='lower')
```

[15]:
```
<matplotlib.image.AxesImage at 0x1ff70989280>
```



Regional

1 degree, nearest neighbor

First we'll interpolate only to a subset of the N. Atlantic at 1 degree.

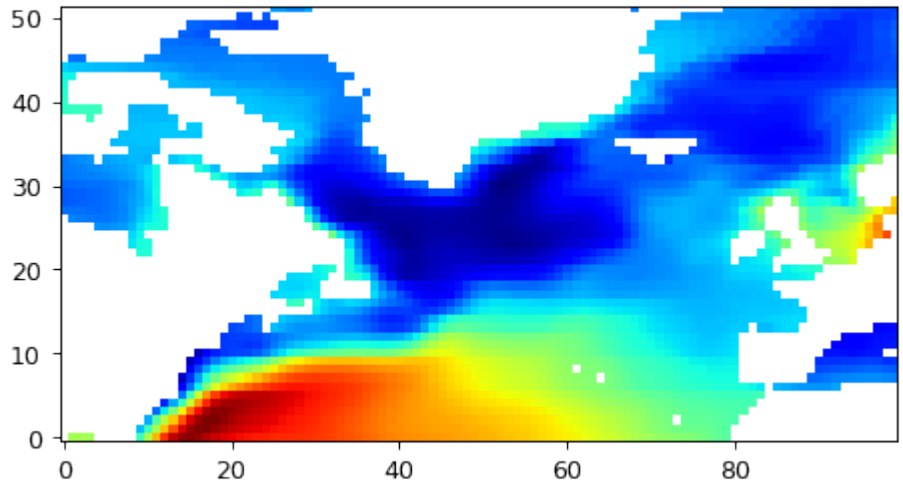
[16]:
```
original_field_with_land_mask= np.where(ecco_ds.maskC.isel(k=0)>0, ecco_ds.SSH.
↪isel(time=0), np.nan)

new_grid_delta_lat = 1
new_grid_delta_lon = 1

new_grid_min_lat = 30+new_grid_delta_lat/2
new_grid_max_lat = 82-new_grid_delta_lat/2

new_grid_min_lon = -90+new_grid_delta_lon/2
```

(continues on next page)

(continued from previous page)

```
new_grid_max_lon = 10-new_grid_delta_lon/2

new_grid_lon, new_grid_lat, field_nearest_1deg =\
        ecco.resample_to_latlon(ecco_ds.XC, \
                                ecco_ds.YC, \
                                original_field_with_land_mask,\
                                new_grid_min_lat, new_grid_max_lat, new_grid_delta_lat,\
                                new_grid_min_lon, new_grid_max_lon, new_grid_delta_lon,\
                                fill_value = np.NaN, \
                                mapping_method = 'nearest_neighbor',
                                radius_of_influence = 120000)

# plot the whole field, this time land values are nans.
plt.figure(figsize=(6,6), dpi= 90)
plt.imshow(field_nearest_1deg,origin='lower',cmap='jet')
```

[16]:
```
<matplotlib.image.AxesImage at 0x1ff708d1ca0>
```



0.05 degree, nearest neighbor

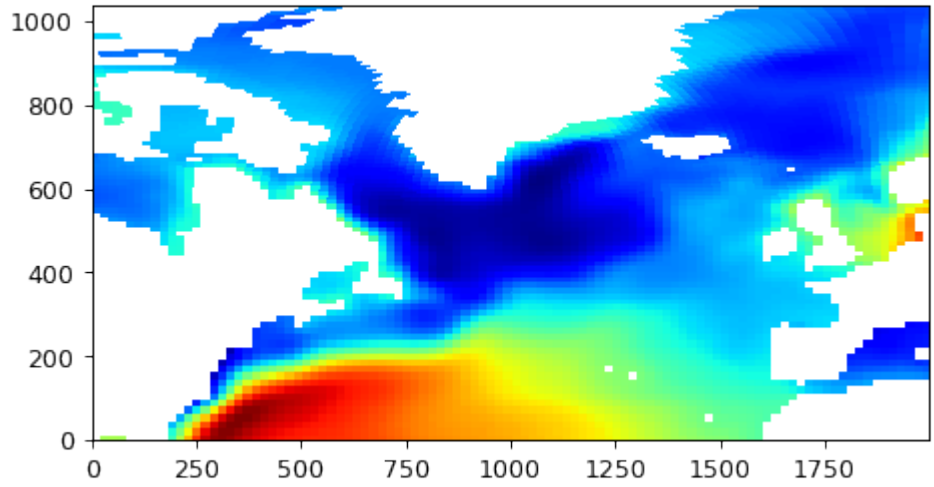
Next we'll interpolate the same domain at a much higher resolution, 0.05 degree:

[17]:
```
original_field_with_land_mask= np.where(ecco_ds.maskC.isel(k=0)>0, ecco_ds.SSH.
↪isel(time=0), np.nan)

new_grid_delta_lat = 0.05
new_grid_delta_lon = 0.05

new_grid_min_lat = 30+new_grid_delta_lat/2
new_grid_max_lat = 82-new_grid_delta_lat/2

new_grid_min_lon = -90+new_grid_delta_lon/2
new_grid_max_lon = 10-new_grid_delta_lon/2
```

(continues on next page)

(continued from previous page)

```
new_grid_lon, new_grid_lat, field_nearest_1deg =\
        ecco.resample_to_latlon(ecco_ds.XC, \
                                ecco_ds.YC, \
                                original_field_with_land_mask,\
                                new_grid_min_lat, new_grid_max_lat, new_grid_delta_lat,\
                                new_grid_min_lon, new_grid_max_lon, new_grid_delta_lon,\
                                fill_value = np.NaN, \
                                mapping_method = 'nearest_neighbor',
                                radius_of_influence = 120000)

# plot the whole field, this time land values are nans.
plt.figure(figsize=(6,6), dpi= 90)
plt.imshow(field_nearest_1deg,origin='lower',cmap='jet');
```



The new grid is much finer than the llc90 grid. Because we are using the nearest neighbor method, you can make out the approximate size and location of the llc90 model grid cells (think about it: nearest neighbor).

0.05 degree, bin average

With bin averaging many values from the source grid are 'binned' together and then averaged to determine the value in the new grid. The numer of grid cells from the source grid depends on the source grid resolution and the radius of influence. If we were to choose a radius of influence of 120000 m (a little longer than 1 degree longitude at the equator) the interpolated lat-lon map would be smoother than if we were to choose a smaller radius. To demonstrate:

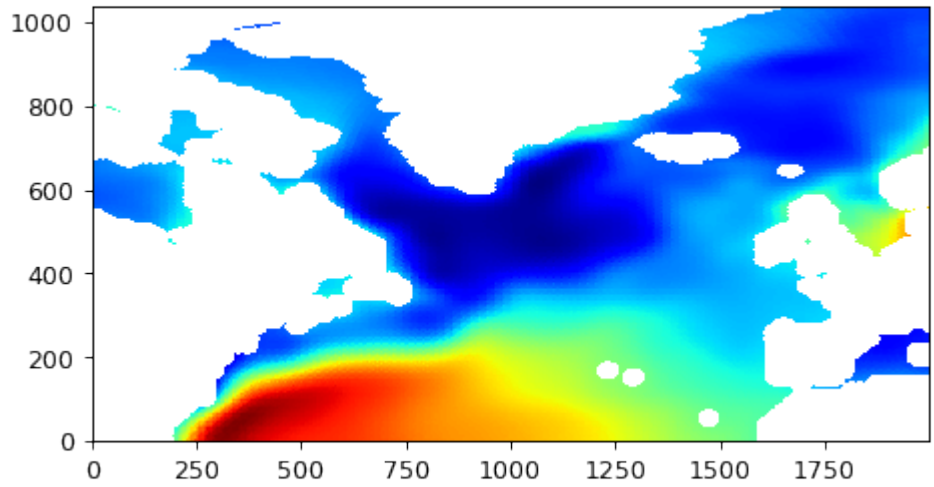
bin average with 120000 m radius

[18]:
```
original_field_with_land_mask= np.where(ecco_ds.maskC.isel(k=0)>0, ecco_ds.SSH.
↪isel(time=0), np.nan)

new_grid_lon, new_grid_lat, field_nearest_1deg =\
        ecco.resample_to_latlon(ecco_ds.XC, \
                                ecco_ds.YC, \
                                original_field_with_land_mask,\
                                new_grid_min_lat, new_grid_max_lat, new_grid_delta_lat,\
                                new_grid_min_lon, new_grid_max_lon, new_grid_delta_lon,\
```

(continues on next page)

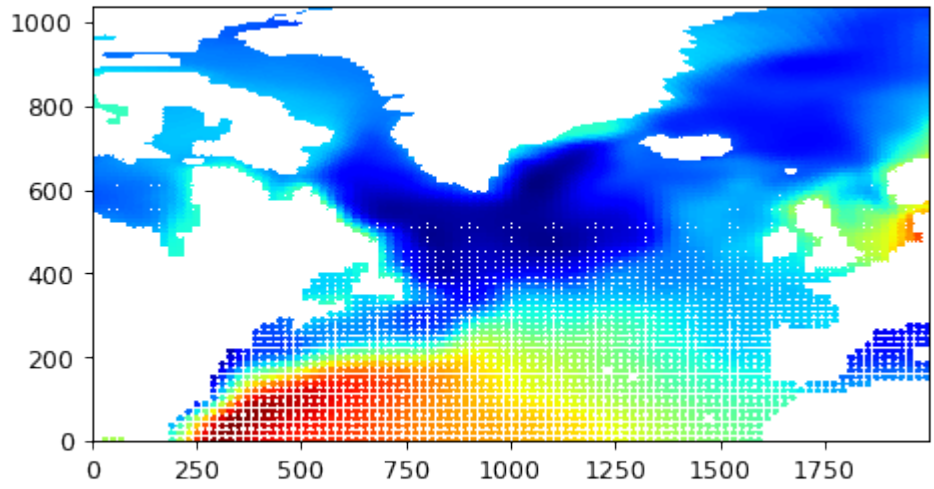
(continued from previous page)

```
                                fill_value = np.NaN, \
                                mapping_method = 'bin_average',
                                radius_of_influence = 120000)

# plot the whole field, this time land values are nans.
plt.figure(figsize=(6,6), dpi= 90)
plt.imshow(field_nearest_1deg,origin='lower',cmap='jet');
```



bin average with 40000m radius

[19]:
```
original_field_with_land_mask= np.where(ecco_ds.maskC.isel(k=0)>0, ecco_ds.SSH.
↪isel(time=0), np.nan)

new_grid_lon, new_grid_lat, field_nearest_1deg =\
        ecco.resample_to_latlon(ecco_ds.XC, \
                                ecco_ds.YC, \
                                original_field_with_land_mask,\
                                new_grid_min_lat, new_grid_max_lat, new_grid_delta_lat,\
                                new_grid_min_lon, new_grid_max_lon, new_grid_delta_lon,\
                                fill_value = np.NaN, \
                                mapping_method = 'bin_average',
                                radius_of_influence = 40000)

# plot the whole field, this time land values are nans.
plt.figure(figsize=(6,6), dpi= 90)
plt.imshow(field_nearest_1deg,origin='lower',cmap='jet');
```



You may wonder why there are missing (white) values in the resulting map. It's because there are some lat-lon grid cells whose center is more than 40km away from the center of any grid cell on the source grid. There are ways to get around this problem by specifying spatially-varying radii of influence, but that will have to wait for another tutorial.

If you want to explore on your own, explore some of the low-level routines of the pyresample package: https://pyresample.readthedocs.io/en/latest/

1.16.6 Interpolating ECCO vectors fields to lat-lon grids

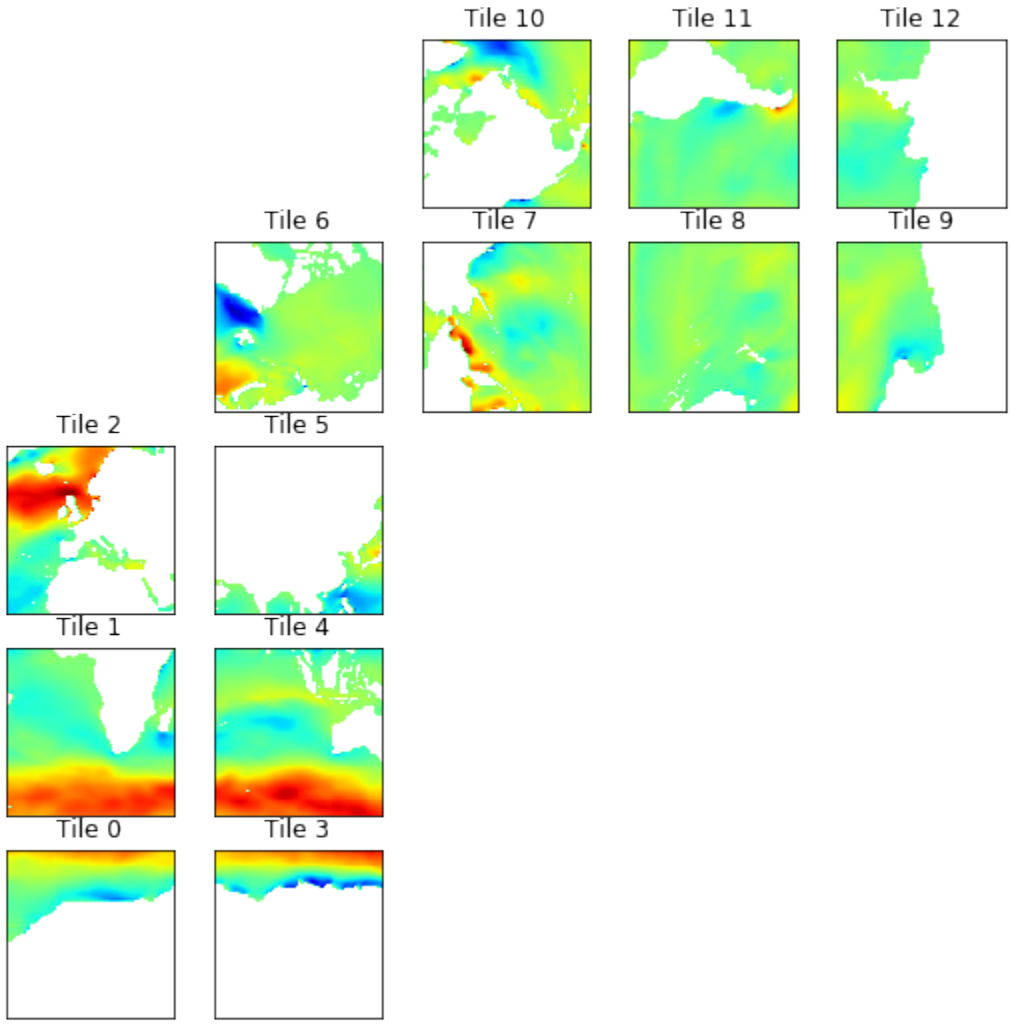
Vector fields require a few more steps to interpolate to the lat-lon grid. At least if what you want are the zonal and meridional vector components. Why? Because in the llc90 grid, vector fields like `UVEL` and `VVEL` are defined with positive in the +x and +y directions of the local coordinate system, respectively. A term like `oceTAUX` correponds to ocean wind stress in the +x direction and does not correspond to *zonal* wind stress. To calculate zonal wind stress we need to rotate the vector from the llc90 x-y grid system to the lat-lon grid.

To demonstrate why rotation is necessary, consider the model field `oceTAUX`

[20]:
```
## Load vector fields
day_mean_dir= ECCO_dir / 'nctiles_monthly/'

ecco_vars = ecco.recursive_load_ecco_var_from_years_nc(day_mean_dir, \
                                        vars_to_load=['oceTAUX', 'oceTAUY'], \
                                        years_to_load=2000,less_output=True)

ecco_ds = []

## Merge the ecco_grid with the ecco_vars to make the ecco_ds
ecco_ds = xr.merge((ecco_grid , ecco_vars)).load()

pprint(ecco_ds.data_vars)
```

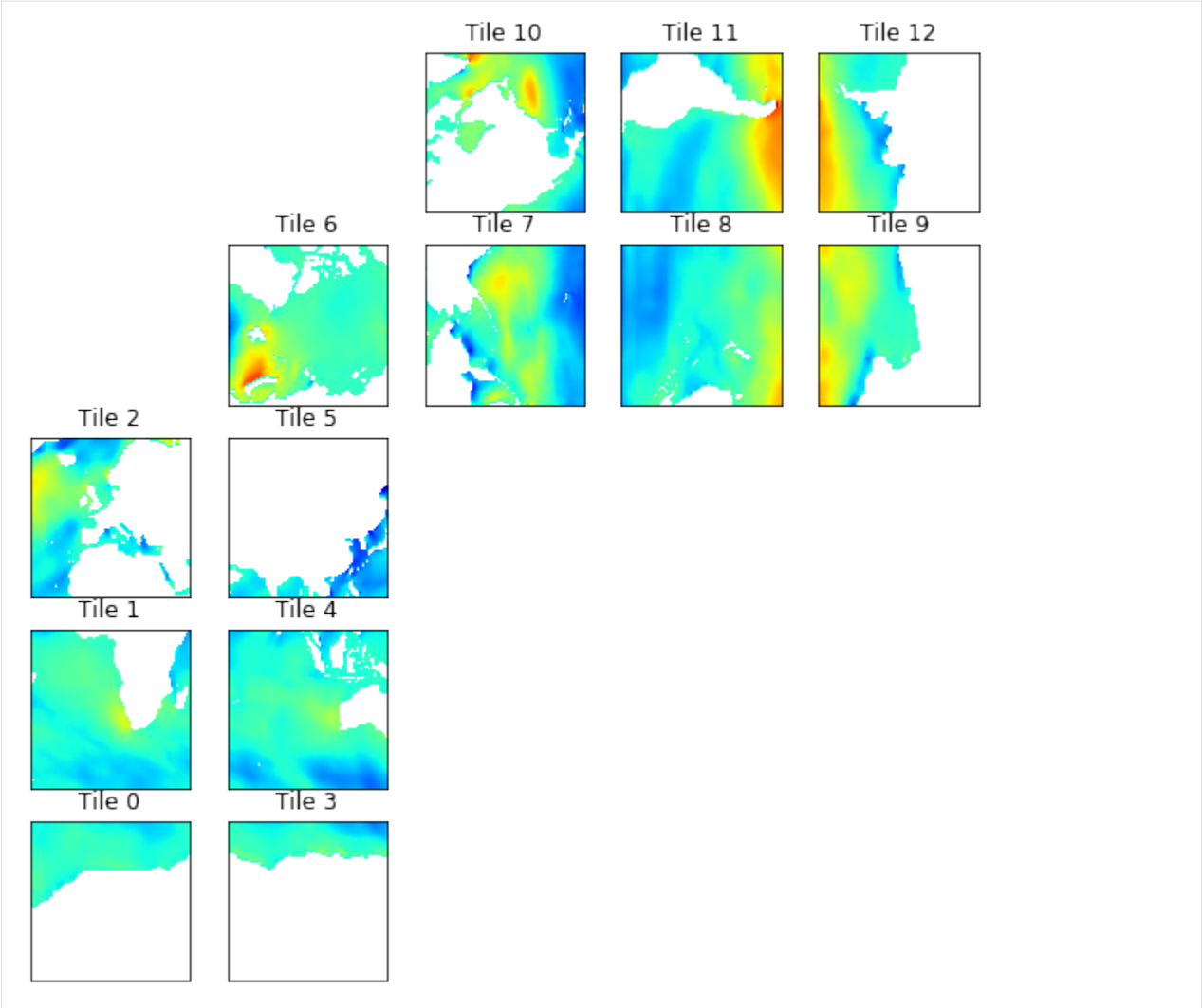
```
loading files of  oceTAUX
loading files of  oceTAUY
Data variables:
    oceTAUX  (time, tile, j, i_g) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    oceTAUY  (time, tile, j_g, i) float32 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
```

[21]:
```
tmp = ecco_ds.oceTAUX.isel(time=0)
tmp_masked = np.where(ecco_ds.maskC.isel(k=0)==1, tmp, np.nan)
ecco.plot_tiles(tmp_masked);
```



We can sees the expected positive zonal wind stress in tiles 0-5 because the x-y coordinates of tiles 0-5 happen to approximately line up with the meridians and parallels of the lat-lon grid. Importantly, for tiles 7-12 wind stress in the +x direction corresponds to mainly wind stress in the SOUTH direction. To see the zonal wind stress in tiles 7-12 one has to plot `oceTAUY` and recognize that for those tiles positive values correspond with wind stress in the tile's +y direction, which is approximately east-west. To wit,

[22]:
```
tmp = ecco_ds.oceTAUY.isel(time=0)
tmp_masked = np.where(ecco_ds.maskC.isel(k=0)==1, tmp, np.nan)
ecco.plot_tiles(tmp_masked);
```



Great, but if you want zonal and meridional wind stress, we need to determine the zonal and meridional components of these vectors.

Vector rotation

For vector rotation we leverage the very useful XGCM package (https://xgcm.readthedocs.io/en/latest/).

> **Note:** *The XGCM documentation contains an MITgcm ECCOv4 Example Page: https://xgcm.readthedocs.io/en/latest/example_eccov4.html. In that example the dimension 'tile' is called 'face' and the fields were loaded from the binary output of the MITgcm, not the netCDF files that we produce for the official ECCOv4r4 product. Differences are largely cosmetic.)*

We use XGCM to map the +x and +y vector components to the grid cell centers from the `u` and `v` grid points of the Arakawa C grid. The ecco-v4-py routine `get_llc_grid` creates the XGCM grid object using a DataSet object containing the following information about the model grid:

```
i,j,i_g,j_g,k,k_l,k_u,k_p1 .
```

Our `ecco_ds` DataSet does have these fields, they come from the `ECCO-GRID.nc` object:

[23]:
```
# dimensions of the ecco_ds DataSet
ecco_ds.dims
```

[23]:
```
Frozen(SortedKeysDict({'k_p1': 51, 'j_g': 90, 'i_g': 90, 'k': 50, 'j': 90, 'k_u': 50, 'i
↪': 90, 'k_l': 50, 'tile': 13, 'time': 12, 'nv': 2}))
```

[24]:
```
# Make the XGCM object
XGCM_grid = ecco.get_llc_grid(ecco_ds)

# look at the XGCM object.
XGCM_grid

# Depending on how much you want to geek out, you can learn about this fancy XGCM_grid␣
↪object here:
# https://xgcm.readthedocs.io/en/latest/grid_topology.html
```

[24]:
```
<xgcm.Grid>
T Axis (not periodic):
  * center   time
X Axis (not periodic):
  * center   i --> left
  * left     i_g --> center
Y Axis (not periodic):
  * center   j --> left
  * left     j_g --> center
Z Axis (not periodic):
  * center   k --> left
  * left     k_l --> center
  * outer    k_p1 --> center
  * right    k_u --> center
```

Once we have the XGCM_grid object, we can use built-in routines of XGCM to interpolate the x and y components of a vector field to the cell centers.

[25]:
```
import xgcm
xfld = ecco_ds.oceTAUX.isel(time=0)
yfld = ecco_ds.oceTAUY.isel(time=0)

velc = XGCM_grid.interp_2d_vector({'X': xfld, 'Y': yfld},boundary='fill')
```

velc is a dictionary of the x and y vector components taken to the model grid cell centers. At this point they are not yet rotated!

[26]:
```
velc.keys()
```

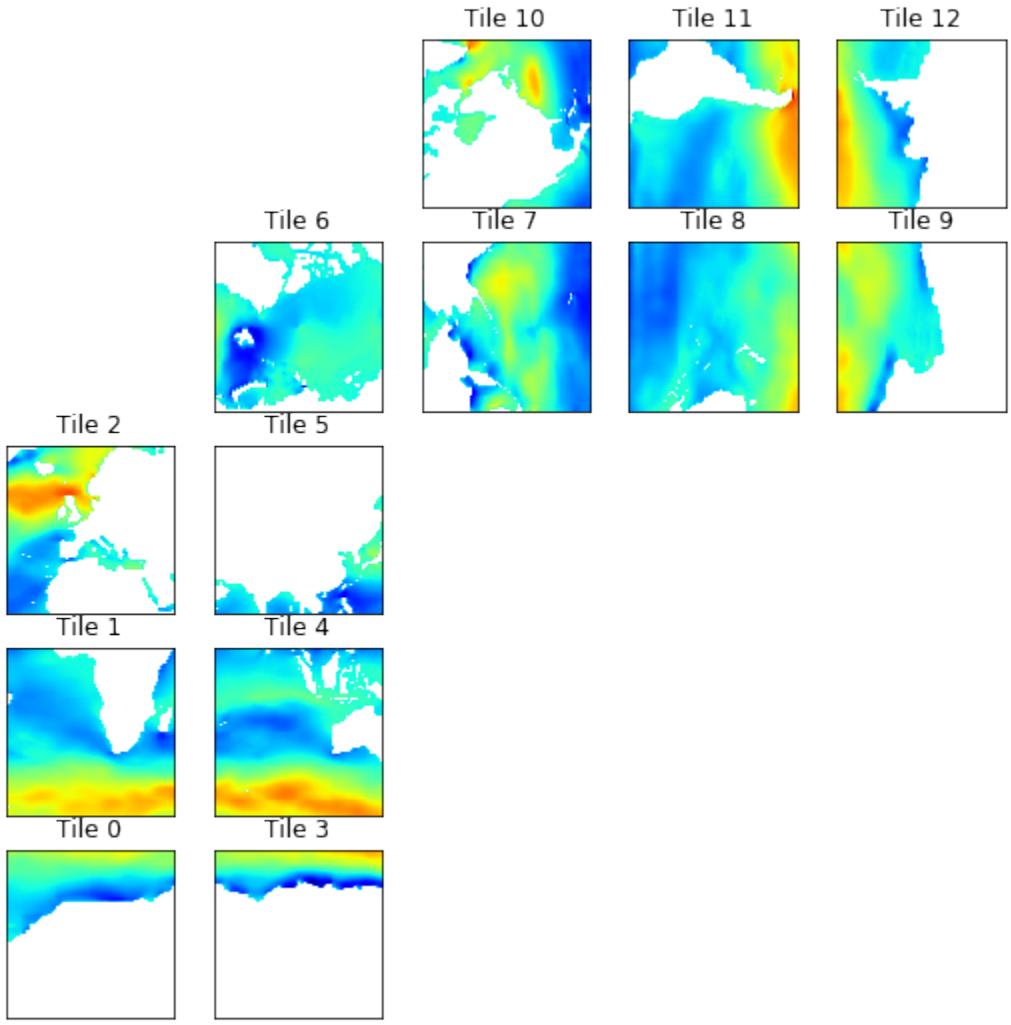
[26]:
```
dict_keys(['X', 'Y'])
```

The magic comes here, with the use of the grid cosine 'cs' and grid sine 'cs' values of the ECCO-GRID object:

[27]:
```
# Compute the zonal and meridional vector components of oceTAUX and oceTAUY
oceTAU_E  = velc['X']*ecco_ds['CS'] - velc['Y']*ecco_ds['SN']
oceTAU_N  = velc['X']*ecco_ds['SN'] + velc['Y']*ecco_ds['CS']
```

Now we have the zonal and meridional components of the vectors, albeit still on the llc90 grid.

[28]:
```
oceTAU_E_masked = np.where(ecco_ds.maskC.isel(k=0)>0, oceTAU_E, np.nan)
ecco.plot_tiles(oceTAU_E_masked);
```



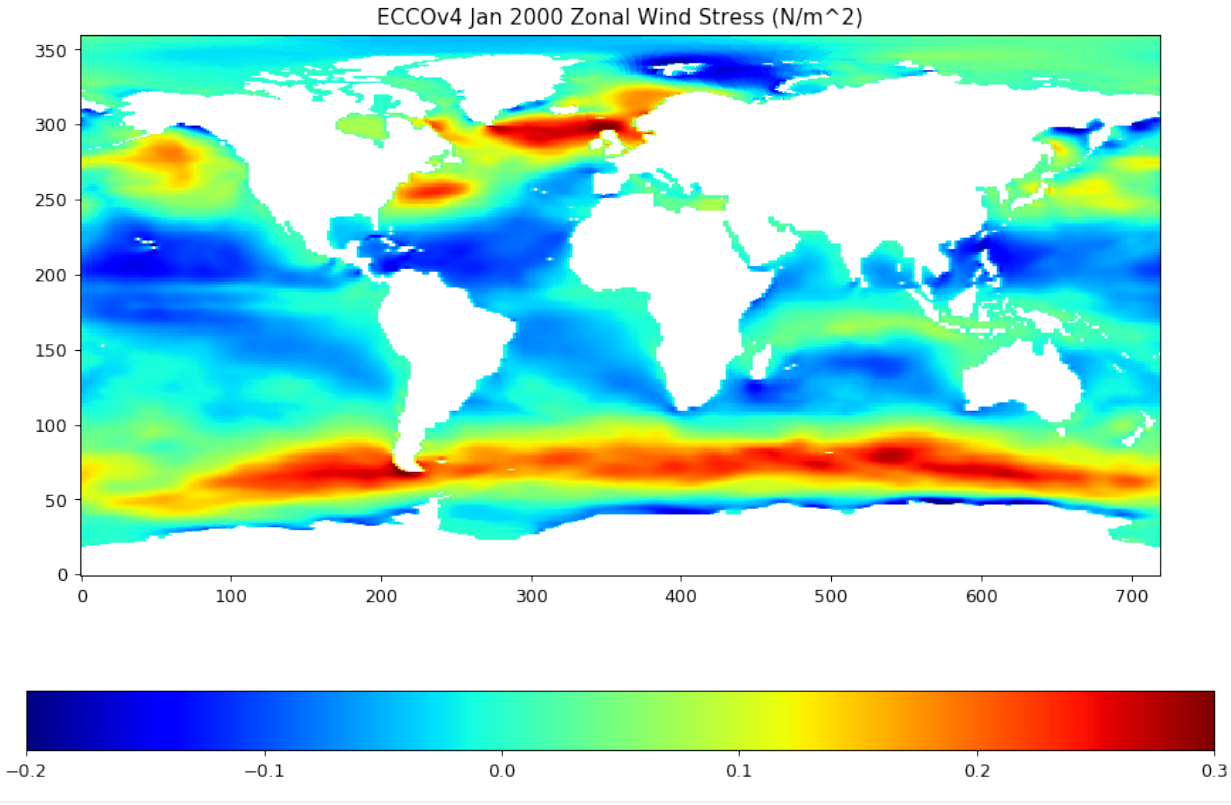
So let's resample `oceTAU_E` to a lat-lon grid (that's why you're here, right?) and plot

[29]:
```
new_grid_delta_lat = .5
new_grid_delta_lon = .5

new_grid_min_lat = -90+new_grid_delta_lat/2
new_grid_max_lat = 90-new_grid_delta_lat/2

new_grid_min_lon = -180+new_grid_delta_lon/2
new_grid_max_lon = 180-new_grid_delta_lon/2

new_grid_lon, new_grid_lat, oceTAU_E_masked_latlon =\
        ecco.resample_to_latlon(ecco_ds.XC, \
                                ecco_ds.YC, \
                                oceTAU_E_masked,\
                                new_grid_min_lat, new_grid_max_lat, new_grid_delta_lat,\
```

(continues on next page)

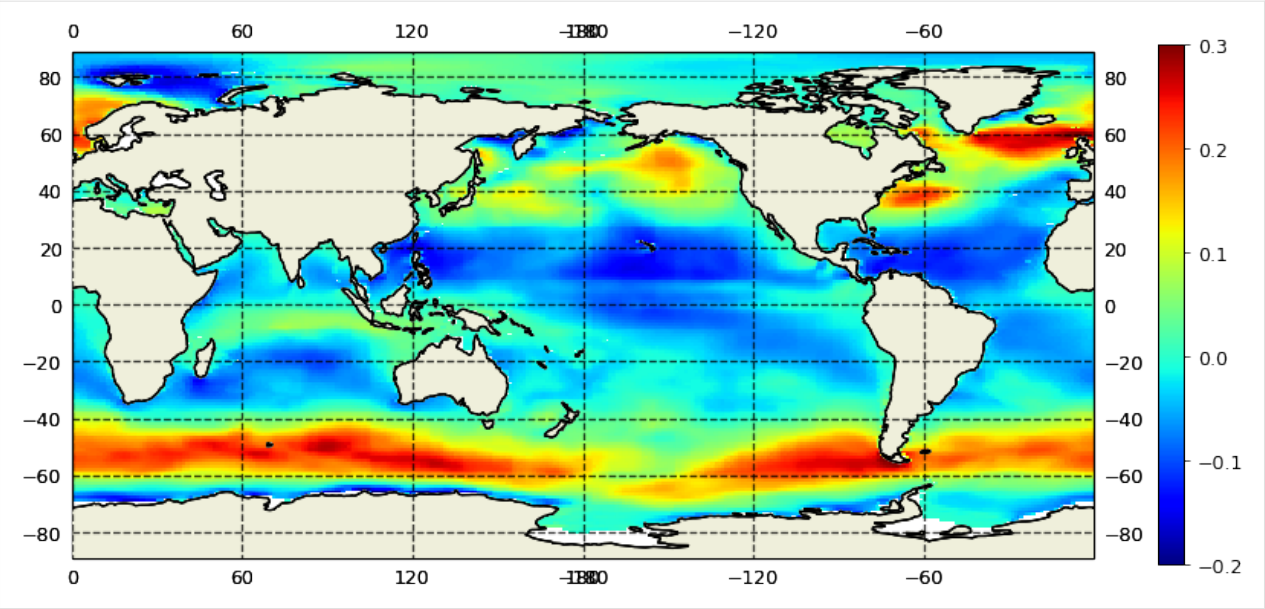
(continued from previous page)

```
                                new_grid_min_lon, new_grid_max_lon, new_grid_delta_lon,\
                                fill_value = np.NaN, \
                                mapping_method = 'nearest_neighbor',
                                radius_of_influence = 120000)

# plot the whole field, this time land values are nans.
plt.figure(figsize=(12,8), dpi= 90);
plt.imshow(oceTAU_E_masked_latlon,origin='lower',vmin=-.2,vmax=.3,cmap='jet');
plt.title('ECCOv4 Jan 2000 Zonal Wind Stress (N/m^2)');
plt.colorbar(orientation='horizontal');
```



Or with extra fanciness:

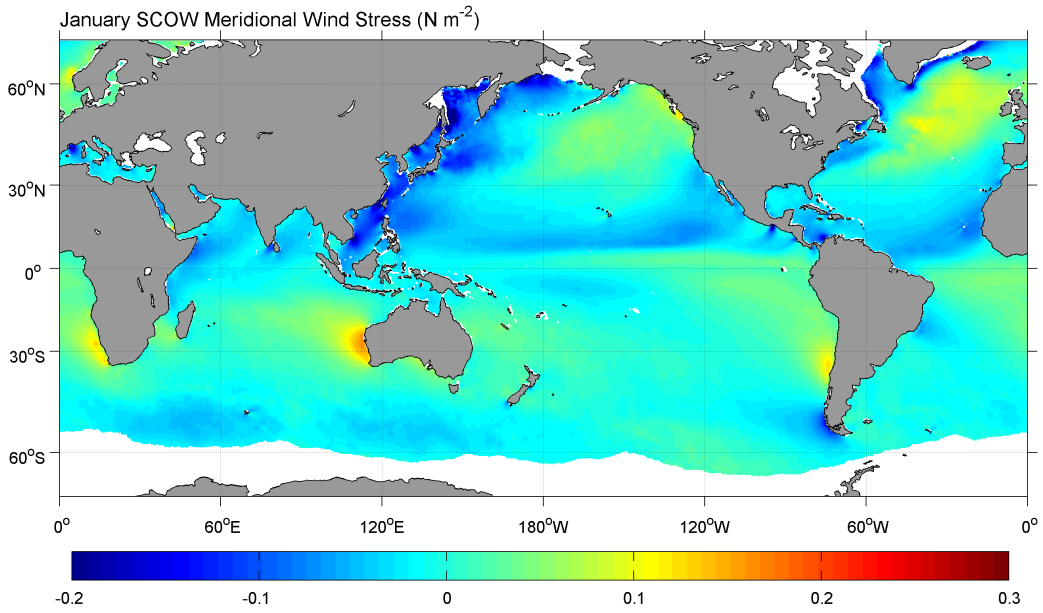
[30]:
```
plt.figure(figsize=(12,5), dpi= 90)

ecco.plot_proj_to_latlon_grid(ecco_ds.XC, ecco_ds.YC, \
                              oceTAU_E_masked, \
                              user_lon_0=180,\
                              projection_type='PlateCaree',\
                              plot_type = 'pcolormesh', \
                              dx=1,dy=1,show_colorbar=True,cmin=-.2, cmax=0.3);
```



Compare vs “The Scatterometer Climatology of Ocean Winds (SCOW)”,

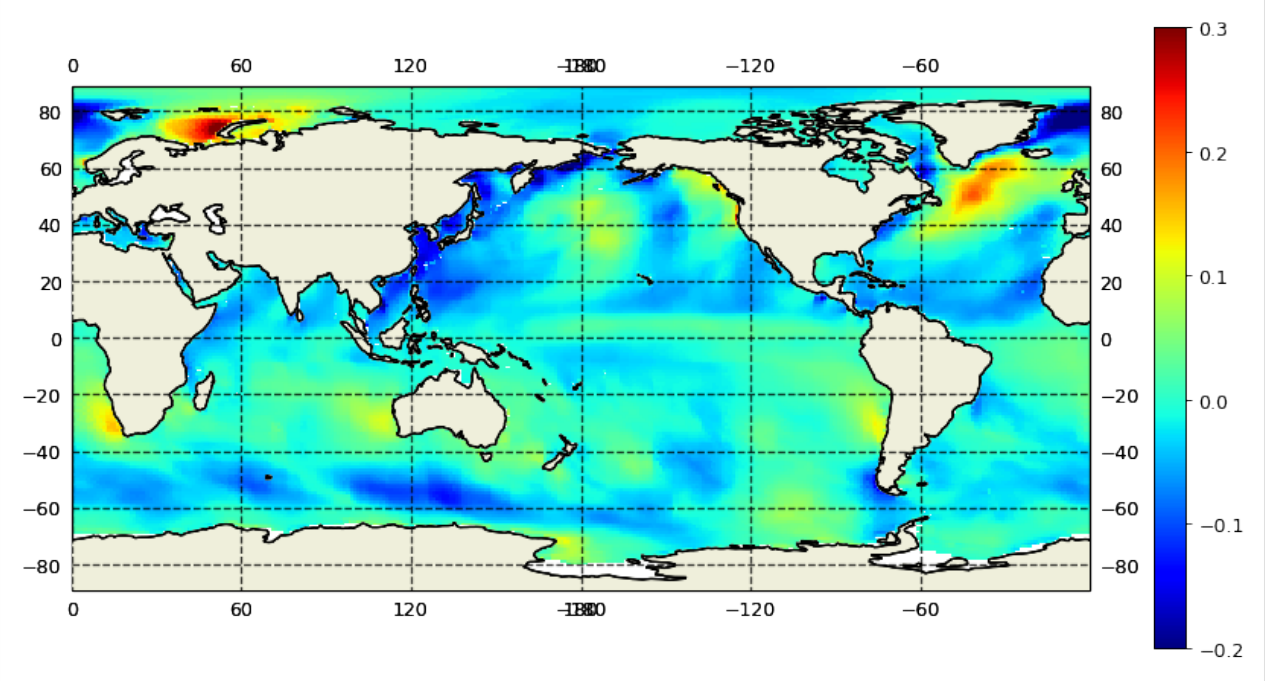
https://nanoos.ceoas.oregonstate.edu/scow/index.html

Risien, C.M., and D.B. Chelton, 2008: A Global Climatology of Surface Wind and Wind Stress Fields from Eight Years of QuikSCAT Scatterometer Data. J. Phys. Oceanogr., 38, 2379-2413.

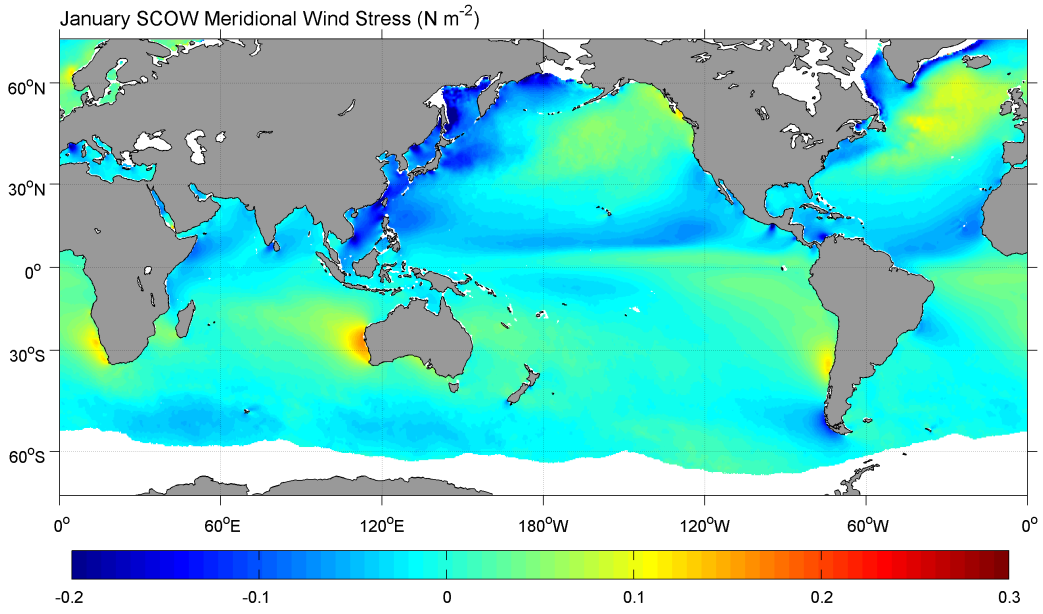


And of course we can't forget about our meridional wind stress:

[31]:
```
oceTAU_N_masked = np.where(ecco_ds.maskC.isel(k=0)>0, oceTAU_N, np.nan)

plt.figure(figsize=(12,6), dpi= 90)
ecco.plot_proj_to_latlon_grid(ecco_ds.XC, ecco_ds.YC, \
                              oceTAU_N_masked, \
                              user_lon_0=180,\
                              projection_type='PlateCaree',\
                              plot_type = 'pcolormesh', \
                              dx=1,dy=1,cmin=-.2, cmax=0.3,show_colorbar=True);
```



Which we also compare against SCOW:



UEVNfromUXVY

The ecco-v4-py library includes a routine, `UEVNfromUXVY` which does the interpolation to the grid cell centers and the rotation in one call:

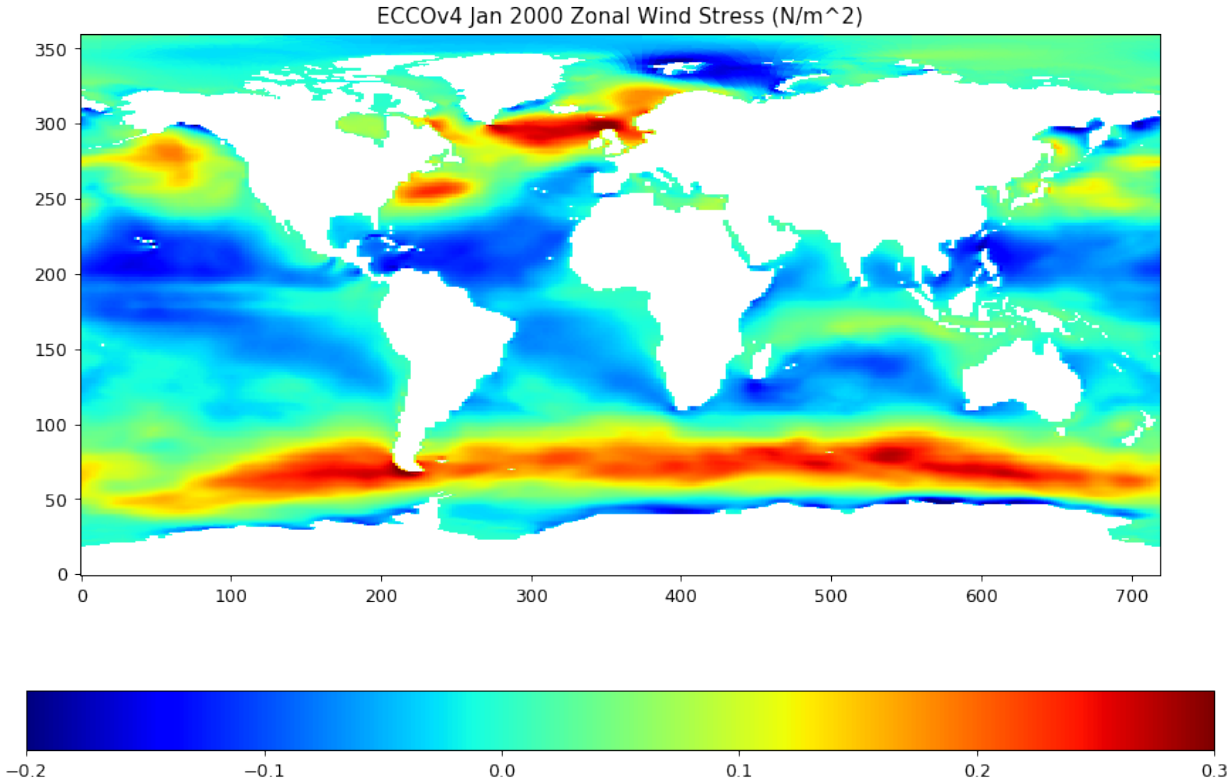
[32]:
```
xfld = ecco_ds.oceTAUX.isel(time=0)
yfld = ecco_ds.oceTAUY.isel(time=0)

# Compute the zonal and meridional vector components of oceTAUX and oceTAUY
oceTAU_E, oceTAU_N  = ecco.vector_calc.UEVNfromUXVY(xfld, yfld, ecco_ds)
```

Before interpolating to the lat-lon grid it is convenient to mask out the land values

[33]:
```
# mask the rotated vectors
oceTAU_E=oceTAU_E.where(ecco_ds.maskC.isel(k=0))
oceTAU_N=oceTAU_N.where(ecco_ds.maskC.isel(k=0))
```

Now plot to verify

[34]:
```
# interpolate to lat-lon
new_grid_lon, new_grid_lat, oceTAU_E_masked_latlon =\
        ecco.resample_to_latlon(ecco_ds.XC, \
                                ecco_ds.YC, \
                                oceTAU_E,\
                                new_grid_min_lat, new_grid_max_lat, new_grid_delta_lat,\
                                new_grid_min_lon, new_grid_max_lon, new_grid_delta_lon,\
                                fill_value = np.NaN, \
                                mapping_method = 'nearest_neighbor',
                                radius_of_influence = 120000)

# plot the whole field, this time land values are nans.
plt.figure(figsize=(12,8), dpi= 90);
plt.imshow(oceTAU_E_masked_latlon,origin='lower',vmin=-.2,vmax=.3,cmap='jet');
plt.title('ECCOv4 Jan 2000 Zonal Wind Stress (N/m^2)');
plt.colorbar(orientation='horizontal');
```



1.16.7 Saving interpolated fields to netCDF

For this demonstration we will rotate the 12 monthly-mean records of oceTAUX and oceTAUY to their zonal and meridional components, interpolate to a lat-lon grids, and then save the output as netCDF format.

[35]:
```
pprint(ecco_ds.oceTAUX.time)
```

```
<xarray.DataArray 'time' (time: 12)>
array(['2000-01-16T12:00:00.000000000', '2000-02-15T12:00:00.000000000',
       '2000-03-16T12:00:00.000000000', '2000-04-16T00:00:00.000000000',
       '2000-05-16T12:00:00.000000000', '2000-06-16T00:00:00.000000000',
       '2000-07-16T12:00:00.000000000', '2000-08-16T12:00:00.000000000',
       '2000-09-16T00:00:00.000000000', '2000-10-16T12:00:00.000000000',
       '2000-11-16T00:00:00.000000000', '2000-12-16T12:00:00.000000000'],
      dtype='datetime64[ns]')
Coordinates:
    timestep  (time) int64 70860 71556 72300 73020 ... 76692 77436 78156 78900
  * time      (time) datetime64[ns] 2000-01-16T12:00:00 ... 2000-12-16T12:00:00
Attributes:
    long_name:  center time of averaging period
    bounds:     time_bnds
    axis:       T
```

We will loop through each month, use `UEVNfromUXVY` to determine the zonal and meridional components of the ocean wind stress vectors, and then interpolate to a 0.5 degree lat-lon grid.

[36]:
```
oceTAUE = np.zeros((12, 360,720))
oceTAUN = np.zeros((12, 360,720))

new_grid_delta_lat = .5
new_grid_delta_lon = .5

new_grid_min_lat = -90+new_grid_delta_lat/2
new_grid_max_lat = 90-new_grid_delta_lat/2

new_grid_min_lon = -180+new_grid_delta_lon/2
new_grid_max_lon = 180-new_grid_delta_lon/2


for m in range(12):
    cur_oceTAUX = ecco_ds.oceTAUX.isel(time=m)
    cur_oceTAUY = ecco_ds.oceTAUY.isel(time=m)

    # Compute the zonal and meridional vector components of oceTAUX and oceTAUY
    tmp_e, tmp_n  = ecco.vector_calc.UEVNfromUXVY(cur_oceTAUX, cur_oceTAUY, ecco_ds)

    # apply landmask
    tmp_e_masked = np.where(ecco_ds.maskC.isel(k=0)>0, tmp_e, np.nan)
    tmp_n_masked = np.where(ecco_ds.maskC.isel(k=0)>0, tmp_n, np.nan)

    # zonal component
    new_grid_lon, new_grid_lat, tmp_e_masked_latlon =\
        ecco.resample_to_latlon(ecco_ds.XC, \
                                ecco_ds.YC, \
                                tmp_e_masked,\
                                new_grid_min_lat, new_grid_max_lat, new_grid_delta_lat,\
                                new_grid_min_lon, new_grid_max_lon, new_grid_delta_lon,\
                                fill_value = np.NaN, \
                                mapping_method = 'nearest_neighbor',
                                radius_of_influence = 120000)

    # meridional component
    new_grid_lon, new_grid_lat, tmp_n_masked_latlon =\
        ecco.resample_to_latlon(ecco_ds.XC, \
                                ecco_ds.YC, \
                                tmp_n_masked,\
                                new_grid_min_lat, new_grid_max_lat, new_grid_delta_lat,\
                                new_grid_min_lon, new_grid_max_lon, new_grid_delta_lon,\
                                fill_value = np.NaN, \
                                mapping_method = 'nearest_neighbor',
                                radius_of_influence = 120000)

    oceTAUE[m,:] = tmp_e_masked_latlon
    oceTAUN[m,:] = tmp_n_masked_latlon
```

[37]:
```
# make the new data array structures for the zonal and meridional wind stress fields
oceTAUE_DA = xr.DataArray(oceTAUE,  name = 'oceTAUE',
                      dims = ['time','latitude','longitude'],
```

(continues on next page)

(continued from previous page)

```
                        coords = {'latitude': new_grid_lat[:,0],
                                  'longitude': new_grid_lon[0,:],
                                  'time': ecco_ds.time})

# make the new data array structures for the zonal and meridional wind stress fields
oceTAUN_DA = xr.DataArray(oceTAUN,  name = 'oceTAUN',
                        dims = ['time','latitude','longitude'],
                        coords = {'latitude': new_grid_lat[:,0],
                                  'longitude': new_grid_lon[0,:],
                                  'time': ecco_ds.time})
```

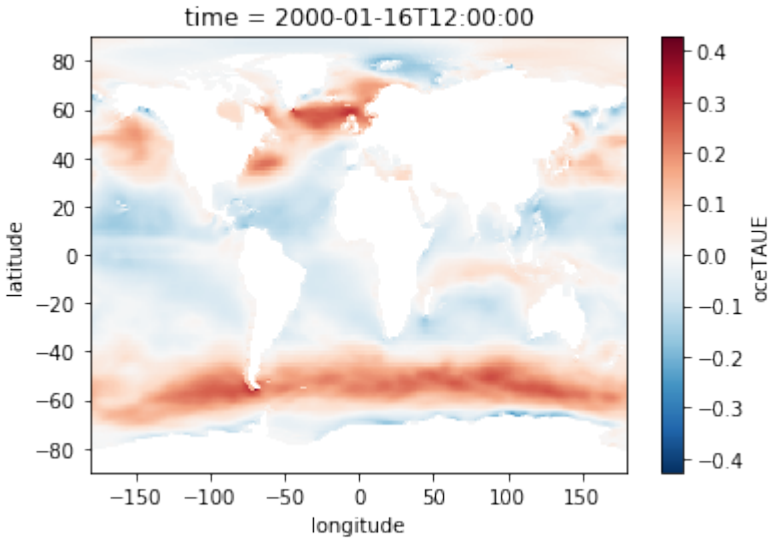
Plot the zonal and meridional wind stresses in these new DataArray objects:

[38]:
```
oceTAUE_DA.dims
```

[38]:
```
('time', 'latitude', 'longitude')
```

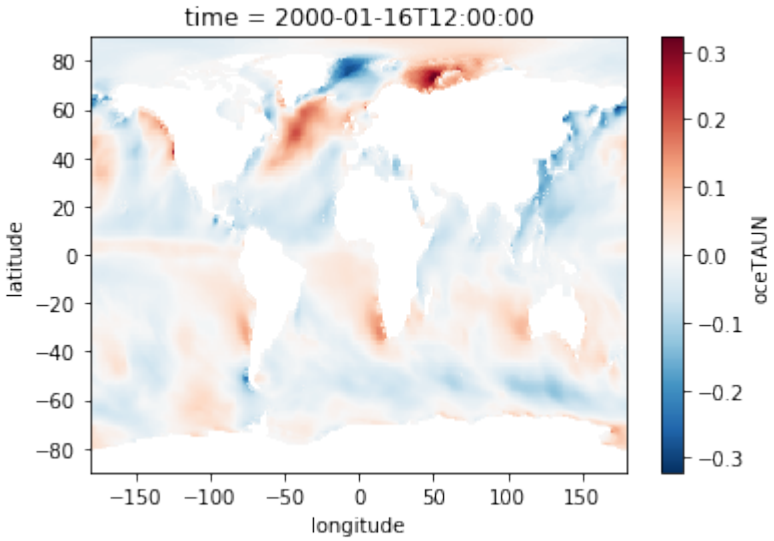
[39]:
```
oceTAUE_DA.isel(time=0).plot()
```

[39]:
```
<matplotlib.collections.QuadMesh at 0x1ff001ebee0>
```



[40]:
```
oceTAUN_DA.isel(time=0).plot()
```

[40]:
```
<matplotlib.collections.QuadMesh at 0x1ff00c6d730>
```



Saving is easy with xarray

[41]:
```
# meridional component
output_path = Path('C:/Users/Ian/Downloads/oceTAUN_latlon_2000.nc')
oceTAUN_DA.to_netcdf(output_path)

# zonal component
output_path = Path('C:/Users/Ian/Downloads/oceTAUE_latlon_2000.nc')
oceTAUE_DA.to_netcdf(output_path)
```

... done!

1.17 Example calculations with scalar quantities

1.17.1 Objectives

To demonstrate basic calculations using scalar fields (e.g., SSH, T, S) from the state estimate including: time series of mean quantities, spatial patterns of mean quantities, spatial patterns of linear trends, and spatial patterns of linear trends over different time periods.

1.17.2 Introduction

We will demonstrate global calculations with SSH (global mean sea level time series, mean dynamic topography, global mean sea level trend) and a regional calculation with THETA (The Nino 3.4 index).

1.17.3 Global calculations with SSH

First, load daily and monthly-mean SSH and THETA fields and the model grid parameters.

[1]:
```
import numpy as np
import sys
import xarray as xr
from copy import deepcopy
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
## Import the ecco_v4_py library into Python
## =========================================

## -- If ecco_v4_py is not installed in your local Python library,
##    tell Python where to find it.  For example, if your ecco_v4_py
##    files are in /Users/ifenty/ECCOv4-py/ecco_v4_py, then use:

sys.path.append('/home/ifenty/ECCOv4-py')
import ecco_v4_py as ecco
```

[3]:
```
## Set top-level file directory for the ECCO NetCDF files
## ======================================================
# base_dir = '/home/username/'
base_dir = '/home/ifenty/ECCOv4-release'

## define a high-level directory for ECCO fields
ECCO_dir = base_dir + '/Release3_alt'
```

Now load daily and monthly mean versions of SSH and THETA

[4]:
```
## Load the model grid
grid_dir= ECCO_dir + '/nctiles_grid/'

ecco_grid = ecco.load_ecco_grid_nc(grid_dir, 'ECCOv4r3_grid.nc', k_subset=[0])
```

[5]:
```
## Load 2D DAILY data, SSH, SST, and SSS
data_dir= ECCO_dir + '/nctiles_daily'

ecco_daily_vars = ecco.recursive_load_ecco_var_from_years_nc(data_dir, \
                                          vars_to_load=['SSH','THETA'],\
                                          years_to_load=range(1993,2018)).load()

## Merge the ecco_grid with the ecco_vars to make the ecco_ds
ecco_daily_ds = xr.merge((ecco_grid , ecco_daily_vars))
```

```
loading files of  SSH
loading files of  THETA
```

[7]:
```
## Load 2D MONTHLY data, SSH, SST, and SSS
data_dir= ECCO_dir + '/nctiles_monthly'
```

(continues on next page)

(continued from previous page)

```
ecco_monthly_vars = ecco.recursive_load_ecco_var_from_years_nc(data_dir, \
                                            vars_to_load=['SSH','THETA'],\
                                            years_to_load=range(1993,2018), k_subset=[0]).
↪load()

## Merge the ecco_grid with the ecco_vars to make the ecco_ds
ecco_monthly_ds = xr.merge((ecco_grid , ecco_monthly_vars))
```

```
loading files of  THETA
loading files of  SSH
```

[8]:
```
print(ecco_daily_ds.time[0].values)
print(ecco_daily_ds.time[-1].values)

print(ecco_monthly_ds.time[0].values)
print(ecco_monthly_ds.time[-1].values)
```

```
1993-01-01T12:00:00.000000000
2015-12-31T12:00:00.000000000
1993-01-16T12:00:00.000000000
2015-12-16T12:00:00.000000000
```

1.17.4 Sea surface height

Global mean sea level

Global mean sea surface height at time t is defined as follows:

$$SSH_{\text{global mean}}(t) = \frac{\sum_i SSH(i,t)\ A(i)}{A_{\text{global ocean}}}$$

$$A_{\text{global ocean}} = \sum_i A(i)$$

Where $SSH(i,t)$ is dynamic height at model grid cell i and time t, $A(i)$ is the area (m^2) of model grid cell i

There are several ways of doing the above calculations. Since this is the first tutorial with actual calcuations, we'll present a few different approaches for getting to the same answer.

Part 1: $A_{\text{global ocean}}$

Let's start on the simplest quantity, the global ocean surface area $A_{\text{global ocean}}$. Our calculation uses *SSH* which is a 'c' point variable. The surface area of tracer grid cells is provided by the model grid parameter *rA*. *rA* is a two-dimensional field that is defined over all model grid points, including land.

To calculate the total ocean surface area we need to ignore the area contributions from land.

We will first construct a 3D mask that is True for model grid cells that are wet and False for model grid cells that are dry cells.

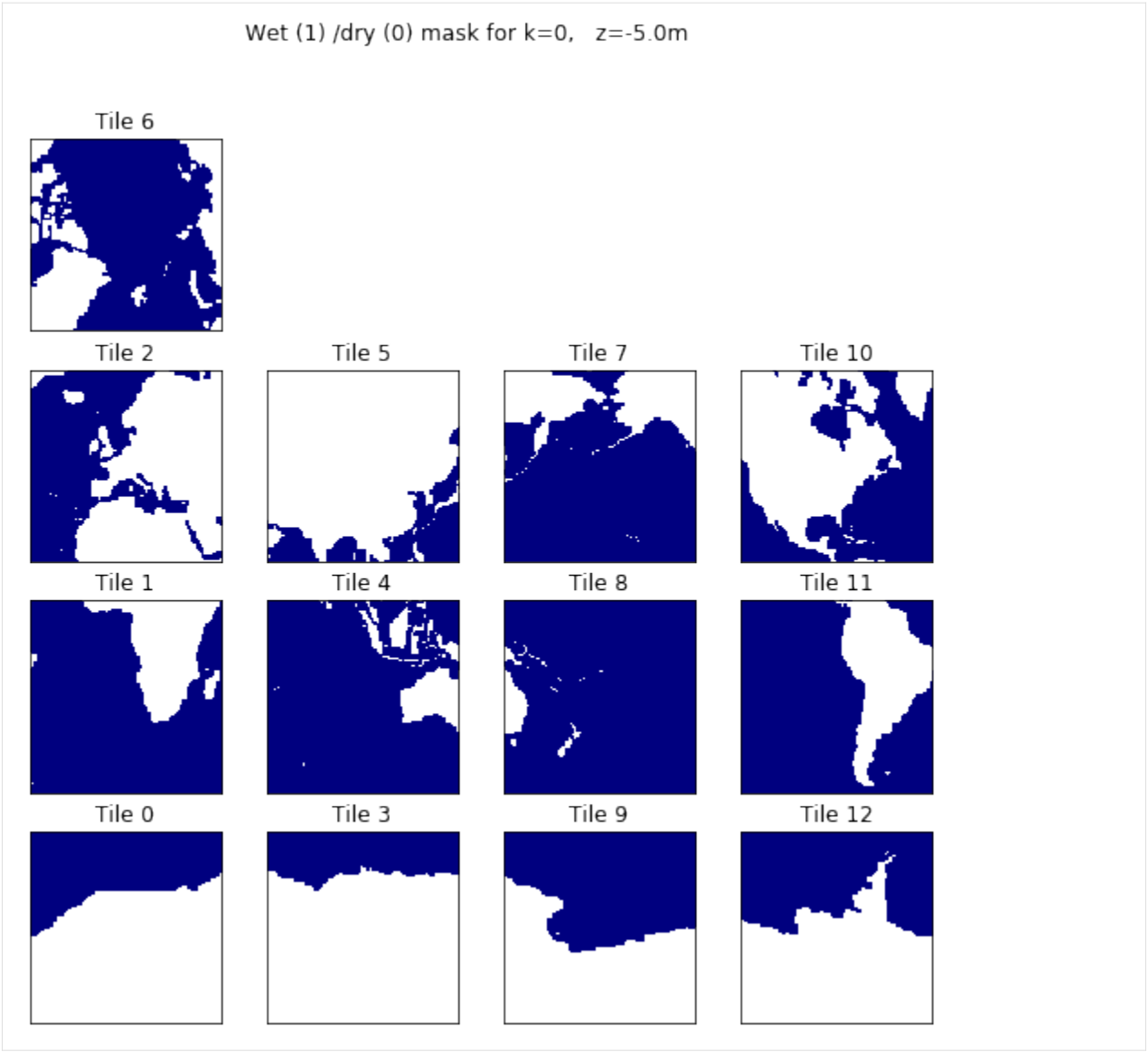
[9]:
```
# ocean_mask is ceiling of hFacC which is 0 for 100% dry cells and
# 0 > hFacC >= 1 for grid cells that are at least partially wet

# hFacC is the fraction of the thickness (h) of the grid cell which
# is wet.  we'll consider all hFacC > 0 as being a wet grid cell
# and so we use the 'ceiling' function to make all values > 0 equal to 1.

ocean_mask = np.ceil(ecco_monthly_ds.hFacC)
ocean_mask = ocean_mask.where(ocean_mask==1, np.nan)
```

[10]:
```
# the resulting ocean_mask variable is a 2D DataArray because we only loaded 1 vertical␣
↪level of the model grid
print(type(ocean_mask))
print((ocean_mask.dims))
```

```
<class 'xarray.core.dataarray.DataArray'>
('k', 'tile', 'j', 'i')
```

[11]:
```
plt.figure(figsize=(12,5), dpi= 90)

ecco.plot_tiles(ocean_mask.isel(k=0),layout='latlon', rotate_to_latlon=True)

# select out the model depth at k=1, round the number and convert to string.
z = str((np.round(ecco_monthly_ds.Z.values[0])))
plt.suptitle('Wet (1) /dry (0) mask for k=' + str(0) + ',   z=' + z + 'm');
```

```
<Figure size 1080x450 with 0 Axes>
```



To calculate $A_{\text{global ocean}}$ we must apply the surface wet/dry mask to rA.

[12]:
```
# Method 1: the array index method, []
#           select land_c at k index 0
total_ocean_area = np.sum(ecco_monthly_ds.rA*ocean_mask[0,:])

# these three methods give the same numerical result.  Here are
# three alternative ways of printing the result
print ('total ocean surface area ( m^2) %d  ' % total_ocean_area.values)
print ('total ocean surface area (km^2) %d ' % (total_ocean_area.values/1.0e6))

# or in scientific notation with 2 decimal points
print ('total ocean surface area (km^2) %.2E' % (total_ocean_area.values/1.0e6))
```

```
total ocean surface area ( m^2) 358013844062208
total ocean surface area (km^2) 358013844
total ocean surface area (km^2) 3.58E+08
```

This compares favorable with *Global surface area of Earth's Oceans : approx 3.60 x :math:`10^8` :math:`km^2` * from https://hypertextbook.com/facts/1997/EricCheng.shtml

Multiplication of DataArrays

You probably noticed that the multiplication of grid cell area with the land mask was done element by element. One useful feature of `DataArrays` is that their dimensions are automatically lined up when doing binary operations. Also, because *rA* and *ocean_mask* are both `DataArrays`, their inner product and their sums are also `DataArrays`.

> Note:: *ocean_mask* has a depth (**k**) dimension while *rA* does not (horizontal model grid cell area does not change as a function of depth in ECCOv4). As a result, when *rA* is multiplied with *ocean_mask* `xarray` **broadcasts** *rA* to all **k** levels. The resulting matrix inherits the **k** dimension from *ocean_mask*.

Another way of summing over `numpy` arrays

As *rA* and ocean both store `numpy` arrays, you can also calculate the sum of their product by invoking the `.sum()` command inherited in all `numpy arrays`:

[13]:
```
total_ocean_area = (ecco_monthly_ds.rA*ocean_mask).isel(k=0).sum()
print ('total ocean surface area (km^2) ' + '%.2E' % (total_ocean_area.values/1e6))
```

```
total ocean surface area (km^2) 3.58E+08
```

Part2 : $SSH_{\text{global mean}}(t)$

The global mean *SSH* at each t is given by,

$$SSH_{\text{global mean}}(t) = \frac{\sum_i SSH(i,t)\ A(i)}{A_{\text{global ocean}}}$$

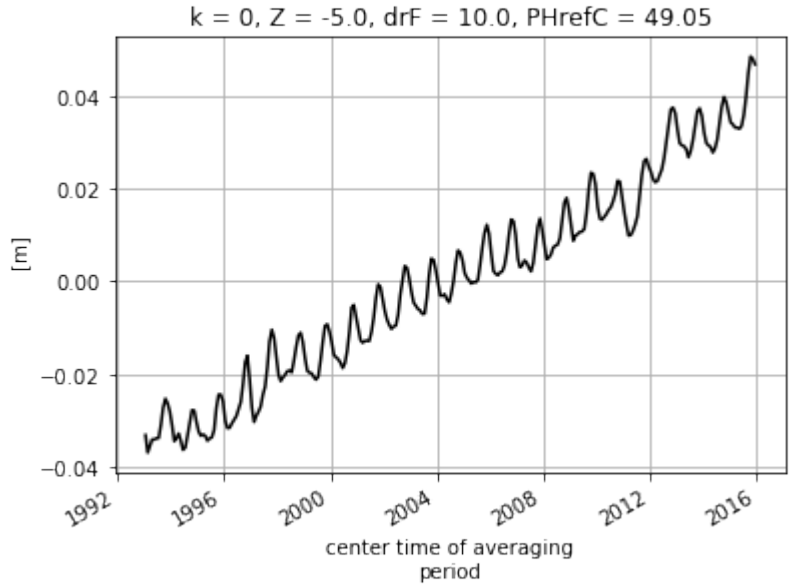
One way of calculating this is to take advantage of `DataArray` coordinate labels and use its *.sum()* functionality to explicitly specify which dimensions to sum over:

[14]:
```
# note no need to multiple RAC by land_c because SSH is nan over land
SSH_global_mean_mon = (ecco_monthly_ds.SSH*ecco_monthly_ds.rA).sum(dim=['i','j','tile'])/
↪total_ocean_area
```

[15]:
```
# remove time mean from time series
SSH_global_mean_mon = SSH_global_mean_mon-SSH_global_mean_mon.mean(dim='time')
```

[16]:
```
# add helpful unit label
SSH_global_mean_mon.attrs['units']='m'
```

[17]:
```
# and plot for fun
SSH_global_mean_mon.plot(color='k');plt.grid()
```



Alternatively we can do the summation over the three non-time dimensions. The time dimension of SSH is along the first dimension (axis) of the array, axis 0.

[18]:
```
# note no need to multiple RAC by land_c because SSH is nan over land
SSH_global_mean = np.sum(ecco_monthly_ds.SSH*ecco_monthly_ds.rA,axis=(1,2,3))/total_
↪ocean_area
SSH_global_mean = SSH_global_mean.compute()
```

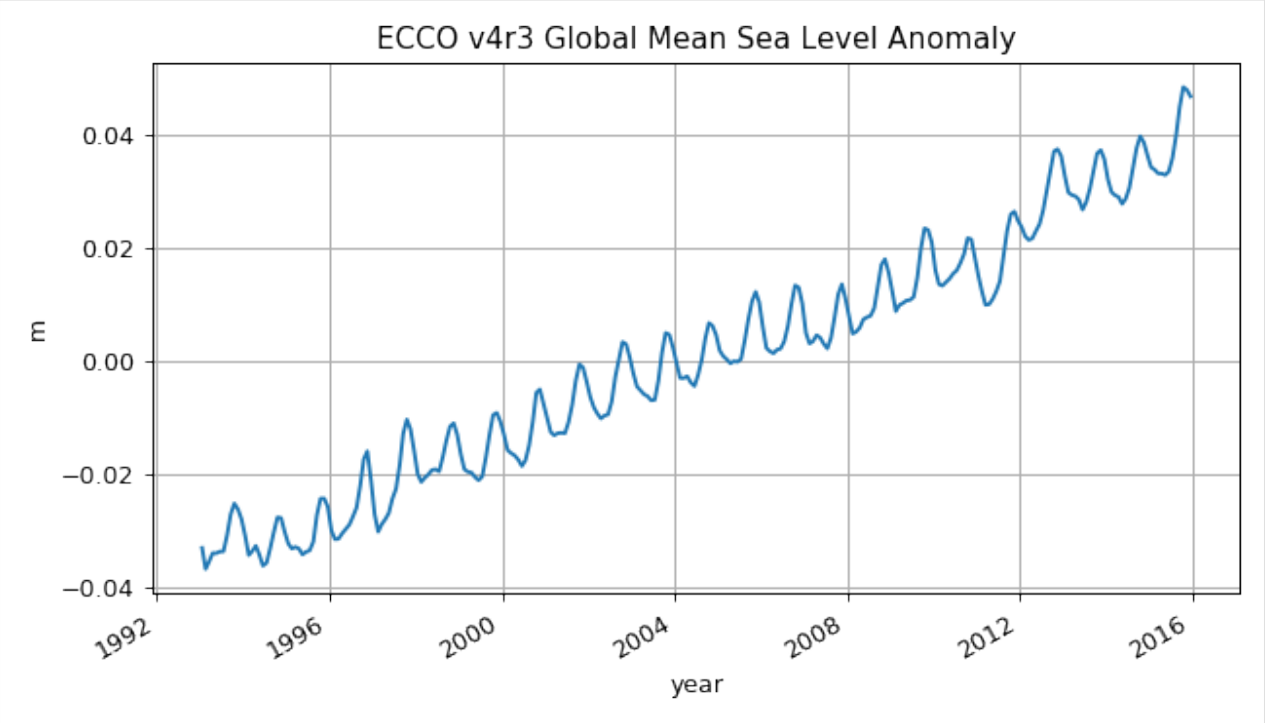
Even though *SSH* has 3 dimensions (time, tile, j, i) and *rA* and *ocean_mask.isel(k=0)* have 2 (j,i), we can mulitply them. With `xarray` the element-by-element multiplication occurs over their common dimension.

The resulting $SSH_{global-mean}$ `DataArray` has a single dimension, time.

Part 3 : Plotting the global mean sea level time series:

Before we plot the global mean sea level curve let's remove its time-mean to make it global mean sea level anomaly (the absolute value has no meaning here anyway).

[19]:
```
plt.figure(figsize=(8,4), dpi= 90)

# Method 1: .mean() method of `DataArrays`
SSH_global_mean_anomaly = SSH_global_mean - SSH_global_mean.mean()

# Method 2: numpy's `mean` method
SSH_global_mean_anomaly = SSH_global_mean - np.mean(SSH_global_mean)

SSH_global_mean_anomaly.plot()
plt.grid()
plt.title('ECCO v4r3 Global Mean Sea Level Anomaly');
plt.ylabel('m');
plt.xlabel('year');
```



Mean Dynamic Topography

Mean dynamic topography is calculated as follows,

$MDT(i) = $\sum_t SSH(i,t) - SSH_{\text{global mean}}(t)_{\overline{nt\$}}$

Where nt is the number of time records.

For *MDT* we presere the spatial dimensions. Summation and averaging are over the time dimensions (axis 0).

[20]:
```
## Two equivalent methods

# Method 1, specify the axis over which to average
MDT = np.mean(ecco_monthly_ds.SSH - SSH_global_mean,axis=0)

# Method 2, specify the coordinate label over which to average
MDT_B = (ecco_monthly_ds.SSH - SSH_global_mean).mean(dim=['time'])

# which can be verified using the '.equals()' method to compare Datasets and DataArrays
print(MDT.equals(MDT_B))
```

```
True
```

As expected, MDT has preserved its spatial dimensions:

[21]:
```
MDT.dims
```

[21]:
```
('tile', 'j', 'i')
```

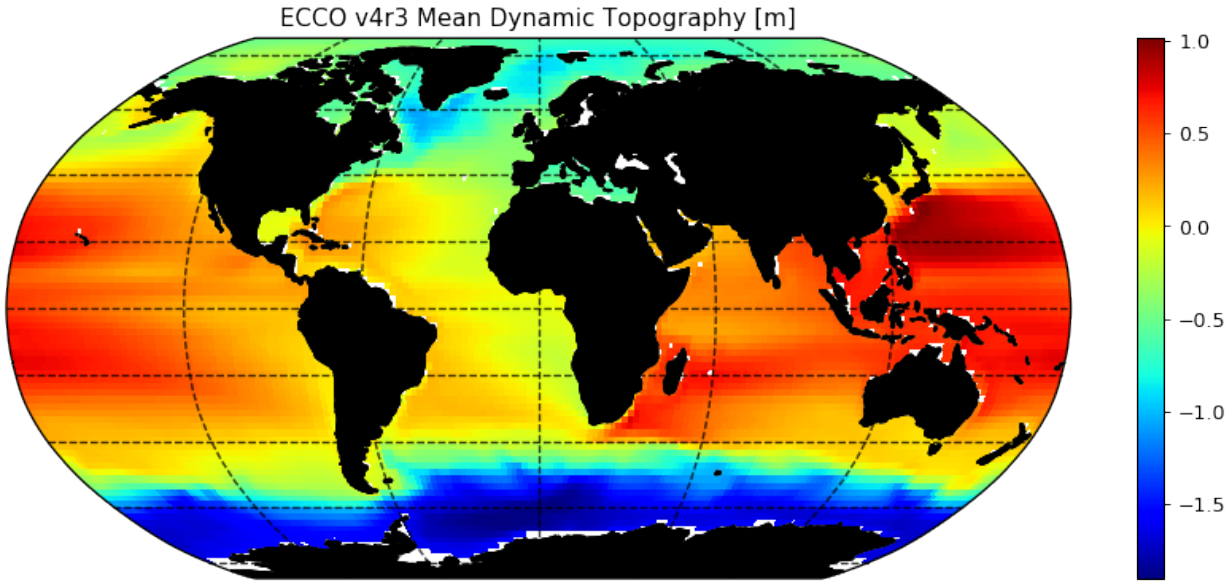
Before plotting the MDT field remove its spatial mean since its spatial mean conveys no dynamically useful information.

[22]:
```
MDT_no_spatial_mean = MDT - MDT*ecco_monthly_ds.rA/total_ocean_area
```

[23]:
```
MDT_no_spatial_mean.shape
```

[23]:
```
(13, 90, 90)
```

[24]:
```
plt.figure(figsize=(12,5), dpi= 90)

# mask land points to Nan
MDT_no_spatial_mean = MDT_no_spatial_mean.where(ocean_mask[0,:] !=0)

ecco.plot_proj_to_latlon_grid(ecco_monthly_ds.XC, \
                              ecco_monthly_ds.YC, \
                              MDT_no_spatial_mean*ocean_mask, \
                              user_lon_0=0,\
                              plot_type = 'pcolormesh', \
                              show_colorbar=True,\
                              dx=2,dy=2);

plt.title('ECCO v4r3 Mean Dynamic Topography [m]');
```



Spatial variations of sea level linear trends

To calculate the linear trend for the each model point we will use on the `polyfit` function of `numpy`. First, define a time variable in years for SSH.

[25]:
```
days_since_first_record = ((ecco_monthly_ds.time - ecco_monthly_ds.time[0])/(86400e9)).
↪astype(int).values
len(days_since_first_record)
```

[25]:
```
276
```

Next, reshape the four dimensional SSH field into two dimensions, time and space (t, i)

[26]:
```
ssh_flat = np.reshape(ecco_monthly_ds.SSH.values, (len(ecco_monthly_ds.SSH.time),␣
↪13*90*90))
ssh_flat.shape
```

[26]:
```
(276, 105300)
```

Now set all SSH values that are 'nan' to zero because the polynominal fitting routine can't handle nans,

[27]:
```
ssh_flat[np.nonzero(np.isnan(ssh_flat))] = 0
ssh_flat.shape
```

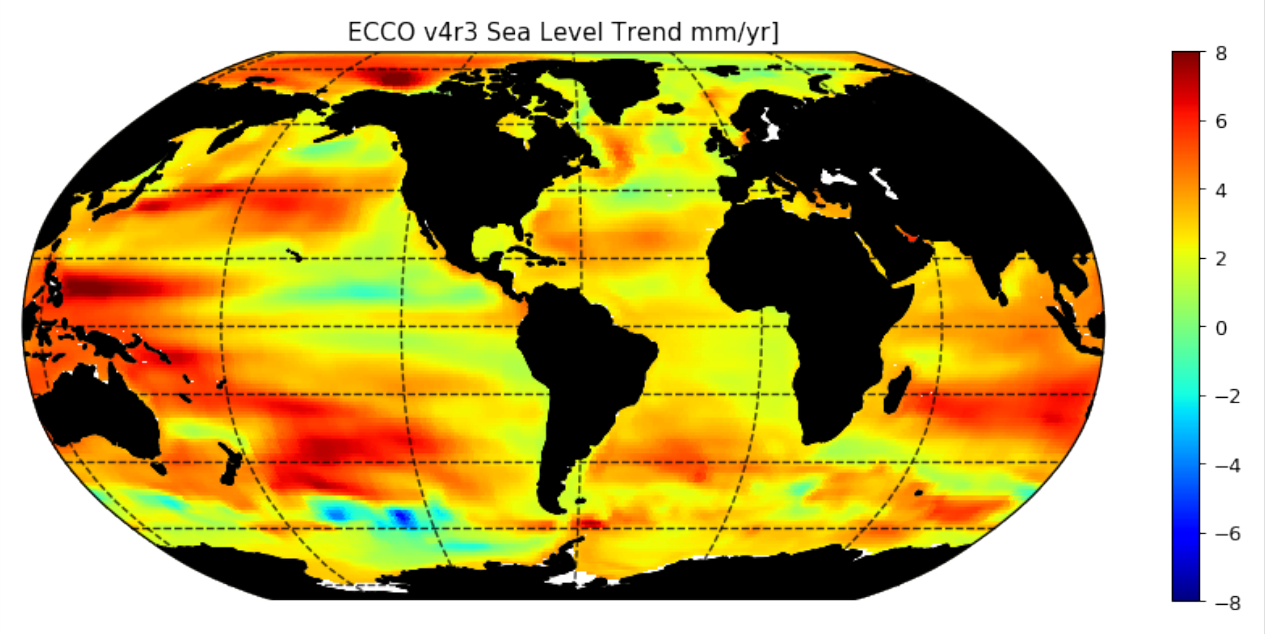
[27]:
```
(276, 105300)
```

Do the polynomial fitting, https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.polyfit.html

[28]:
```
# slope is in m / day
ssh_slope, ssh_intercept = np.polyfit(days_since_first_record, ssh_flat, 1)

print(ssh_slope.shape)

# and reshape the slope result back to 13x90x90
ssh_slope = np.reshape(ssh_slope, (13, 90,90))

# mask
ssh_slope_masked = np.where(ocean_mask[0,:] > 0, ssh_slope, np.nan)

# convert from m / day to mm/year
ssh_slope_mm_year = ssh_slope_masked*365*1e3
```

```
(105300,)
```

[33]:
```
plt.figure(figsize=(12,5), dpi= 90)

ecco.plot_proj_to_latlon_grid(ecco_monthly_ds.XC, \
                              ecco_monthly_ds.YC, \
                              ssh_slope_mm_year, \
                              user_lon_0=-66,\
                              plot_type = 'pcolormesh', \
                              show_colorbar=True,\
                              dx=1, dy=1, cmin=-8, cmax=8)

plt.title('ECCO v4r3 Sea Level Trend mm/yr]');
```



And the mean rate of global sea level change in mm/year over the 1993-2018 period is:

[34]:
```
((ssh_slope_mm_year*ecco_monthly_ds.rA)/(ecco_monthly_ds.rA*ocean_mask).sum()).sum()
```

[34]:
```
<xarray.DataArray ()>
array(3.18565693)
```

1.17.5 Constructing Monthly means from Daily means

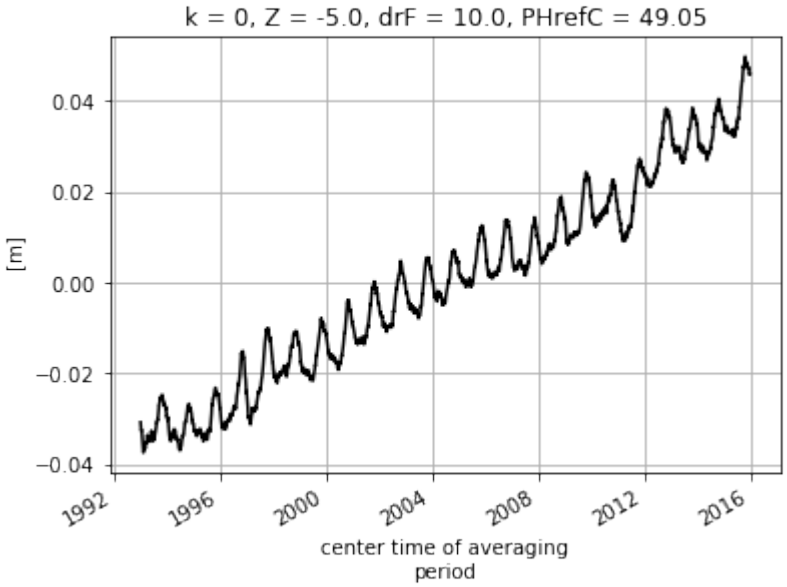
We can also construct our own monthly means from the daily means using this command: (See http://xarray.pydata.org/en/stable/generated/xarray.Dataset.resample.html for more information)

[35]:
```
# note no need to multiple RAC by land_c because SSH is nan over land
SSH_global_mean_day = (ecco_daily_ds.SSH*ecco_daily_ds.rA).sum(dim=['i','j','tile'])/
↪total_ocean_area
```

[36]:
```
# remove time mean from time series
SSH_global_mean_day = SSH_global_mean_day-SSH_global_mean_day.mean(dim='time')
```

[37]:
```
# add helpful unit label
SSH_global_mean_day.attrs['units']='m'
```

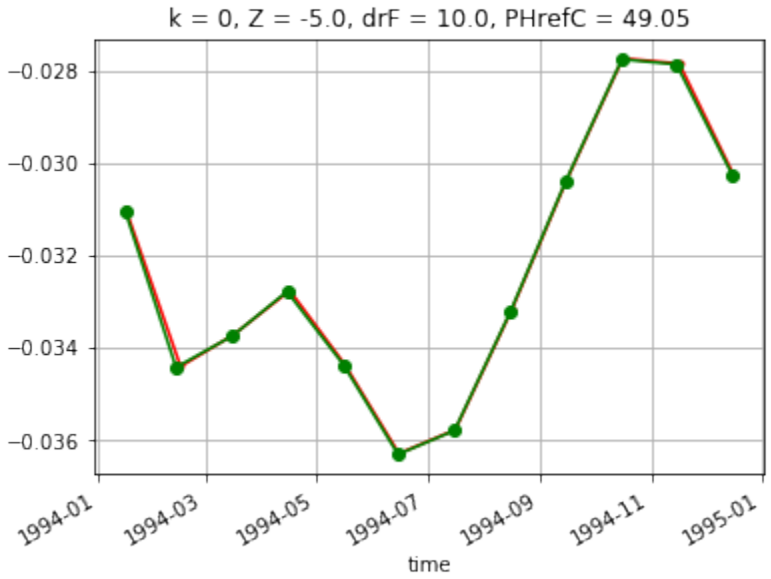
[38]:
```
# and plot for fun
SSH_global_mean_day.plot(color='k');plt.grid()
```



[39]:
```
SSH_global_mean_mon_alt = SSH_global_mean_day.resample(time='1M', loffset='-15D').mean()
```

Plot to compare.

[40]:
```
SSH_global_mean_mon.sel(time='1994').plot(color='r', marker='.');
SSH_global_mean_mon_alt.sel(time='1994').plot(color='g', marker='o');
plt.grid()
```

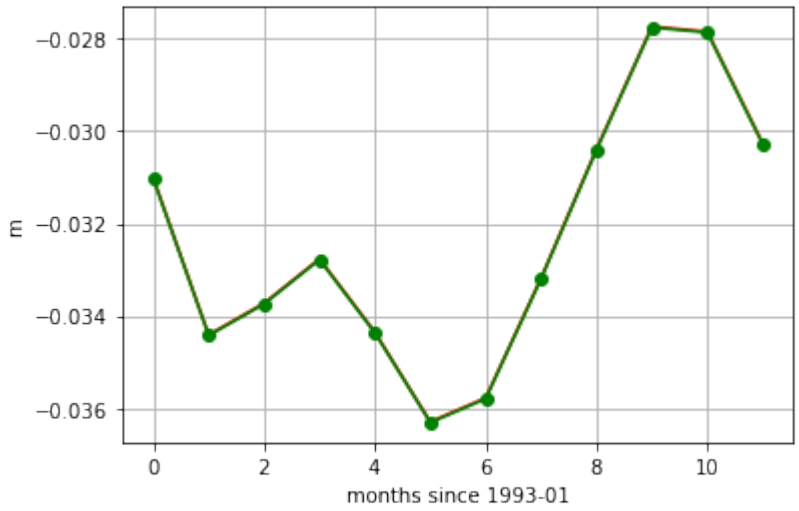


These small differences are simply an artifact of the time indexing. We used loffset='15D' to shift the time of the monthly mean SSH back 15 days, close to the center of the month. The SSH_global_mean_mon field is centered exactly at the middle of the month, and since months aren't exactly 30 days long, this results in a small discrepancy when plotting with a time x-axis. If we plot without a time object x axis we find the values to be the same. That's because ECCO monthly means are calculated over calendar months.

[41]:
```
print ('date in middle of month')
print(SSH_global_mean_mon.time.values[0:2])
print ('\ndate with 15 day offset from the end of the month')
print(SSH_global_mean_mon_alt.time.values[0:2])
```

```
date in middle of month
['1993-01-16T12:00:00.000000000' '1993-02-15T12:00:00.000000000']

date with 15 day offset from the end of the month
['1993-01-16T00:00:00.000000000' '1993-02-13T00:00:00.000000000']
```

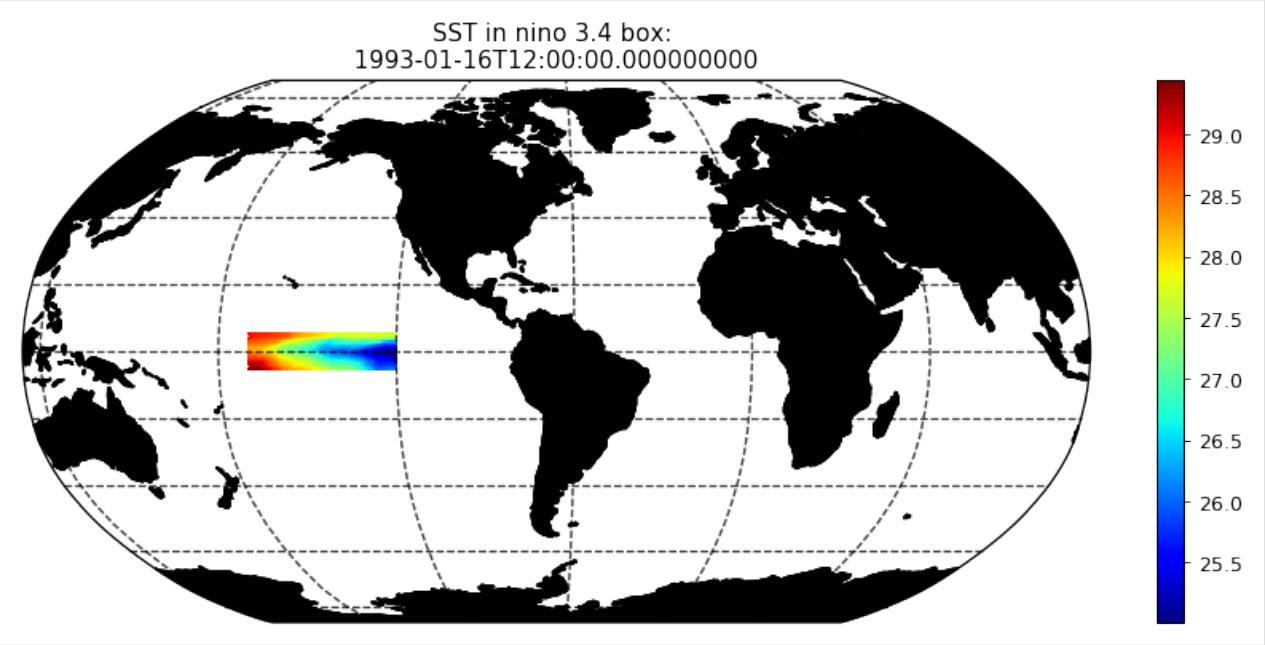
[42]:
```
plt.plot(SSH_global_mean_mon.sel(time='1994').values, color='r', marker='.');
plt.plot(SSH_global_mean_mon_alt.sel(time='1994').values, color='g', marker='o');
plt.xlabel('months since 1993-01');
plt.ylabel('m')
plt.grid()
```



1.17.6 Regional calculations with THETA

[43]:
```
lat_bounds = np.logical_and(ecco_monthly_ds.YC >= -5, ecco_monthly_ds.YC <= 5)
lon_bounds = np.logical_and(ecco_monthly_ds.XC >= -170, ecco_monthly_ds.XC <= -120)

SST = ecco_monthly_ds.THETA.isel(k=0)
SST_masked=SST.where(np.logical_and(lat_bounds, lon_bounds))
```

[44]:
```
plt.figure(figsize=(12,5), dpi= 90)

ecco.plot_proj_to_latlon_grid(ecco_monthly_ds.XC, \
                              ecco_monthly_ds.YC, \
                              SST_masked.isel(time=0),\
                              user_lon_0 = -66,\
                              show_colorbar=True);

plt.title('SST in nino 3.4 box: \n %s ' % str(ecco_monthly_ds.time[0].values));
```



[45]:
```
# Create the same mask for the grid cell area
rA_masked=ecco_monthly_ds.rA.where(np.logical_and(lat_bounds, lon_bounds));

# Calculate the area-weighted mean in the box
SST_masked_mean=(SST_masked*rA_masked).sum(dim=['tile','j','i'])/np.sum(rA_masked)

# Substract the temporal mean from the area-weighted mean to get a time series, the Nino␣
↪3.4 index
SST_nino_34_anom_ECCO_monthly_mean = SST_masked_mean - np.mean(SST_masked_mean)
```

Load up the Nino 3.4 index values from ESRL

[46]:
```
# https://www.esrl.noaa.gov/psd/gcos_wgsp/Timeseries/Nino34/
# https://www.esrl.noaa.gov/psd/gcos_wgsp/Timeseries/Data/nino34.long.anom.data
# NINA34
# 5N-5S 170W-120W
# HadISST
#  Anomaly from 1981-2010
#  units=degC

import urllib.request
data = urllib.request.urlopen('https://www.esrl.noaa.gov/psd/gcos_wgsp/Timeseries/Data/
↪nino34.long.anom.data')

# the following code parses the ESRL text file and puts monthly-mean nino 3.4 values␣
↪into an array
start_year = 1993
end_year = 2015
num_years = end_year-start_year+1
nino34_noaa = np.zeros((num_years, 12))
```

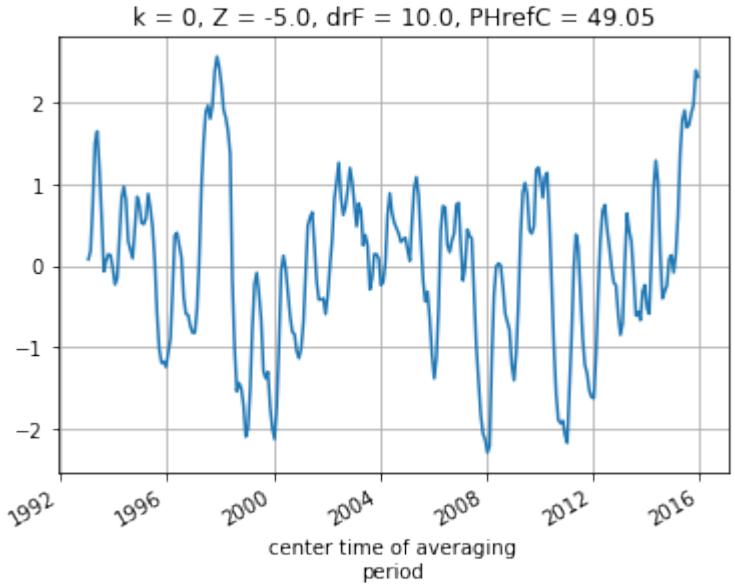
(continues on next page)

(continued from previous page)

```
for i,l in enumerate(data):
    line_str = str(l, "utf-8")
    x=line_str.split()
    try:
        year = int(x[0])
        row_i = year-start_year
        if row_i >= 0 and year <= end_year:

            print('loading nino 3.4 for year %s  row %s' % (year, row_i))

            for m in range(0,12):
                nino34_noaa[row_i, m] = float(x[m+1])
    except:
        continue
```

```
loading nino 3.4 for year 1993  row 0
loading nino 3.4 for year 1994  row 1
loading nino 3.4 for year 1995  row 2
loading nino 3.4 for year 1996  row 3
loading nino 3.4 for year 1997  row 4
loading nino 3.4 for year 1998  row 5
loading nino 3.4 for year 1999  row 6
loading nino 3.4 for year 2000  row 7
loading nino 3.4 for year 2001  row 8
loading nino 3.4 for year 2002  row 9
loading nino 3.4 for year 2003  row 10
loading nino 3.4 for year 2004  row 11
loading nino 3.4 for year 2005  row 12
loading nino 3.4 for year 2006  row 13
loading nino 3.4 for year 2007  row 14
loading nino 3.4 for year 2008  row 15
loading nino 3.4 for year 2009  row 16
loading nino 3.4 for year 2010  row 17
loading nino 3.4 for year 2011  row 18
loading nino 3.4 for year 2012  row 19
loading nino 3.4 for year 2013  row 20
loading nino 3.4 for year 2014  row 21
loading nino 3.4 for year 2015  row 22
```

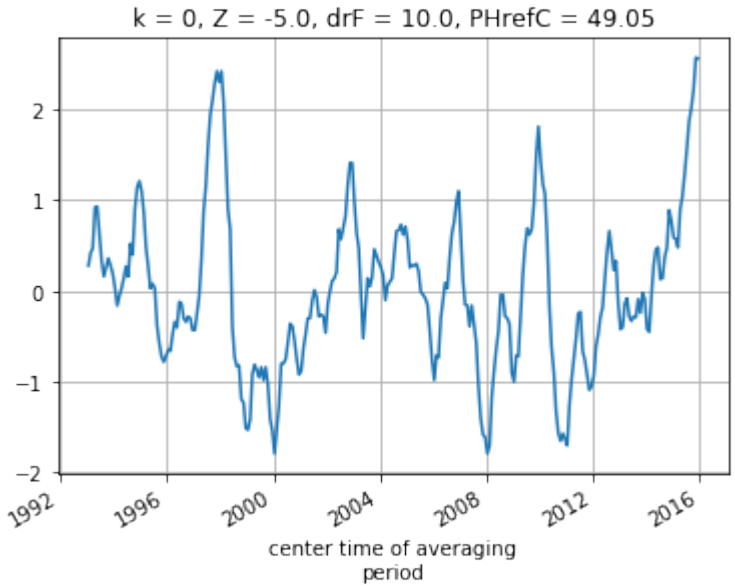
[47]:
```
SST_nino_34_anom_ECCO_monthly_mean.plot();plt.grid()
```



we'll make a new DataArray for the NOAA SST nino_34 data by copying the DataArryay for the ECCO SST data and replacing the values

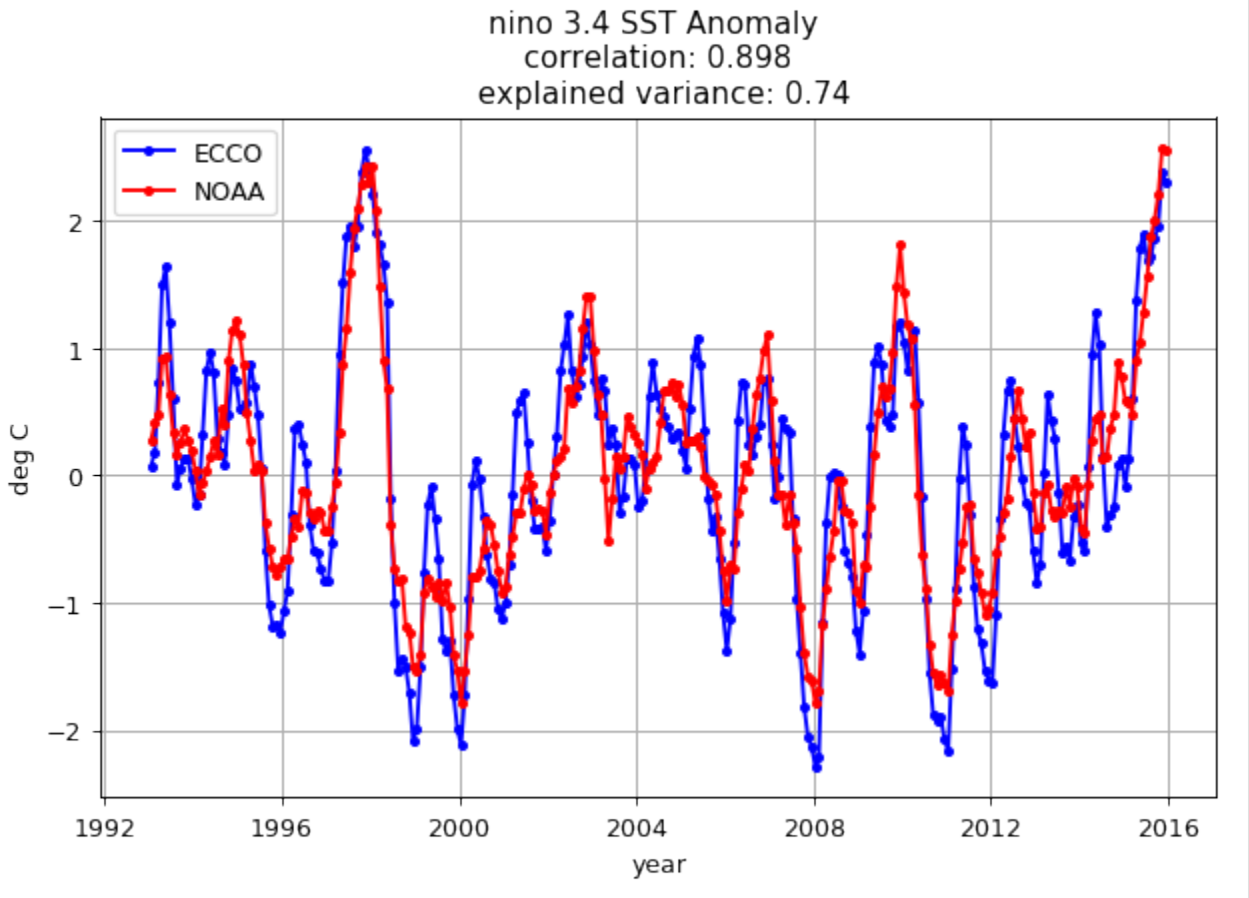
[48]:
```
SST_nino_34_anom_NOAA_monthly_mean = SST_nino_34_anom_ECCO_monthly_mean.copy(deep=True)
SST_nino_34_anom_NOAA_monthly_mean.values[:] = nino34_noaa.ravel()
```

[49]:
```
SST_nino_34_anom_NOAA_monthly_mean.plot();plt.grid()
```



Plot the ECCOv4r3 and ESRL nino 3.4 index

[50]:
```
# calculate correlation between time series
nino_corr = np.corrcoef(SST_nino_34_anom_ECCO_monthly_mean, SST_nino_34_anom_NOAA_
↪monthly_mean)[1]
nino_ev   = 1 - np.var(SST_nino_34_anom_ECCO_monthly_mean-SST_nino_34_anom_NOAA_monthly_
↪mean)/np.var(SST_nino_34_anom_NOAA_monthly_mean)
plt.figure(figsize=(8,5), dpi= 90)
plt.plot(SST_nino_34_anom_ECCO_monthly_mean.time, \
             SST_nino_34_anom_ECCO_monthly_mean - SST_nino_34_anom_ECCO_monthly_mean.
↪mean(),'b.-')
plt.plot(SST_nino_34_anom_NOAA_monthly_mean.time, \
             SST_nino_34_anom_NOAA_monthly_mean - SST_nino_34_anom_NOAA_monthly_mean.
↪mean(),'r.-')
plt.title('nino 3.4 SST Anomaly \n correlation: %s \n explained variance: %s' % (np.
↪round(nino_corr[0],3), \
                                                                                np.
↪round(nino_ev.values,3)));
plt.legend(('ECCO','NOAA'))
plt.ylabel('deg C');
plt.xlabel('year');
plt.grid()
```



ECCO is able to match the NOAA Nino 3.4 index faily well well.

1.17.7 Conclusion

You should now be familiar with doing some doing calculations using scalar quantities.

Suggested exercises

1. Create the SSH time series from 1992-2015
2. Create the global mean sea level trend (map) from 1992-2015
3. Create the global mean sea level trend (map) for two epochs 1992-2003, 2003-2015
4. Compare other nino indices

[ ]:

1.18 Compute meridional heat transport

This notebook shows how to compute meridional heat transport (MHT) across any particular latitude band. Additionally, we show this for both global and basin specific cases.

Oceanographic computations:

- use xgcm to compute masks and grab values at a particular latitude band
- use ecco_v4_py to select a specific basin
- compute meridional heat transport at one or more latitude bands

Python basics on display:

- how to open a dataset using xarray (a one liner!)
- how to save a dataset using xarray (another one liner!)
- one method for making subplots
- some tricks for plotting quantities defined as dask arrays

Note that each of these tasks can be accomplished more succinctly with ecco_v4_py functions, but are shown explicitly to illustrate these tools. Throughout, we will note the ecco_v4_py (python) and gcmfaces (MATLAB) functions which can perform these computations.

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
import os
import matplotlib.pyplot as plt
import numpy as np
import xarray as xr
import cartopy as cart
import sys
```

1.18.1 Load Model Variables

Because we're computing transport, we want the files containing 'UVELMASS' and 'VVELMASS' for volumetric transport, and 'ADVx_TH', 'ADVy_TH' and 'DFxE_TH', 'DFyE_TH' for the advective and diffusive components of heat transport, respectively.

[3]:
```
## Import the ecco_v4_py library into Python
## =========================================

## -- If ecco_v4_py is not installed in your local Python library,
##    tell Python where to find it.  For example, if your ecco_v4_py
##    files are in /Users/ifenty/ECCOv4-py/ecco_v4_py, then use:

sys.path.append('/home/ifenty/ECCOv4-py')
import ecco_v4_py as ecco
```

[4]:
```
## Set top-level file directory for the ECCO NetCDF files
## ======================================================
# base_dir = '/home/username/'
base_dir = '/home/ifenty/ECCOv4-release'

## define a high-level directory for ECCO fields
ECCO_dir = base_dir + '/Release3_alt'
```

[5]:
```
## Load the model grid
grid_dir= ECCO_dir + '/nctiles_grid/'

ecco_grid = ecco.load_ecco_grid_nc(grid_dir, 'ECCOv4r3_grid.nc')

## Load one year of 2D daily data, SSH, SST, and SSS
data_dir= ECCO_dir + '/nctiles_monthly'

ecco_vars = ecco.recursive_load_ecco_var_from_years_nc(data_dir, \
                                          vars_to_load=['ADVx_TH', 'ADVy_TH', \
                                                        'DFxE_TH', 'DFyE_TH'],\
                                          years_to_load = 'all')
## Merge the ecco_grid with the ecco_vars to make the ecco_ds
ecco_ds = xr.merge((ecco_grid , ecco_vars))
```

```
loading files of  ADVx_TH
loading files of  ADVy_TH
loading files of  DFxE_TH
loading files of  DFyE_TH
```

1.18.2 Grab latitude band: 26°N array as an example

Here we want to grab the transport values which along the band closest represented in the model to 26°N. In a latitude longitude grid this could simply be done by, e.g. `U.sel(lat=26)`. However, the LLC grid is slightly more complicated. Luckily, the functionality enabled by the xgcm Grid object makes this relatively easy.

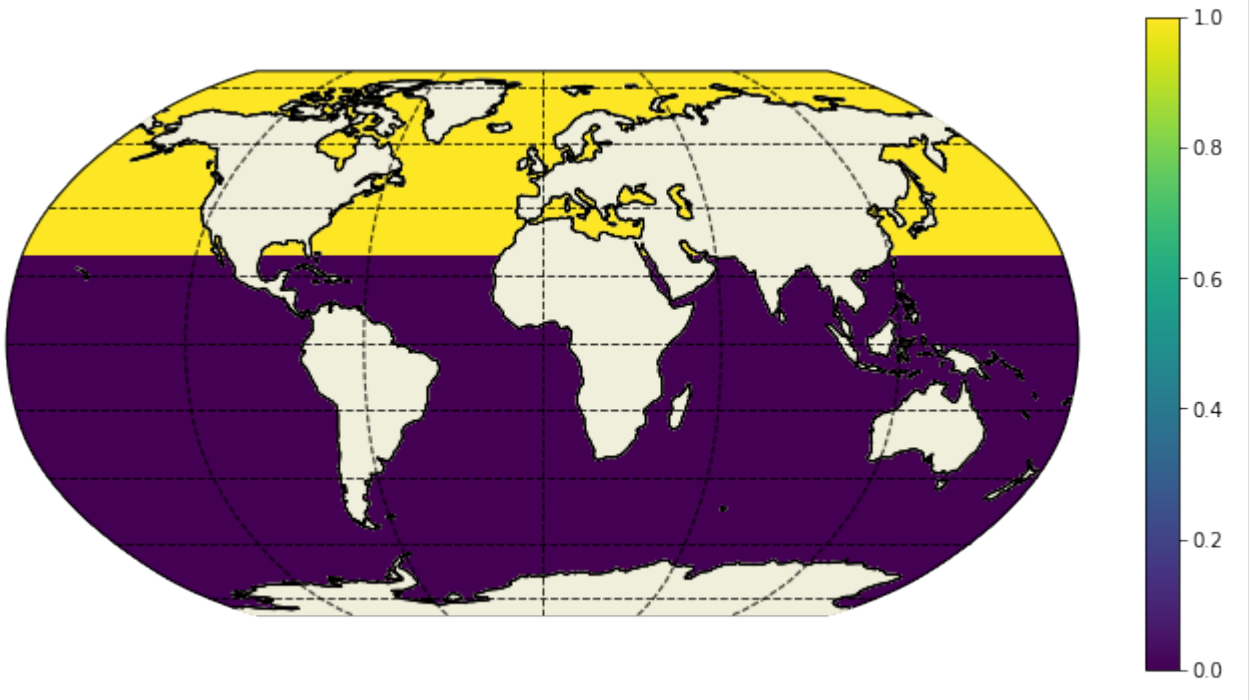
Note that this subsection can be performed with the with the ecco_v4_py modules vector_calc and scalar_calc as follows:

```
from ecco_v4_py import vector_calc, scalar_calc
grid = ecco_v4_py.get_llc_grid(ds)
rapid_maskW, rapid_maskS = vector_calc.get_latitude_masks(lat_val=26,yc=ds.YC,grid=grid)
rapid_maskC = scalar_calc.get_latitude_mask(lat_val=26,yc=ds.YC,grid=grid)
```

One can also use the gcmfaces function gcmfaces_calc/gcmfaces_lines_zonal.m.

[6]:
```
# Get array of 1's at and north of latitude
lat = 26
ones = xr.ones_like(ecco_ds.YC)
dome_maskC = ones.where(ecco_ds.YC>=lat,0)
```

[7]:
```
plt.figure(figsize=(12,6))
ecco.plot_proj_to_latlon_grid(ecco_ds.XC,ecco_ds.YC,dome_maskC,
                              projection_type='robin',cmap='viridis',user_lon_0=0,show_
↪colorbar=True);
```



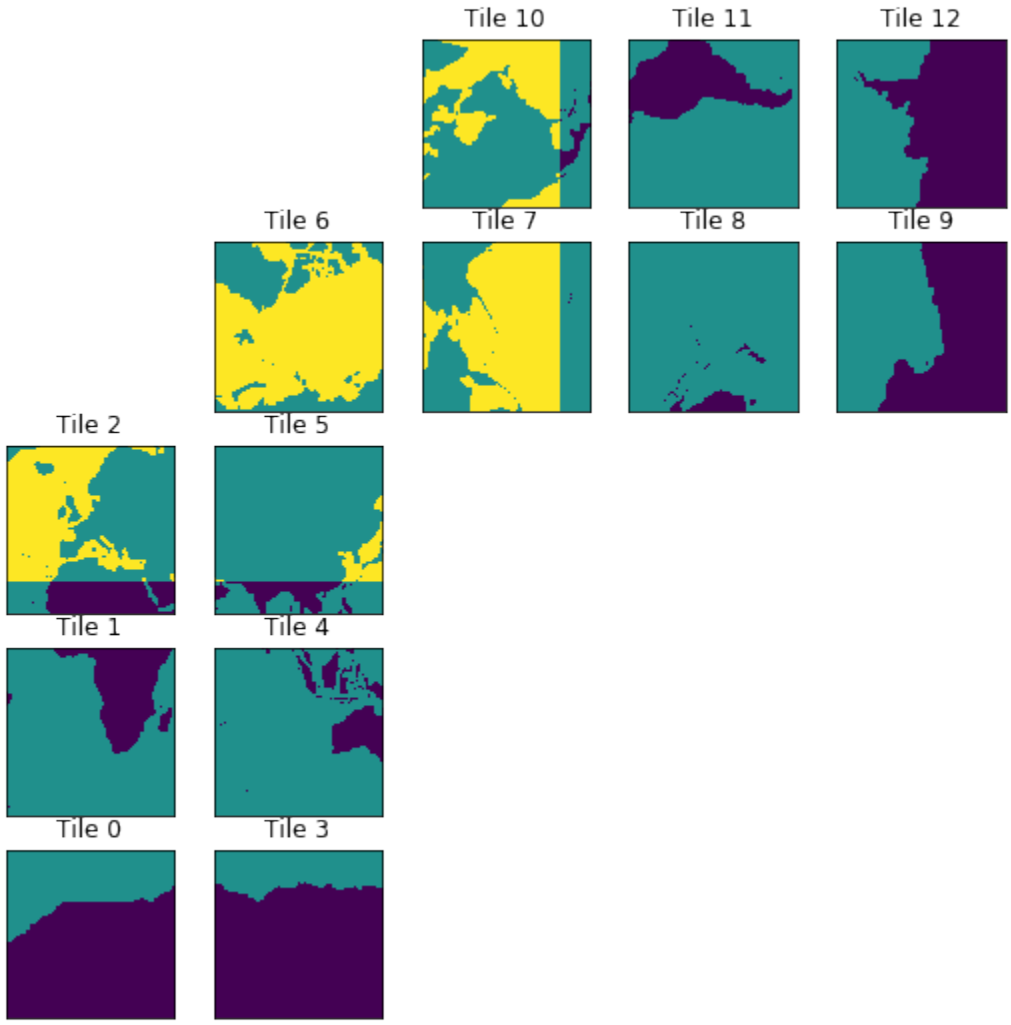
Again, if this were a lat/lon grid we could simply take a finite difference in the meridional direction. The only grid cells with 1's remaining would be at the southern edge of grid cells at approximately 26°N.

However, recall that the LLC grid has a different picture.

[8]:
```
maskC = ecco_ds.maskC
maskS = ecco_ds.maskS
maskW = ecco_ds.maskW
```

[9]:
```
plt.figure(figsize=(12,6))
ecco.plot_tiles(dome_maskC+maskC.isel(k=0), cmap='viridis');
```

```
<Figure size 864x432 with 0 Axes>
```



Recall that for tiles 7-12, the y-dimension actually runs East-West. Therefore, we want to compute a finite difference in the x-dimension on these tiles to get the latitude band at 26°N. For tiles 1-5, we clearly want to difference in the y-dimension. Things get more complicated on tile 6.

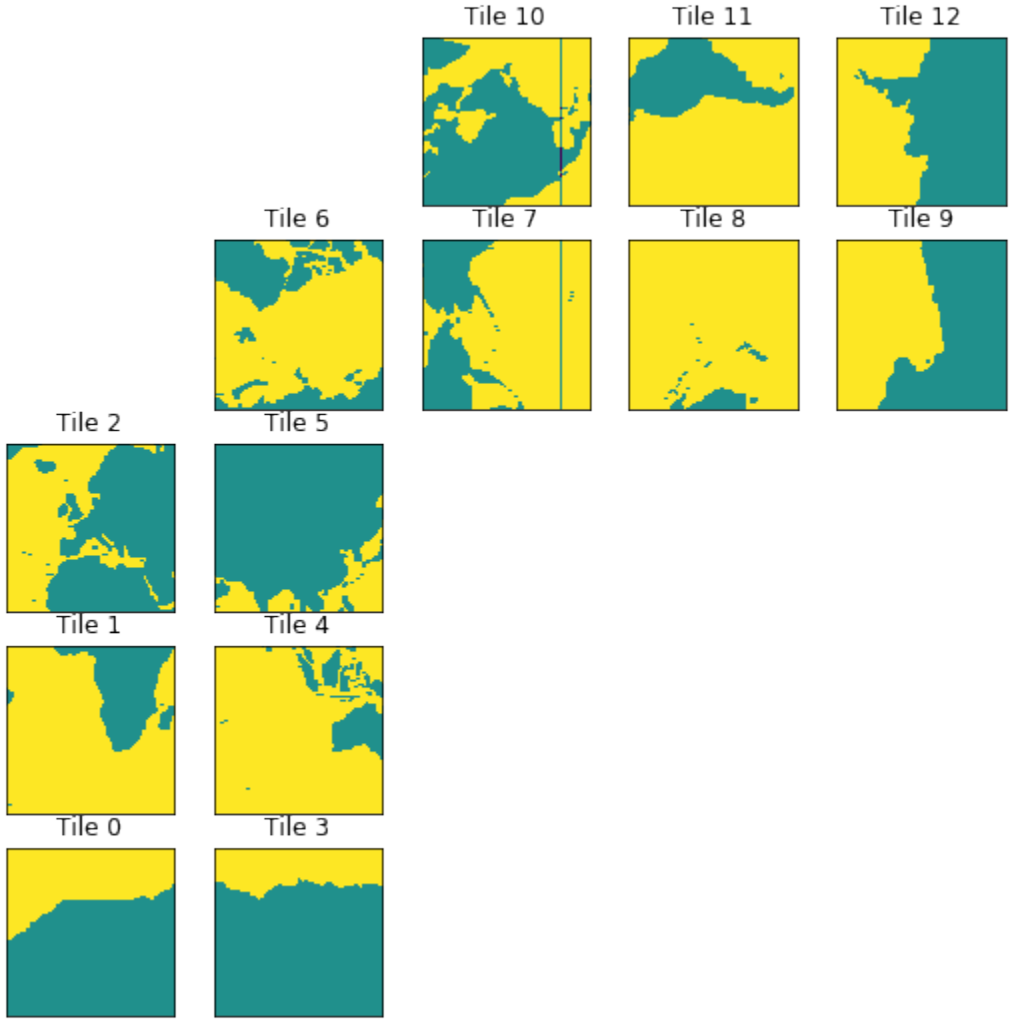
Here we make the xgcm Grid object which allows us to compute finite differences in simple one liners. This object understands how each of the tiles on the LLC grid connect, because we provide that information. To see under the hood, checkout the utility function get_llc_grid where these connections are defined.

[10]:
```
grid = ecco.get_llc_grid(ecco_ds)
```

[11]:
```
lat_maskW = grid.diff(dome_maskC,'X',boundary='fill')
lat_maskS = grid.diff(dome_maskC,'Y',boundary='fill')
```

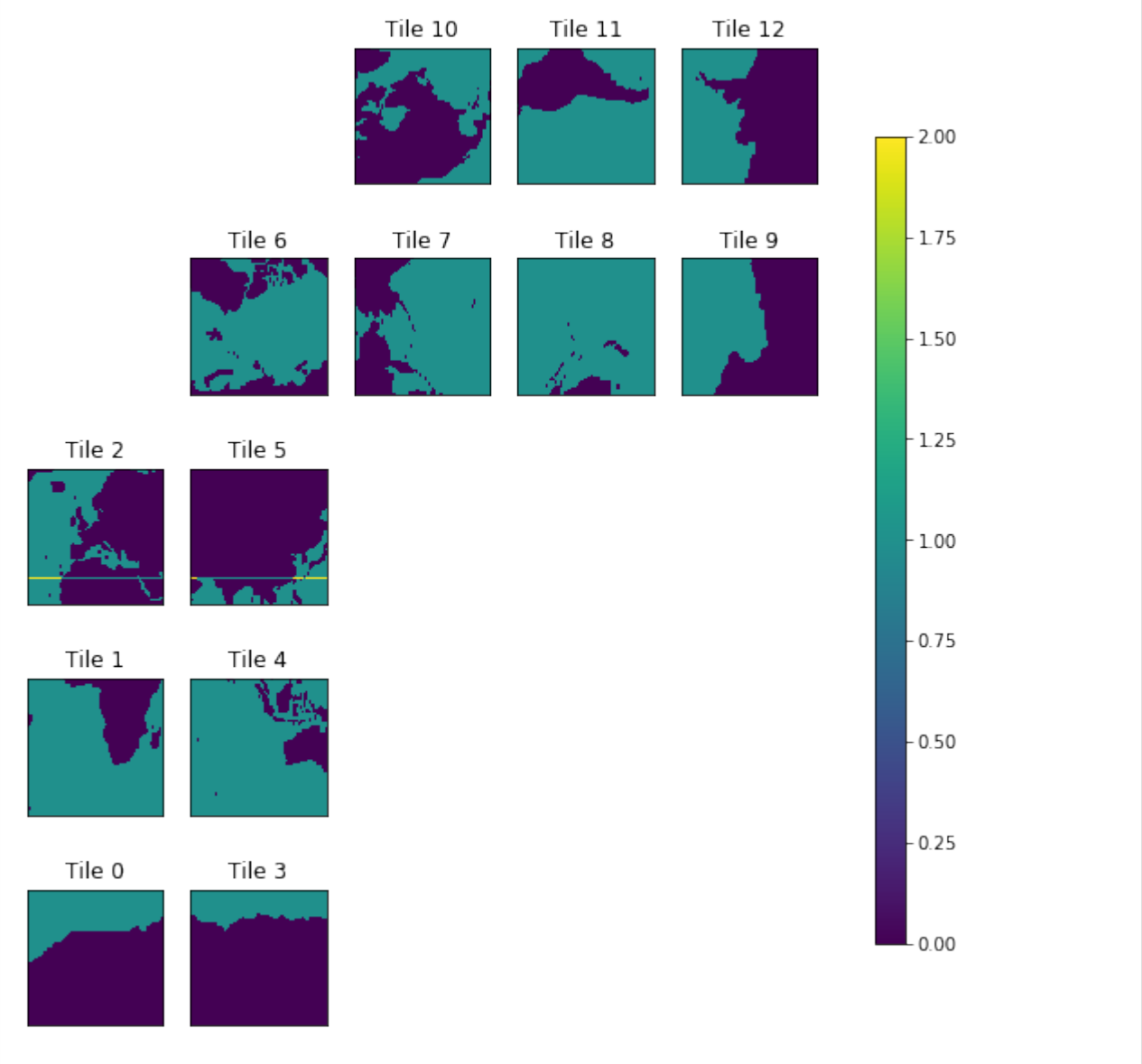
[12]:
```
plt.figure(figsize=(12,6))
ecco.plot_tiles(lat_maskW+maskW.isel(k=0), cmap='viridis');
```

```
<Figure size 864x432 with 0 Axes>
```



[13]:
```
plt.figure(figsize=(12,6))
ecco.plot_tiles(lat_maskS+maskS.isel(k=0), cmap='viridis',show_colorbar=True);
```

```
<Figure size 864x432 with 0 Axes>
```



1.18.3 Select the Atlantic ocean basin for RAPID-MOCHA MHT

Now that we have 26°N we just need to select the Atlantic. This can be done with the ecco_v4_py.get_basin module, specifically `ecco_v4_py.get_basin.get_basin_mask`. Note that this function requires a mask as an input, and then returns the values within a particular ocean basin. Therefore, provide the function with `ds['maskC']` for a mask at tracer points, `ds['maskW']` for west grid cell edges, etc.

Note: this mirrors the gcmfaces functionality ecco_v4/v4_basin.m.

Here we just want the Atlantic ocean, but lets look at what options we have …

[14]:
```
print(ecco.get_available_basin_names())
```

```
['pac', 'atl', 'ind', 'arct', 'bering', 'southChina', 'mexico', 'okhotsk', 'hudson', 'med
↪', 'java', 'north', 'japan', 'timor', 'eastChina', 'red', 'gulf', 'baffin', 'gin',
↪'barents']
```

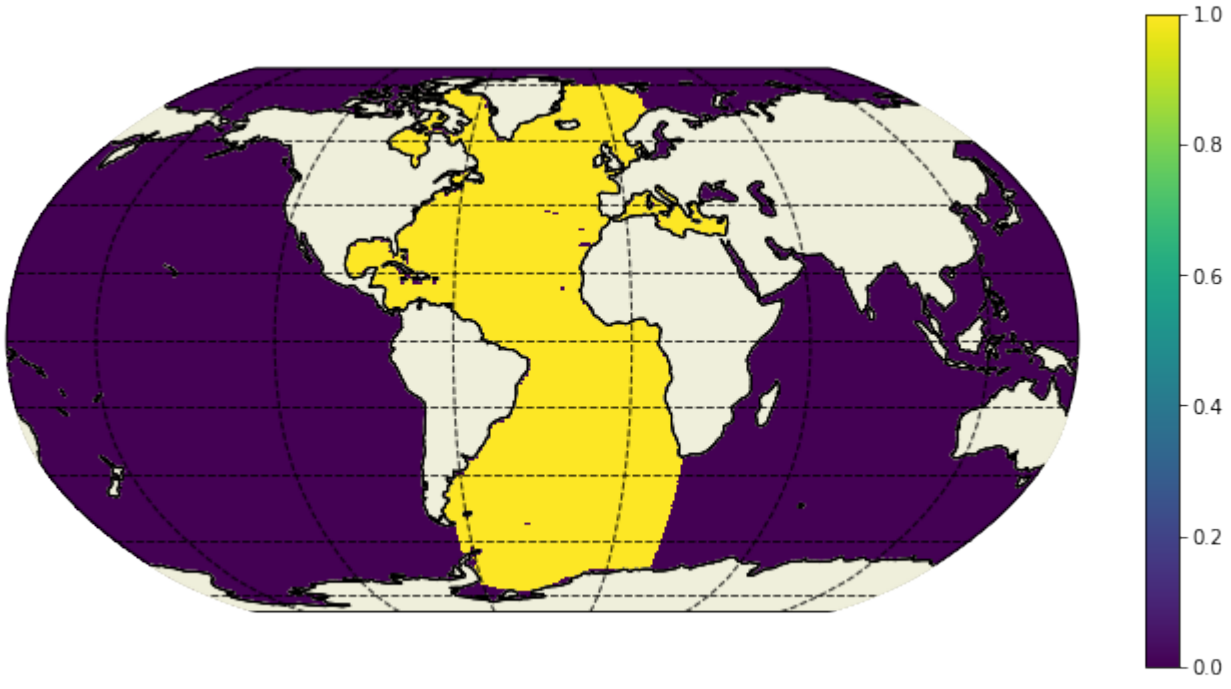
Notice that, for instance, 'mexico' exists separately from the Atlantic ('atl'). This is to provide as fine granularity as possible (and sensible). To grab the broader Atlantic ocean basin, i.e. the one people usually refer to, use the option 'atlExt'. Similar options exist for the Pacific and Indian ocean basins.

[15]:
```
atl_maskW = ecco.get_basin_mask(basin_name='atlExt',mask=maskW.isel(k=0))
atl_maskS = ecco.get_basin_mask(basin_name='atlExt',mask=maskS.isel(k=0))
```

```
loading  basins.data
data shape  (1170, 90)
dims, num_dims, llc  (1170, 90) 2 90
2 dimensions
f3 shape  (90, 90)
f5 shape  (90, 270)
2D, data_compact shape  (1170, 90)
data_tiles shape  (13, 90, 90)
data_tiles shape =  (13, 90, 90)
loading  basins.data
data shape  (1170, 90)
dims, num_dims, llc  (1170, 90) 2 90
2 dimensions
f3 shape  (90, 90)
f5 shape  (90, 270)
2D, data_compact shape  (1170, 90)
data_tiles shape  (13, 90, 90)
data_tiles shape =  (13, 90, 90)
```

Notice that we pass the routine a 2D mask by selecting the first depth level. This is simply to make things run faster.

[16]:
```
plt.figure(figsize=(12,6))
ecco.plot_proj_to_latlon_grid(ecco_ds.XC,ecco_ds.YC,atl_maskW,
                              projection_type='robin',cmap='viridis',user_lon_0=-30,show_
↪colorbar=True);
```



1.18.4 MHT at the approximate RAPID array latitude

This can be done with the ecco_v4_py.calc_meridional_trsp module for heat, salt, and volume transport as follows:

```
mvt = ecco_v4_py.calc_meridional_vol_trsp(ecco_ds,lat_vals=26,basin_name='atlExt')
mht = ecco_v4_py.calc_meridional_heat_trsp(ecco_ds,lat_vals=26,basin_name='atlExt')
mst = ecco_v4_py.calc_meridional_salt_trsp(ecco_ds,lat_vals=26,basin_name='atlExt')
```

Additionally, one could compute the overturning streamfunction at this latitude band with `ecco_v4_py.calc_meridional_stf`. The inputs are the same as the functions above, see the module ecco_v4_py.calc_stf.

In MATLAB, one can use the functions:

- compute meridional transports: gcmfaces_calc/calc_MeridionalTransport.m
- compute the overturning streamfunction: gcmfaces_calc/calc_overturn.m

A note about computational performance

When we originally open the dataset with all of the variables, we don't actually load anything into memory. In fact, nothing actually happens until "the last minute". For example, the data are only loaded once we do any computation like compute a mean value, plot something, or explicitly provide a `load` command for either the entire dataset or an individual DataArray within the dataset. This 'lazy execution' is enabled by the data structure underlying the xarray Datasets and DataArrays, the dask array.

In short, the when the data are opened, dask builds an execution task graph which it saves up to execute at the last minute. Dask also allows for parallelism, and by default runs in parallel across threads for a single core architecture. For now, this is what we will show.

Some quick tips are:

- Don't load the full 25 years of ECCOv4r3 output unless you're on a machine with plenty of memory. I am doing this in the cell below because I'm on a Skylake node with 192GB. Proceed with caution before copying and pasting.
- If you're in this situation where you can't load all months into memory, it's a good idea to load a final result before plotting, in case you need to plot it many times in a row, see below...

[17]:
```
%%time
ecco_ds['ADVx_TH'].load();
ecco_ds['DFxE_TH'].load();
ecco_ds['ADVy_TH'].load();
ecco_ds['DFyE_TH'].load();
ecco_ds['dxG'].load();
ecco_ds['dyG'].load();
ecco_ds['drF'].load();
```

```
CPU times: user 15.8 s, sys: 52.1 s, total: 1min 7s
Wall time: 47.8 s
```

[17]:
```
<xarray.DataArray 'drF' (k: 50)>
array([ 10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.01,
        10.03,  10.11,  10.32,  10.8 ,  11.76,  13.42,  16.04,  19.82,
        24.85,  31.1 ,  38.42,  46.5 ,  55.  ,  63.5 ,  71.58,  78.9 ,
        85.15,  90.18,  93.96,  96.58,  98.25,  99.25, 100.01, 101.33,
       104.56, 111.33, 122.83, 139.09, 158.94, 180.83, 203.55, 226.5 ,
       249.5 , 272.5 , 295.5 , 318.5 , 341.5 , 364.5 , 387.5 , 410.5 ,
```

(continues on next page)

(continued from previous page)

```
       433.5 , 456.5 ], dtype=float32)
Coordinates:
  * k        (k) int64 0 1 2 3 4 5 6 7 8 9 10 ... 40 41 42 43 44 45 46 47 48 49
    Z        (k) float32 -5.0 -15.0 -25.0 -35.0 ... -5039.25 -5461.25 -5906.25
    drF      (k) float32 10.0 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    PHrefC   (k) float32 49.05 147.15 245.25 ... 49435.043 53574.863 57940.312
Attributes:
    standard_name:  cell_z_size
    long_name:      cell z size
    units:          m
```

[18]:
```
%%time
trsp_x = ((ecco_ds['ADVx_TH'] + ecco_ds['DFxE_TH']) *  atl_maskW).sum('k')
trsp_y = ((ecco_ds['ADVy_TH'] + ecco_ds['DFyE_TH']) *  atl_maskS).sum('k')
```

```
CPU times: user 26.2 s, sys: 44.2 s, total: 1min 10s
Wall time: 1min 10s
```

[19]:
```
%%time
trsp_x = trsp_x * lat_maskW
trsp_y = trsp_y * lat_maskS

# Sum horizontally
trsp_x = trsp_x.sum(dim=['i_g','j','tile'])
trsp_y = trsp_y.sum(dim=['i','j_g','tile'])

# Add together to get transport at depth level
trsp_at_depth = trsp_x + trsp_y

# Sum over full depth and convert to PW
mht = (trsp_at_depth *10**-15 * 1000 * 4000)
mht.attrs['units']='PW'
```

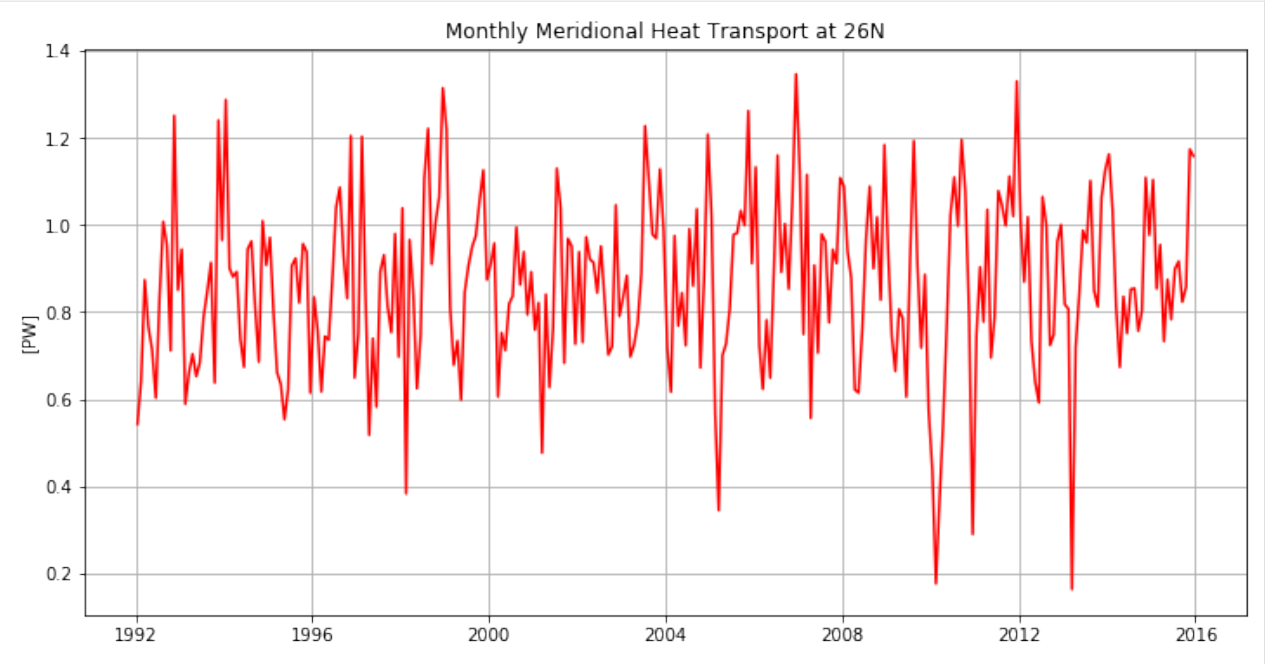
```
CPU times: user 388 ms, sys: 788 ms, total: 1.18 s
Wall time: 1.17 s
```

Now that we have computed MHT, lets load the result for iterative plotting

For some reason when dask arrays are plotted, they are computed on the spot but don't stay in memory. This takes a bit to get the hang of, but keep in mind that this allows us to scale the same code on distributed architecture, so we could use these same routines for high resolution output. This seems worthwhile!

Note that we probably don't need this load statement if we have already loaded the underlying datasets.

[20]:
```
plt.figure(figsize=(12,6))
plt.plot(mht['time'],mht,'r')
plt.grid()
plt.title('Monthly Meridional Heat Transport at 26N')
plt.ylabel(f'[{mht.attrs["units"]}]')
plt.show()
```



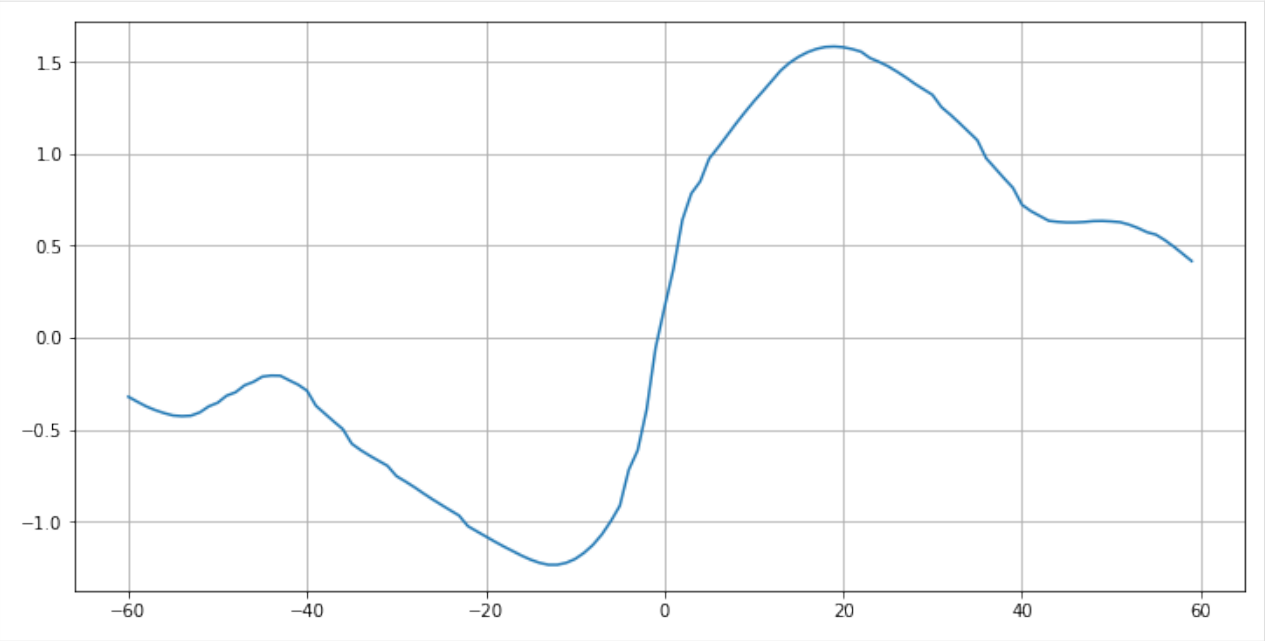
1.18.5 Now compare global and Atlantic MHT at many latitudes

[21]:
```
# pick a temp directory for yourself
nc_save_dir = '/home/ifenty/tmp/'

if not os.path.isdir(nc_save_dir):
    os.makedirs(nc_save_dir)

nc_file = f'{nc_save_dir}/eccov4r3_mht.nc'
```

[25]:
```
global_lats = np.arange(-60,60,1)
mht = ecco.calc_meridional_heat_trsp(ecco_ds, lat_vals=global_lats)
mht = mht.rename({'heat_trsp':'global_heat_trsp'})
mht = mht.rename({'heat_trsp_z':'global_heat_trsp_z'})
print(' --- Done with global --- ')
```

```
 --- Done with global ---
```

[66]:
```
plt.figure(figsize=(12,6))
plt.plot(mht['lat'], mht['global_heat_trsp'].mean('time'))
plt.grid();
```



[27]:
```
basin_lats = np.arange(-35,60,1)
atl = ecco.calc_meridional_heat_trsp(ecco_ds,lat_vals=basin_lats,basin_name='atlExt')
atl = atl.rename({'heat_trsp':'atl_heat_trsp'})
atl = atl.rename({'heat_trsp_z':'atl_heat_trsp_z'})
print(' --- Done with Atlantic --- ')
```

```
loading  basins.data
data shape  (1170, 90)
dims, num_dims, llc  (1170, 90) 2 90
2 dimensions
f3 shape  (90, 90)
f5 shape  (90, 270)
2D, data_compact shape  (1170, 90)
data_tiles shape  (13, 90, 90)
data_tiles shape =  (13, 90, 90)
loading  basins.data
data shape  (1170, 90)
dims, num_dims, llc  (1170, 90) 2 90
2 dimensions
f3 shape  (90, 90)
f5 shape  (90, 270)
2D, data_compact shape  (1170, 90)
data_tiles shape  (13, 90, 90)
data_tiles shape =  (13, 90, 90)
 --- Done with Atlantic ---
```

[28]:
```
atl
```

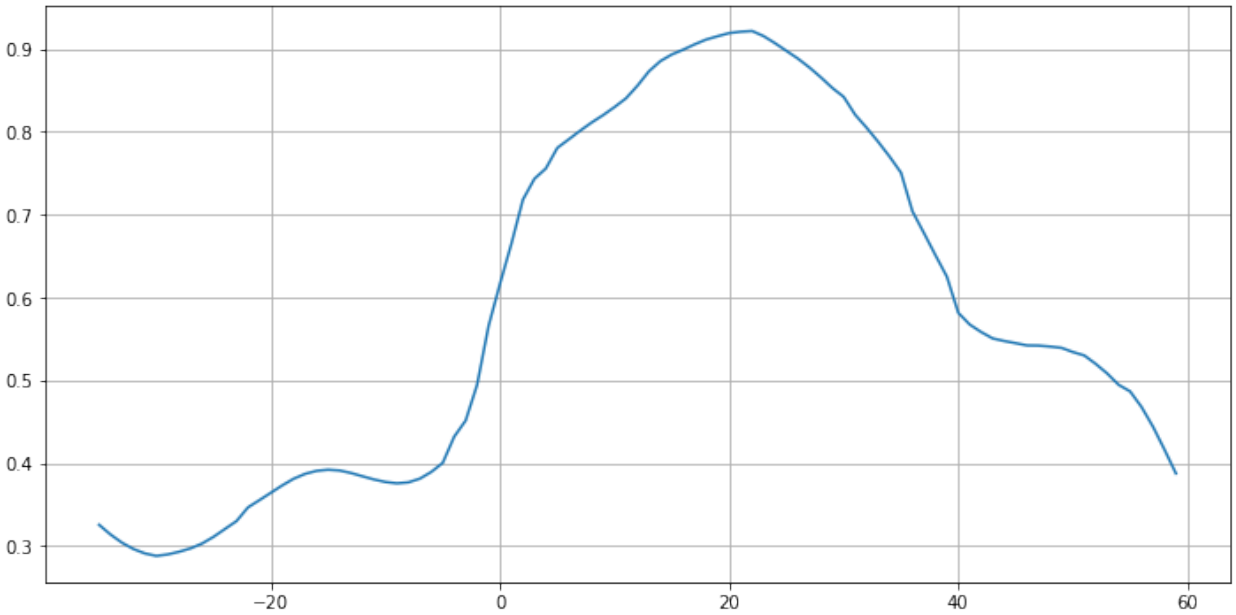
[28]:
```
<xarray.Dataset>
Dimensions:          (k: 50, lat: 95, time: 288)
Coordinates:
  * time             (time) datetime64[ns] 1992-01-16T12:00:00 ... 2015-12-16T12:00:00
  * k                (k) int64 0 1 2 3 4 5 6 7 8 ... 41 42 43 44 45 46 47 48 49
```

(continues on next page)

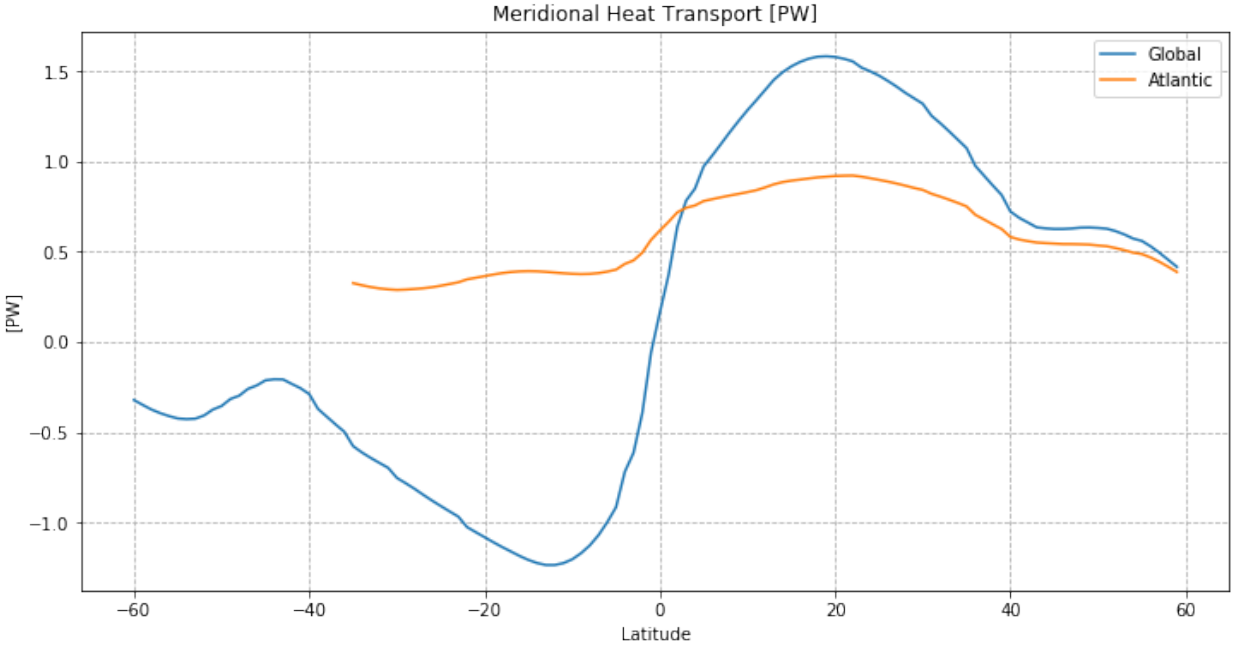
(continued from previous page)

```
  * lat              (lat) int64 -35 -34 -33 -32 -31 -30 ... 54 55 56 57 58 59
    drF              (k) float32 10.0 10.0 10.0 10.0 ... 387.5 410.5 433.5 456.5
    PHrefC           (k) float32 49.05 147.15 245.25 ... 53574.863 57940.312
    Z                (k) float32 -5.0 -15.0 -25.0 ... -5039.25 -5461.25 -5906.25
Data variables:
    atl_heat_trsp_z  (time, k, lat) float64 -0.2068 -0.2489 -0.2775 ... 0.0 0.0
    atl_heat_trsp    (time, lat) float64 0.1535 0.1445 0.1371 ... 0.5322 0.5043
```

[65]:
```
plt.figure(figsize=(12,6))
plt.plot(atl['lat'], atl['atl_heat_trsp'].mean('time'))
plt.grid()
```



[30]:
```
plt.figure(figsize=(12,6))
plt.plot(mht['lat'], mht['global_heat_trsp'].mean('time'))
plt.plot(atl['lat'], atl['atl_heat_trsp'].mean('time'))
plt.legend(('Global','Atlantic'))
plt.grid(linestyle='--')
plt.title(f'Meridional Heat Transport [{mht["global_heat_trsp"].attrs["units"]}]')
plt.ylabel(f'[{mht["global_heat_trsp"].attrs["units"]}]')
plt.xlabel('Latitude')
plt.show()
```



1.18.6 MHT as a function of depth

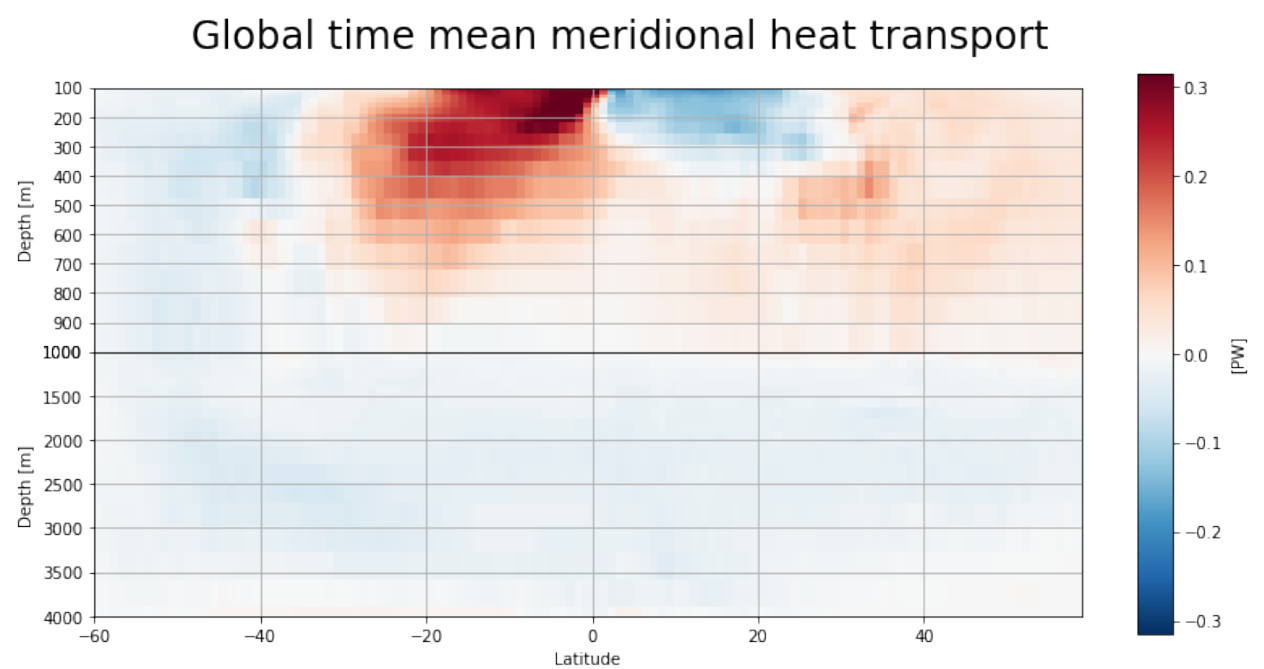
[63]:
```
def lat_depth_plot(mht,fld,label):
    fig = plt.figure(figsize=(12,6))

    # Set up depth coordinate
    depth = -mht['Z']
    stretch_depth = 1000

    # Set up colormap and colorbar
    cmap = 'RdBu_r'
    fld_mean = mht[fld].mean('time')
    abs_max = np.max(np.abs([fld_mean.min(),fld_mean.max()]))
    cmin = -abs_max*.1
    cmax = -cmin

    # First top 500m
    ax1 = plt.subplot(2,1,1)
    p1 = ax1.pcolormesh(mht['lat'],depth,fld_mean,cmap=cmap,vmin=cmin,vmax=cmax)
    plt.grid()

    # Handle y-axis
    ax1.invert_yaxis()
    plt.ylim([stretch_depth, 0])
    ax1.yaxis.axes.set_yticks(np.arange(stretch_depth,0,-100))
    plt.ylabel(f'Depth [{mht["Z"].attrs["units"]}]')

    # Remove top plot xtick label
    ax1.xaxis.axes.set_xticklabels([])
```

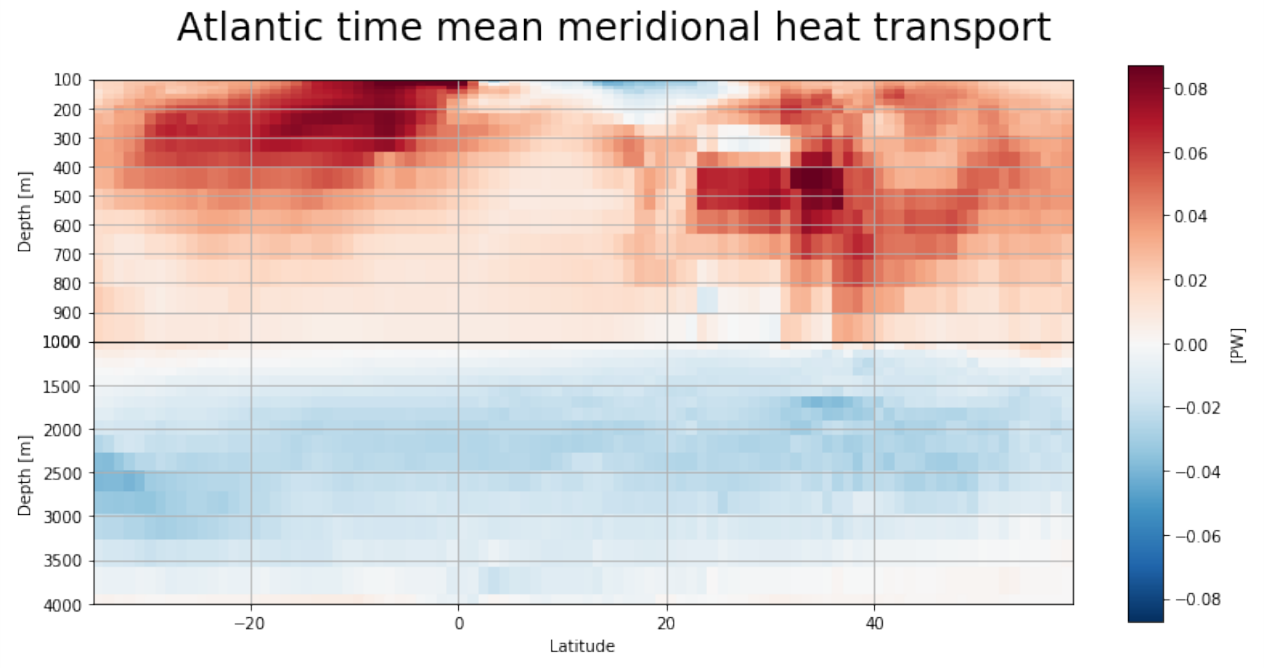
(continues on next page)

(continued from previous page)

```
    # Now the rest ...
    ax2 = plt.subplot(2,1,2)
    p2 = ax2.pcolormesh(mht['lat'],depth, fld_mean, cmap=cmap,vmin=cmin,vmax=cmax)
    plt.grid()

    # Handle y-axis
    ax2.invert_yaxis()
    plt.ylim([4000, 1000])
    #yticks = np.flip(np.arange(6000,stretch_depth,-1000))
    #ax2.yaxis.axes.set_yticks(yticks)
    plt.ylabel(f'Depth [{mht["Z"].attrs["units"]}]')

    # Label  axis
    plt.xlabel('Latitude')

    # Reduce space between subplots
    fig.subplots_adjust(hspace=0.0)

    # Make a single title
    fig.suptitle(f'{label} time mean meridional heat transport',verticalalignment='top',
↪fontsize=24)

    # Make an overyling colorbar
    fig.subplots_adjust(right=0.83)
    cbar_ax = fig.add_axes([0.87, 0.1, 0.025, 0.8])
    fig.colorbar(p2,cax=cbar_ax)
    cbar_ax.set_ylabel(f'[{mht[fld].attrs["units"]}]')

    plt.show()
```

[64]:
```
lat_depth_plot(mht,'global_heat_trsp_z','Global')
lat_depth_plot(atl,'atl_heat_trsp_z','Atlantic')
```


Global time mean meridional heat transport
Depth [m]
Depth [m]
Latitude
[PW]


Atlantic time mean meridional heat transport
Depth [m]
Depth [m]
Latitude
[PW]

1.18.7 Exercise: reproduce figure from (Ganachaud and Wunsch, 2000)

[33]:
```
from IPython.display import Image
Image('../figures/buckley_mht.png')
```

[33]:

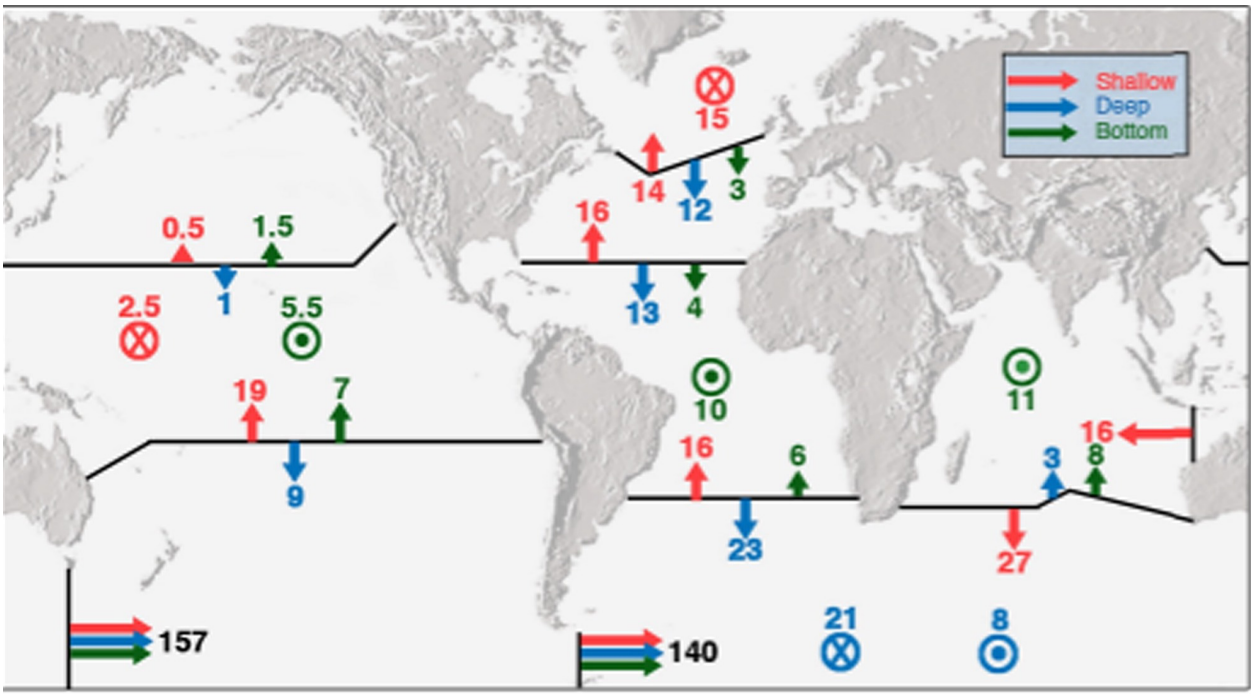


Figure from (Buckley and Marshall, 2016), which is adapted from (Ganachaud and Wunsch, 2000).

Note: to do this, you may need to pair latitude masks with those defined through the `get_section_masks` module. An example of this functionality is shown in the next tutorial.

1.18.8 References

Buckley, M. W. and Marshall, J. (2016), Observations, inferences, and mechanisms of the Atlantic Meridional Overturning Circulation: A review, Rev. Geophys., 54, doi:10.1002/2015RG000493.

Ganachaud, A., & Wunsch, C. (2000). Improved estimates of global ocean circulation, heat transport and mixing from hydrographic data. Nature, 408(6811), 453-7. doi:http://dx.doi.org.ezproxy.lib.utexas.edu/10.1038/35044048

1.19 Compute MOC along the approximate OSNAP array from ECCO

Here we compute volumetric transport along the OSNAP lines in depth space, which can be compared to recent observations.

Here we show:

- how to get masks denoting the great circle arc between two points in space
- how to compute the transport or streamfunction across this section
- a comparison to observations

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
import os
import sys
import matplotlib.pyplot as plt
import numpy as np
import xarray as xr
import cartopy as cart
import cartopy.crs as ccrs
notebook_path = os.getcwd()
```

[3]:
```
## Import the ecco_v4_py library into Python
## =========================================

## -- If ecco_v4_py is not installed in your local Python library,
##    tell Python where to find it.  For example, if your ecco_v4_py
##    files are in /Users/ifenty/ECCOv4-py/ecco_v4_py, then use:

sys.path.append('/home/ifenty/ECCOv4-py')
import ecco_v4_py as ecco
```

[4]:
```
## Set top-level file directory for the ECCO NetCDF files
## ======================================================================
# base_dir = '/home/username/'
base_dir = '/home/ifenty/ECCOv4-release'

## define a high-level directory for ECCO fields
ECCO_dir = base_dir + '/Release3_alt'
```

[5]:
```
## Load the model grid
grid_dir= ECCO_dir + '/nctiles_grid/'

ecco_grid = ecco.load_ecco_grid_nc(grid_dir, 'ECCOv4r3_grid.nc')

## Load one year of 2D daily data, SSH, SST, and SSS
data_dir= ECCO_dir + '/nctiles_monthly'

ecco_vars = ecco.recursive_load_ecco_var_from_years_nc(data_dir, \
                                          vars_to_load=['ADVx_TH', 'ADVy_TH', \
                                                        'UVELMASS', 'VVELMASS'],\
                                          years_to_load = 'all')
## Merge the ecco_grid with the ecco_vars to make the ecco_ds
ds = xr.merge((ecco_grid , ecco_vars))
```

```
loading files of  ADVx_TH
loading files of  ADVy_TH
loading files of  UVELMASS
loading files of  VVELMASS
```

1.19.1 Load O-SNAP data

These data are available at https://www.o-snap.org/observations/data/ and are published in

Lozier, M. S., Li, F., Bacon, S., Bahr, F., Bower, A. S., Cunningham, S. A., … Zhao, J. (2019). A sea change in our view of overturning in the subpolar North Atlantic. Science, 363(6426), 516–521. https://doi.org/10.1126/science.aau6592

Download these files from the o-snap data page: https://www.o-snap.org/observations/data/

OSNAP_West_East_Transports_201408_201604_2018.nc : https://duke.box.com/s/335ephjrcw09zhb8kivq8y26w4fz5eag OSNAP_Transports_201408_201604_2018.nc : https://dx.doi.org/10.7924/r4z60gf0f

and put them in a path:

[6]:
```
osnap_data_dir = '/home/ifenty/ECCOv4-py/binary_data/'
```

[7]:
```
obs1 = xr.open_dataset(osnap_data_dir + '/OSNAP_West_East_Transports_201408_201604_2018.
↪nc')
obs2 = xr.open_dataset(osnap_data_dir + '/OSNAP_Transports_201408_201604_2018.nc')
obs = xr.merge((obs1,obs2))
print(obs)
```

```
<xarray.Dataset>
Dimensions:              (TIME: 21)
Coordinates:
  * TIME                 (TIME) datetime64[ns] 2014-07-31 ... 2016-03-22
Data variables:
    MOC_WEST_SIGMA       (TIME) float64 ...
    MOC_WEST_SIGMA_ERR   (TIME) float64 ...
    MOC_WEST_Z           (TIME) float64 ...
    MOC_WEST_Z_ERR       (TIME) float64 ...
    MHT_WEST             (TIME) float64 ...
    MHT_WEST_ERR         (TIME) float64 ...
    MFT_WEST             (TIME) float64 ...
    MFT_WEST_ERR         (TIME) float64 ...
    MOC_EAST_SIGMA       (TIME) float64 ...
    MOC_EAST_SIGMA_ERR   (TIME) float64 ...
    MOC_EAST_Z           (TIME) float64 ...
    MOC_EAST_Z_ERR       (TIME) float64 ...
    MHT_EAST             (TIME) float64 ...
    MHT_EAST_ERR         (TIME) float64 ...
    MFT_EAST             (TIME) float64 ...
    MFT_EAST_ERR         (TIME) float64 ...
    MOC_SIGMA            (TIME) float64 ...
    MOC_SIGMA_ERR        (TIME) float64 ...
    MOC_Z                (TIME) float64 ...
    MOC_Z_ERR            (TIME) float64 ...
    MHT                  (TIME) float64 ...
    MHT_ERR              (TIME) float64 ...
    MFT                  (TIME) float64 ...
    MFT_ERR              (TIME) float64 ...
```

Define the OSNAP lines

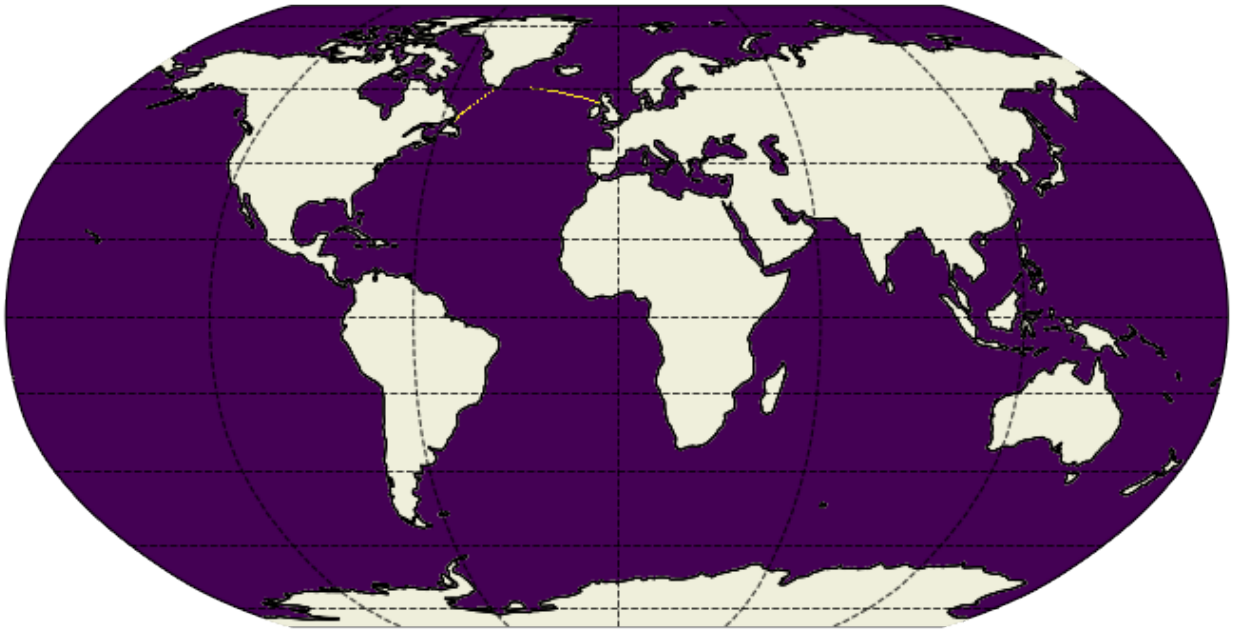
We define the OSNAP lines roughly by point pairs in `[longitude, latitude]` space. The lines are then computed via the function `ecco_v4_py.get_section_line_masks` as the great circle arc between these two points.

See below for similar MATLAB functions

[8]:
```
pt1_east = [-44, 60]
pt2_east = [-5, 56]

pt1_west = [-56, 51]
pt2_west = [-45, 60]
```

[9]:
```
maskC_east, maskW_east, maskS_east = ecco.get_section_line_masks(pt1_east,pt2_east,ds)
maskC_west, maskW_west, maskS_west = ecco.get_section_line_masks(pt1_west,pt2_west,ds)
maskC_tot = (maskC_east+maskC_west).where(maskC_east+maskC_west==1,0)
maskW_tot = (maskW_east+maskW_west).where(np.abs(maskW_east)+np.abs(maskW_west)==1,0)
maskS_tot = (maskS_east+maskS_west).where(np.abs(maskS_east)+np.abs(maskS_west)==1,0)
```

[10]:
```
plt.figure(figsize=(12,6))
ecco.plot_proj_to_latlon_grid(ds.XC,ds.YC,maskC_tot,cmap='viridis',projection_type='robin
↪',user_lon_0=0);
```

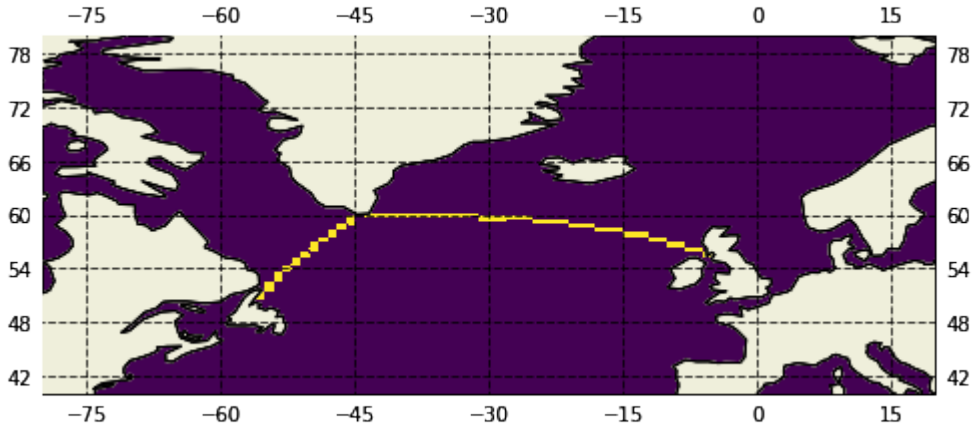


[11]:
```
plt.figure(figsize=(8,8))
# use dx=.1, dy=.1 so that plot shows the osnap array as a thin
# line.  remember, plot_proj_to_latlon_grid first interpolates
# the model grid to lat-lon with grid spacing as dx, dy
P = ecco.plot_proj_to_latlon_grid(ds.XC, ds.YC, maskC_tot, \
                                  cmap='viridis',\
                                  projection_type='PlateCaree',\
                                  lat_lim=45,dx=.1,dy=.1)
```

(continues on next page)

(continued from previous page)

```
#ax.add_feature(cart.feature.LAND,facecolor='0.7',zorder=2)
P[1].set_extent([-80, 20, 40, 80], crs=ccrs.PlateCarree())
plt.show()
```



We have defined many commonly used sections in oceanography so that users can access these easily. The available sections are shown below. This allows one to, e.g. compute volumetric transport across the Drake Passage as follows:

```
drake_vol = ecco_v4_py.calc_section_vol_trsp(ds,section_name='Drake Passage')
```

Similarly, we can do the same with `calc_section_heat_trsp` and `calc_section_salt_trsp`.

One can also get these pre-defined section masks as follows:

```
pt1,pt2 = ecco_v4_py.get_section_endpoints('Drake Passage')
maskC, maskW, maskS = ecco_v4_py.get_section_line_masks(ds,pt1,pt2)
```

Finally, one can see similar functions in MATLAB:

- define general section masks: gcmfaces_calc/gcmfaces_lines_transp.m
- see pre-defined section endpoints: gcmfaces_calc/gcmfaces_lines_pairs.m

[12]:
```
ecco.get_available_sections()
```

[12]:
```
['Bering Strait',
 'Gibraltar',
 'Florida Strait',
 'Florida Strait W1',
 'Florida Strait S1',
 'Florida Strait E1',
 'Florida Strait E2',
 'Florida Strait E3',
 'Florida Strait E4',
 'Davis Strait',
 'Denmark Strait',
 'Iceland Faroe',
 'Faroe Scotland',
 'Scotland Norway',
 'Drake Passage',
 'Indonesia W1',
```

(continues on next page)

(continued from previous page)

```
 'Indonesia W2',
 'Indonesia W3',
 'Indonesia W4',
 'Australia Antarctica',
 'Madagascar Channel',
 'Madagascar Antarctica',
 'South Africa Antarctica']
```

Compute the overturning streamfunction in depth space

The function `calc_section_stf` computes the overturning streamfunction across the plane normal to the section denoted by the west and south masks. It is also possible to compute the overturning streamfunction at a particular latitude band, as is done to compare to the RAPID array, for instance. Please see the function `calc_meridional_stf` to do this, which is also in ecco_v4_py.calc_stf.

Note that we can also compute the volumetric, heat, or salt transport across these sections using the first three functions defined in ecco_v4_py.calc_section_trsp.

In MATLAB, we can compute meridional overturning streamfunctions with gcmfaces_calc/calc_overturn.m. Section transports can be computed with gcmfaces_calc/calc_transports.m once the masks are defined.

[13]:
```
%%time
osnap_z_stf_east = ecco.calc_section_stf(ds,\
                                         pt1=pt1_east, \
                                         pt2=pt2_east,\
                                         section_name='OSNAP East Overturning␣
↪Streamfunction').compute()

osnap_z_stf_west = ecco.calc_section_stf(ds, \
                                         pt1=pt1_west, \
                                         pt2=pt2_west,\
                                         section_name='OSNAP West Overturning␣
↪Streamfunction').compute()

osnap_z_stf_tot = ecco.calc_section_stf(ds,\
                                        maskW=maskW_tot, \
                                        maskS=maskS_tot,\
                                        section_name='OSNAP Total Overturning␣
↪Streamfunction').compute()
```

```
CPU times: user 1min 52s, sys: 4min 5s, total: 5min 57s
Wall time: 3min 5s
```

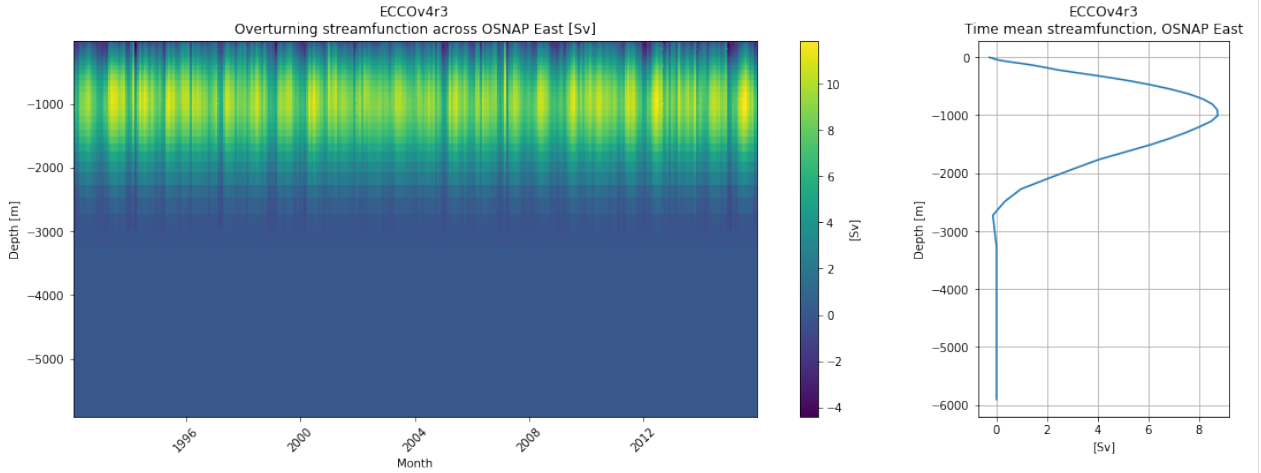
[14]:
```
def osnap_depth_stf_vs_time(stf_ds,label):
    fig = plt.figure(figsize=(18,6))

    # Time evolving
    plt.subplot(1,4,(1,3))
    plt.pcolormesh(stf_ds['time'],stf_ds['Z'],stf_ds['psi_moc'].T)
    plt.title('ECCOv4r3\nOverturning streamfunction across OSNAP %s [Sv]' % label)
    plt.ylabel('Depth [m]')
    plt.xlabel('Month')
```

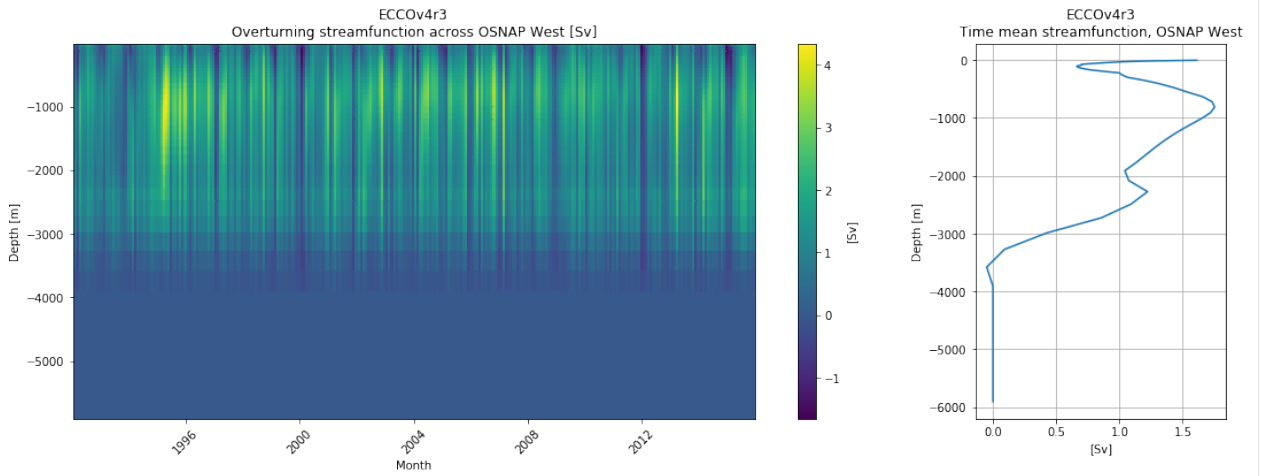
(continues on next page)

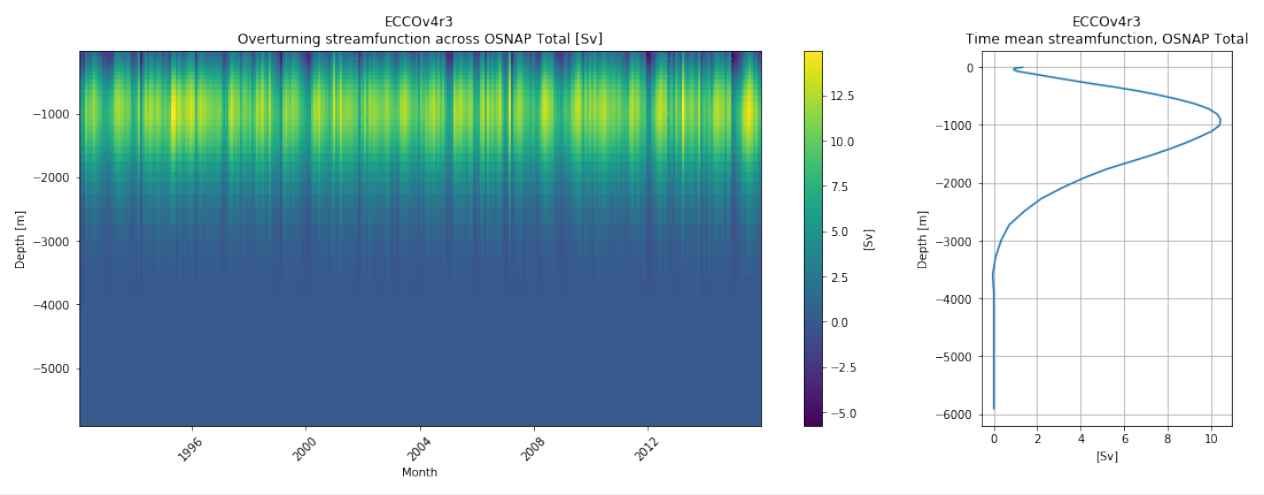
(continued from previous page)

```
    plt.xticks(rotation=45)
    cb = plt.colorbar()
    cb.set_label('[Sv]')

    plt.subplot(1,4,4)
    plt.plot(stf_ds['psi_moc'].mean('time'),stf_ds['Z'])
    plt.title('ECCOv4r3\nTime mean streamfunction, OSNAP %s' % label)
    plt.ylabel('Depth [m]')
    plt.xlabel('[Sv]')
    plt.grid()
    plt.show()
```

[15]:
```
osnap_depth_stf_vs_time(osnap_z_stf_east,'East')
osnap_depth_stf_vs_time(osnap_z_stf_west,'West')
osnap_depth_stf_vs_time(osnap_z_stf_tot,'Total')
```

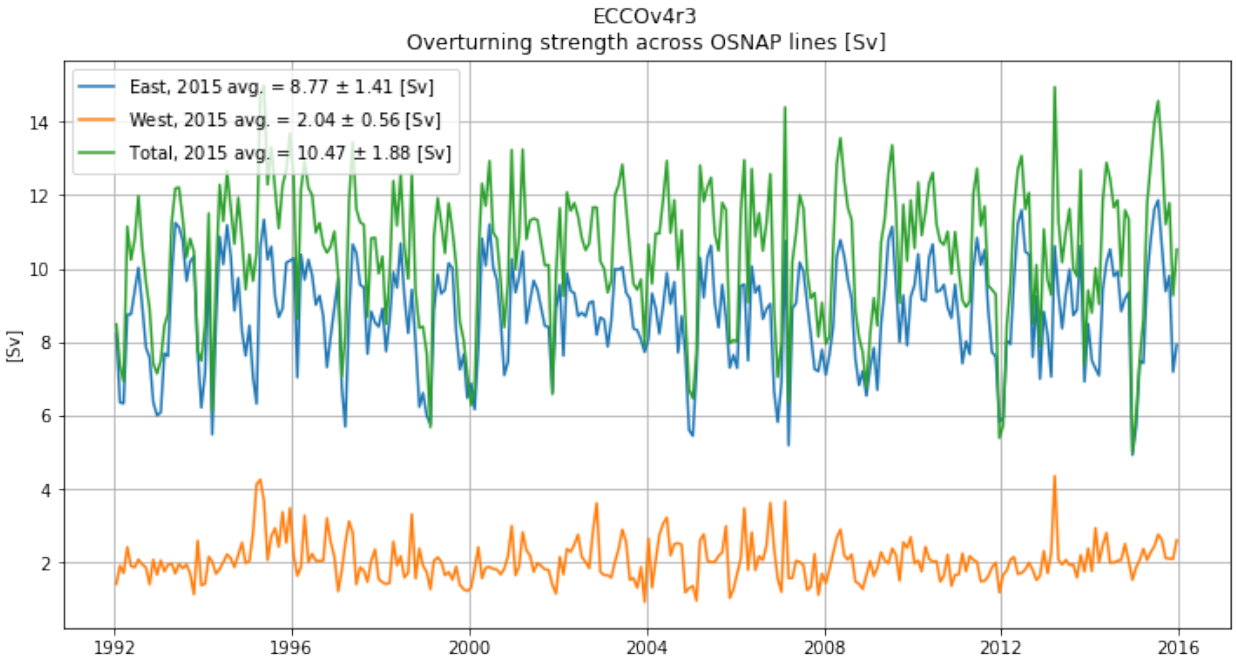






[16]:
```
osnap_z_ov_east = osnap_z_stf_east['psi_moc'].max(dim='k')
osnap_z_ov_west = osnap_z_stf_west['psi_moc'].max(dim='k')
osnap_z_ov_tot = osnap_z_stf_tot['psi_moc'].max(dim='k')
```

[17]:
```
fig = plt.figure(figsize=(12,6))
plt.plot(osnap_z_ov_east['time'],osnap_z_ov_east)
plt.plot(osnap_z_ov_west['time'],osnap_z_ov_west)
plt.plot(osnap_z_ov_tot['time'],osnap_z_ov_tot)
plt.title('ECCOv4r3\nOverturning strength across OSNAP lines [Sv]')
plt.ylabel('[Sv]')
plt.legend((('East, 2015 avg. = %.2f $\pm$ %.2f [Sv]' %
             (osnap_z_ov_east.mean('time').values,osnap_z_ov_east.std('time').values)),
            ('West, 2015 avg. = %.2f $\pm$ %.2f [Sv]' %
             (osnap_z_ov_west.mean('time').values,osnap_z_ov_west.std('time').values)),
            ('Total, 2015 avg. = %.2f $\pm$ %.2f [Sv]' %
             (osnap_z_ov_tot.mean('time').values,osnap_z_ov_tot.std('time').values))))
plt.grid()
plt.show()
```



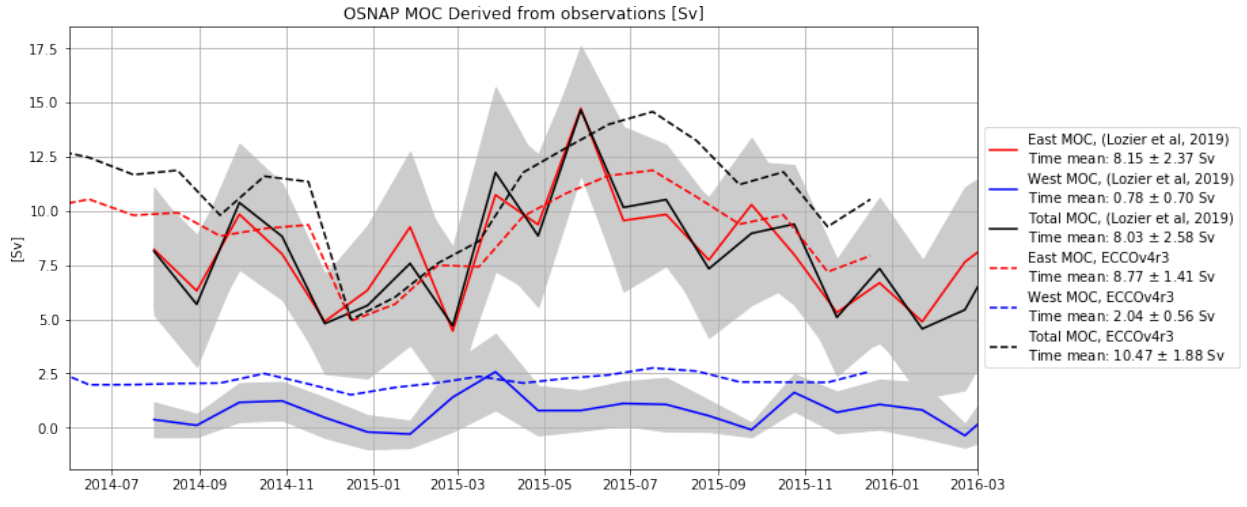
Compare to Observations

[18]:
```
plt.figure(figsize=(12,6))

var_list = ['MOC_EAST_Z','MOC_WEST_Z','MOC_Z']
err_list = ['MOC_EAST_Z_ERR','MOC_WEST_Z_ERR','MOC_Z_ERR']
for var, err in zip(var_list,err_list):
    if 'MOC_Z' in var:
        clr = 'k'
    elif 'MOC_EAST_Z' in var:
        clr = 'r'
    elif 'MOC_WEST_Z' in var:
        clr = 'b'


    plt.plot(obs['TIME'],obs[var], color=clr)

    plt.fill_between(obs['TIME'].values,
                     obs[var]-obs[err],
                     obs[var]+obs[err],
                     color=[0.8,0.8,0.8])

# plot ECCO v4r3 equivalent
plt.plot(osnap_z_ov_east['time'],osnap_z_ov_east, 'r--')
plt.plot(osnap_z_ov_west['time'],osnap_z_ov_west, 'b--',)
plt.plot(osnap_z_ov_tot['time'],osnap_z_ov_tot, 'k--')
plt.xlim(('2014-06','2016-03'))

plt.ylabel('[%s]' % obs['MOC_Z'].units)
```

(continues on next page)

(continued from previous page)

```
plt.grid()
plt.title('OSNAP MOC Derived from observations [%s]' % obs['MOC_Z'].units)
plt.legend((
    ('East MOC, (Lozier et al, 2019)\nTime mean: %.2f $\pm$ %.2f %s' %
            (obs['MOC_EAST_Z'].mean('TIME'),obs['MOC_EAST_Z'].std('TIME'),obs['MOC_EAST_Z
↪'].units)),
    ('West MOC, (Lozier et al, 2019)\nTime mean: %.2f $\pm$ %.2f %s' %
            (obs['MOC_WEST_Z'].mean('TIME'),obs['MOC_WEST_Z'].std('TIME'),obs['MOC_WEST_Z
↪'].units)),
    ('Total MOC, (Lozier et al, 2019)\nTime mean: %.2f $\pm$ %.2f %s' %
            (obs['MOC_Z'].mean('TIME'),obs['MOC_Z'].std('TIME'),obs['MOC_Z'].units)),
    ('East MOC, ECCOv4r3\nTime mean: %.2f $\pm$ %.2f %s' %
            (osnap_z_ov_east.mean('time'),osnap_z_ov_east.std('time'), \
             osnap_z_stf_east['psi_moc'].attrs['units'])),
    ('West MOC, ECCOv4r3\nTime mean: %.2f $\pm$ %.2f %s' %
            (osnap_z_ov_west.mean('time'),osnap_z_ov_west.std('time'), \
             osnap_z_stf_west['psi_moc'].attrs['units'])),
    ('Total MOC, ECCOv4r3\nTime mean: %.2f $\pm$ %.2f %s' %
            (osnap_z_ov_tot.mean('time'),osnap_z_ov_tot.std('time'), \
             osnap_z_stf_tot['psi_moc'].attrs['units']))),
    loc='center left', bbox_to_anchor=(1, 0.5))

plt.show()
```



1.20 ECCOv4 Global Volume Budget Closure

Here we demonstrate the closure of volume budgets in ECCOv4 configurations. This notebook is draws heavily from `evaluating_budgets_in_eccov4r3.pdf` which explains the procedure with Matlab code examples by Christopher G. Piecuch. See ECCO Version 4 release documents: */doc/evaluating_budgets_in_eccov4r3.pdf*).

1.20.1 Objectives

Illustrate how volume budgets are closed globally.

1.20.2 Introduction

ECCOv4 uses the z^* coordinate system in which the depth of the vertical coordinate, z^* varies with time as:

$$z^* = \frac{z - \eta(x,y,t)}{H(x,y) + \eta(x,y,t)} H(x,y) \tag{1.1}$$

With H being the model depth, η being the model sea level anomaly, and z being depth.

If the vertical coordinate didn't change through time then volume fluxes across the 'u' and 'v' grid cell faces of a tracer cell could be calculated by multiplying the velocities at the face with the face area:

volume flux across 'u' face in the +x direction = $UVEL(x,y,k) \times drF(k) \times dyG(x,y) \times hFacW(x,y,k)$

volume flux across 'v' face in the +y direction = $VVEL(x,y,k) \times drF(k) \times dxG(x,y) \times hFacS(x,y,k)$

With `dyG` and `dxG` being the lengths of the 'u' and 'v' faces, `drF` being the grid cell height and `hFacW` and `hFacS` being the vertical fractions of the 'u' and 'v' grid cell faces that are open water (ECCOv4 uses partial cells to better represent bathymetry which can allows 0 < hfac $\leq$ 1).

However, because the vertical coordiate varies with time in the z^* system, the grid cell height `drF` varies with time as `drF`$\times s^*(t)$, with

$$s^*(x,y,k,t) = 1 + \frac{\eta(x,y,t)}{H} \tag{1.2}$$

with $s^* > 1$ when $\eta > 0$

Thus, to calculate the volume fluxes grid cell through horizontal faces we must account for the time-varying grid cell face areas:

volume flux across 'u' in the +x direction face with z^* coordinates = $UVEL(x,y,k) \times drF(k) \times dyG(x,y) \times hFacW(x,y,k) \times s^*(x,y,k,t)$

volume flux across 'v' in the +y direction face with z^* coordinates = $VVEL(x,y,k) \times drF(k) \times dxG(x,y) \times hFacS(x,y,k) \times s^*(x,y,k,t)$

To make budget calculations easier we provide the scaled velocities quantities `UVELMASS` and `VVELMASS`,

$$UVELMASS(x,y,k) = UVEL(x,y,k) \times hFacW(x,y,k) \times s^*(x,y,k,t) \tag{1.3}$$

and

$$VVELMASS(x,y,k) = VVEL(x,y,k) \times hFacS(x,y,k) \times s^*(x,y,k,t) \tag{1.4}$$

It is worth noting that the word **mass** in UVELMASS and VVELMASS is confusing since there is no mass involved here. Think of these terms as simply being UVEL and VVEL multiplied by the fraction of the grid cell height that is open grid cell face across which the volume transport occurs. Partial cell bathymetry can make this fraction (hFacW, hFacS) less than one, and the s^* scaling factor further adjusts this fraction higher or lower through time.

Fully closing the budget requires the vertical volume fluxes across the top and bottom 'w' faces of the grid cell and surface freshwater fluxes. Regarding vertical volume fluxes, there are no s^* or hFac equivalent scaling factors that modify our top and bottom grid cell areas. Therefore, vertical volume fluxes through 'w' faces are simply:

volume flux across 'w' face in the +z direction = $WVEL(x, y, k) \times rA(x, y)$

> **Note:** Inexplicably, the term WVEL is provided with the silly name WVELMASS. Sometimes it's difficult to ignore other people's poor life choices, but please try to do so here. Ignore the confusing name, WVELMASS is identical to WVEL.

In the z^* coordinate system the depth of the surface grid cell is always $z^* = 0$. In the MITgcm, WELMASS at the top of the surface grid cell is the liquid volume flux out of the ocean surface and is proportional to the vertical ocean mass flux, oceFWflx

1.20.3 ETAN in a Boussinesq Model

ECCOv4 uses a volume-conserving Boussinesq formulation of the MITgcm. Because volume is conserved in Boussinesq formulations, seawater density changes do not change model sea level anomaly, ETAN. The following demonstration of ETAN budget closure considers volumetric fluxes but does not take into consideration expansion/contraction due to changes in density. Furthermore, model ETAN changes with the exchange of water between ocean and sea-ice while in reality the Archimedes principle holds that sea level should not change following the growth or melting of sea-ice because floating sea-ice displaces a volume of seawater equal to its weight. Thus, ETAN is not comparable to observed sea level.

We correct ETAN to make a sea surface height field that is comaprable with observations by making three corrections: with a) the "Greatbatch correction", a time varying, globally-uniform correction to ocean volume due to changes in global mean density, b) the inverted barometer (IB) correction (see SSHIBC) and c) the 'sea ice load' correction to account for the displacement of seawater due to submerged sea-ice and snow (see sIceLoad). A demonstration of these corrections is outside the scope of this tutorial. Here we focus on closing the model budget keeping in mind that we are neglecting sea level changes from changes in global mean density and the fact that ETAN does not account for volume displacement due to submerged sea-ice.

Greatbatch, 1994. J. of Geophys. Res. Oceans, https://doi.org/10.1029/94JC00847

1.20.4 Evaluating the model sea level anomaly ETAN volume budget

We will evalute

$$\underbrace{\frac{\partial \eta}{\partial t}}_{G_{\text{total tendency}}} = \underbrace{\int_{-H}^{0} \left(-\nabla_{z^*}(s^* \mathbf{v} - \frac{\partial w}{\partial z^*} \right) dz^*}_{G_{\text{volumetric divergence}}} + \underbrace{F}_{G_{surfacefluxes}} \tag{1.5}$$

The total tendency of η, $G_{\text{total tendency}}$ is the sum of the η tendencies from volumetric divergence, $G_{\text{volumetric divergence}}$, and volumetric surface fluxes, $G_{\text{surface fluxes}}$.

In discrete form, using indexes that start from k=0 (surface tracer cell) and running to k=nk-1 (bottom tracer cell)

$$\frac{\eta(i,j)}{\partial t} = \sum_{k=nk-1}^{0} \underbrace{[UVELMASS(i_g,j,k) - UVELMASS(i_g+1,j,k)]\ dyG(i_g,j)\ drF(k)}_{\text{volumetric flux in minus out in x direction}} + \tag{1.6}$$

$$\sum_{k=nk-1}^{0} \underbrace{[VVELMASS(i,j_g,k) - VVELMASS(i,j_g+1,k)]\ dxG(i,j_g)\ drF(k)}_{\text{volumetric flux in minus out in y direction}} + \tag{1.7}$$

$$\sum_{k_l=nk}^{1} \underbrace{WVELMASS(i,j,k_l)\ drA(i,j)}_{\text{volumetric flux through grid cell bottom surface}} + \tag{1.8}$$

$$\underbrace{oceFWflx(i,j)/rhoConst}_{\text{volumetric flux through the top surface of the uppermost tracer cell}} \tag{1.9}$$

In the above we intentionally sum `WVELMASS` fluxes from the BOTTOM surface of the lowermost grid cell (at $k_l = 50$) to the BOTTOM face of the uppermost grid cell ($k_l = 1$) so that we can explicitly include the surface volume flux (forcing) term, $oceFWflx(i,j)/rhoConst$.

We will calculate $\partial\eta/\partial t$ by differencing instantaneous monthly snapshots of η as

$$\frac{\partial \eta}{\partial t} = \frac{\eta(i,j,t+1) - \eta(i,j,t)}{\Delta t}$$

The `UVELMASS, VVELMASS, WVELMASS` and `oceFWflx` terms must be time-average quantities between the monthly η snapshots.

Prepare environment and loading the relevant model variables

[1]:
```
import numpy as np
import sys
import xarray as xr
from copy import deepcopy
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
import cmocean
warnings.filterwarnings('ignore')
```

[62]:
```
# Density kg/m^3
rhoconst = 1029

## needed to convert surface mass fluxes to volume fluxes

# lat/lon resolution in degrees to interpolate the model
# fields for the purposes of plotting
map_dx = .2
map_dy = .2
```

[3]:
```
## Import the ecco_v4_py library into Python
## =========================================

## -- If ecco_v4_py is not installed in your local Python library,
```

(continues on next page)

(continued from previous page)

```
##    tell Python where to find it.  For example, if your ecco_v4_py
##    files are in /Users/ifenty/ECCOv4-py/ecco_v4_py, then use:

sys.path.append('/home/ifenty/ECCOv4-py')
import ecco_v4_py as ecco
```

Load ecco_grid

[4]:
```
## Set top-level file directory for the ECCO NetCDF files
## ================================================================
# base_dir = '/home/username/'
base_dir = '/home/ifenty/ECCOv4-release'

ecco_version = 'v4r3'

## define a high-level directory for ECCO fields
ECCO_dir = base_dir + '/Release3_alt'
```

[5]:
```
## Load the model" grid
grid_dir= ECCO_dir + '/nctiles_grid/'
ecco_grid = ecco.load_ecco_grid_nc(grid_dir, 'ECCOv4r3_grid.nc')
```

Load 2D MONTHLY η snapshots

[6]:
```
data_dir= ECCO_dir + '/nctiles_monthly_snapshots'

year_start = 1993
year_end = 2017


# load one extra year worth of snapshots
ecco_monthly_snaps = ecco.recursive_load_ecco_var_from_years_nc(data_dir, \
                         vars_to_load=['ETAN'],\
                         years_to_load=range(year_start, year_end+1)).load()

num_months = len(ecco_monthly_snaps.time.values)
# drop the last 11 months so that we have one snapshot at the beginning and end of each␣
↪month within the
# range t1993/1/1 to 2015/1/1)

ecco_monthly_snaps = ecco_monthly_snaps.isel(time=np.arange(0, num_months-11))
```

```
loading files of  ETAN
```

[7]:
```
# 1993-01 (beginning of first month) to 2015-01-01 (end of last month, 2014-12)
print(ecco_monthly_snaps.ETAN.time.isel(time=[0, -1]).values)
```

```
['1993-01-01T00:00:00.000000000' '2015-01-01T00:00:00.000000000']
```

[8]:
```
# find the record of the last ETAN snapshot
last_record_date =
    ecco.extract_yyyy_mm_dd_hh_mm_ss_from_datetime64(ecco_monthly_snaps.time[-1].values)
print(last_record_date)
last_record_year = last_record_date[0]
```

```
(2015, 1, 1, 0, 0, 0)
```

Load MONTHLY mean data

[9]:
```
data_dir= ECCO_dir + '/nctiles_monthly'

year_end = last_record_year
ecco_monthly_mean = ecco.recursive_load_ecco_var_from_years_nc(data_dir, \
                        vars_to_load=['oceFWflx',
                                      'UVELMASS',
                                      'VVELMASS',
                                      'WVELMASS'],\
                        years_to_load=range(year_start, year_end)).load()
```

```
loading files of  UVELMASS
loading files of  VVELMASS
loading files of  WVELMASS
loading files of  oceFWflx
```

[10]:
```
# first and last monthly-mean records
print(ecco_monthly_mean.time.isel(time=[0, -1]).values)
```

```
['1993-01-16T12:00:00.000000000' '2014-12-16T12:00:00.000000000']
```

[11]:
```
# each monthly mean record is bookended by a snapshot.
#we should have one more snapshot than monthly mean record
print('number of monthly mean records: ', len(ecco_monthly_mean.time))
print('number of monthly snapshot records: ', len(ecco_monthly_snaps.time))
```

```
number of monthly mean records:  264
number of monthly snapshot records:  265
```

Create the xgcm 'grid' object

the xgcm grid object makes it easy to make flux divergence calculations across different tiles of the lat-lon-cap grid.

[12]:
```
ecco_xgcm_grid = ecco.get_llc_grid(ecco_grid)
ecco_xgcm_grid
```

[12]:
```
<xgcm.Grid>
Y Axis (not periodic):
  * center   j --> left
  * left     j_g --> center
Z Axis (not periodic):
  * center   k --> left
  * left     k_l --> center
```

(continues on next page)

(continued from previous page)

```
  * outer    k_p1 --> center
  * right    k_u --> center
X Axis (not periodic):
  * center   i --> left
  * left     i_g --> center
```

1.20.5 Calculate LHS: η time tendency: $G_{\text{total tendency}}$

We calculate the monthly-averaged time tendency of ETAN by differencing monthly ETAN snapshots. Subtract the numpy arrays $\eta(t+1)$ - $\eta(t)$. This operation gives us Δ ETAN $/\Delta$ t (month) records.

[13]:
```
num_months = len(ecco_monthly_snaps.time)
G_total_tendency_month = \
    ecco_monthly_snaps.ETAN.isel(time=range(1,num_months)).values - \
    ecco_monthly_snaps.ETAN.isel(time=range(0,num_months-1)).values

# The result is a numpy array of 264 months
print('shape of G_total_tendency_month: ', G_total_tendency_month.shape)
```

```
shape of G_total_tendency_month:  (264, 13, 90, 90)
```

[14]:
```
ecco_monthly_mean.oceFWflx.shape
```

[14]:
```
(264, 13, 90, 90)
```

[15]:
```
# Convert this numpy array to an xarray Dataarray to take advantage
# of xarray time indexing.
# The easiest way is to copy an existing DataArray that has the
# dimensions and time indexes that we want,
# replace its values, and change its name.

tmp = ecco_monthly_mean.oceFWflx.copy(deep=True)
tmp.values = G_total_tendency_month
tmp.name = 'G_total_tendency_month'
G_total_tendency_month = tmp

# the nice thing is that now the time values of G_total_tendency_month now line
# up with the time values of the time-mean fields (middle of the month)
print('\ntime of first array of G_total_tendency_month');
print(G_total_tendency_month.time[0].values)
```

```

time of first array of G_total_tendency_month
1993-01-16T12:00:00.000000000
```

Now convert Δ ETAN $/\Delta$ t (month) to Δ ETAN $/\Delta$ t (seconds) by dividing by the number of seconds in each month. To find the number of seconds in each month, subtract the model time step number (which is hourly) from the beginning and end of each month:

[16]:
```
if ecco_version == 'v4r4':
    hrs_per_month = ecco_monthly_snaps.timestep[1:].values - \
    ecco_monthly_snaps.timestep[0:-1].values
```

(continues on next page)

(continued from previous page)

```
elif ecco_version == 'v4r3':
    hrs_per_month = ecco_monthly_snaps.iter[1:].values - \
    ecco_monthly_snaps.iter[0:-1].values

# convert hours per month to seconds per month:
secs_per_month = hrs_per_month * 3600

# Make a DataArray with the number of seconds in each month,
#time indexed to the times in dETAN_dT_perMonth (middle of each month)
secs_per_month = xr.DataArray(secs_per_month, \
                              coords={'time': G_total_tendency_month.time.values}, \
                              dims='time')

# show number of seconds in the first two months:
print('# of seconds in Jan and Feb 1993 ', secs_per_month[0:2].values)

# sanity check: show number of days in the first two months:
print('# of days in Jan and Feb 1993 ', secs_per_month[0:2].values/3600/24)
```

```
# of seconds in Jan and Feb 1993  [2678400 2419200]
# of days in Jan and Feb 1993  [31. 28.]
```

Convert the `ns_in_month` from timedelta64 object to float so we can use it to use it for a mathematical operation: converting G_total_tendency_month to G_total_tendency. Also, convert from ns to seconds.

[17]:
```
# convert dETAN_dT_perMonth to perSeconds
G_total_tendency = G_total_tendency_month / secs_per_month
```

1.20.6 Plot the time-mean $\partial\eta/\partial t$, total $\Delta\eta$, and one example $\partial\eta/\partial t$ field

Time-mean $\partial\eta/\partial t$

To calculate the time averaged $G_{total_tendency}$ one might be tempted to do the following

```
> G_total_tendency_mean = G_total_tendency.mean('time')
```

But that would be folly because the 'mean' function does not know that the number of days in each month is different! The result would downweight Februarys and upweight Julys. We have to weight the tendency records by the length of each month. A clever way of doing that is provided in the xarray documents: https://xarray-test.readthedocs.io/en/latest/examples/monthly-means.html

In our case we know the length of each month, we just calculated it above in `secs_per_month`. We will weight each month by the relative # of seconds in each month and sum to get a weighted average.

[18]:
```
# the weights are just the # of seconds per month divided by total seconds
month_length_weights = secs_per_month / secs_per_month.sum()
```

The time mean of the `ETAN` tendency, $\overline{G_{\text{total tendency}}}$, is given by

$\overline{G_{\text{total tendency}}} = \sum_{i=1}^{nm} w_i G_{\text{total tendency}}$

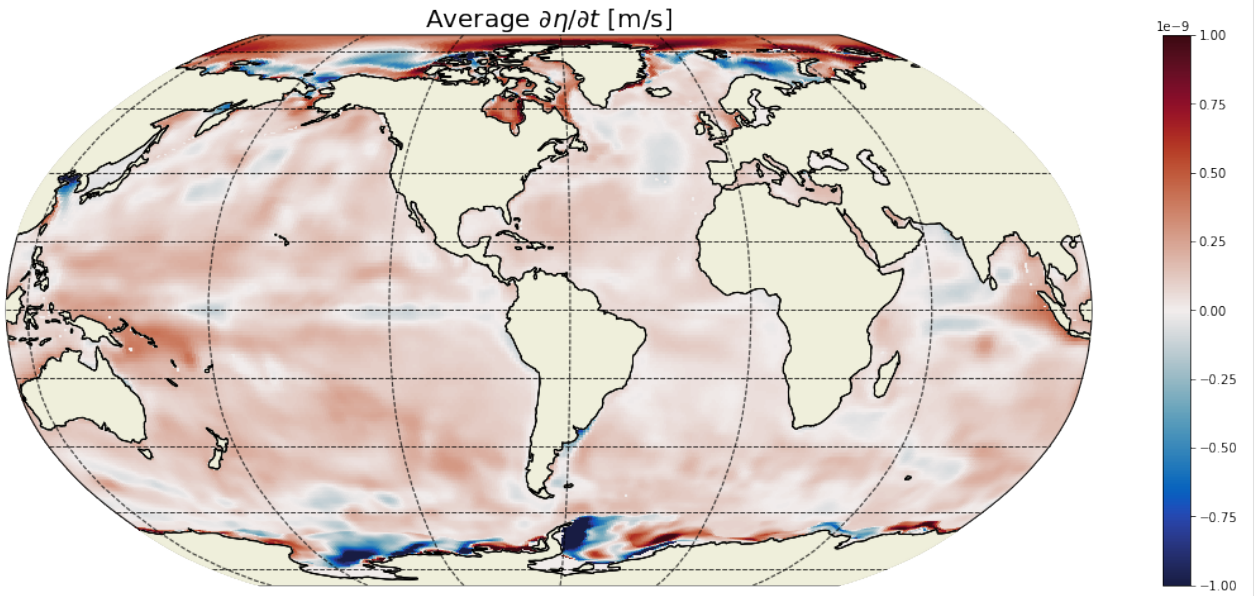
with $\sum_{i=1}^{nm} w_i = 1$ and nm=number of months

[19]:
```
# the weights sum to 1
print(month_length_weights.sum())
```

```
<xarray.DataArray ()>
array(1.)
```

[20]:
```
# the weighted mean weights by the length of each month (in seconds)
G_total_tendency_mean = (G_total_tendency*month_length_weights).sum('time')
```

[21]:
```
plt.figure(figsize=(20,8));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC,\
                              G_total_tendency_mean,show_colorbar=True,\
                              cmin=-1e-9, cmax=1e-9, \
                              cmap=cmocean.cm.balance, user_lon_0=-67,\
                              dx=map_dx,dy=map_dy);
plt.title('Average $\partial \eta / \partial t$ [m/s]', fontsize=20);
```



Total $\Delta\eta$

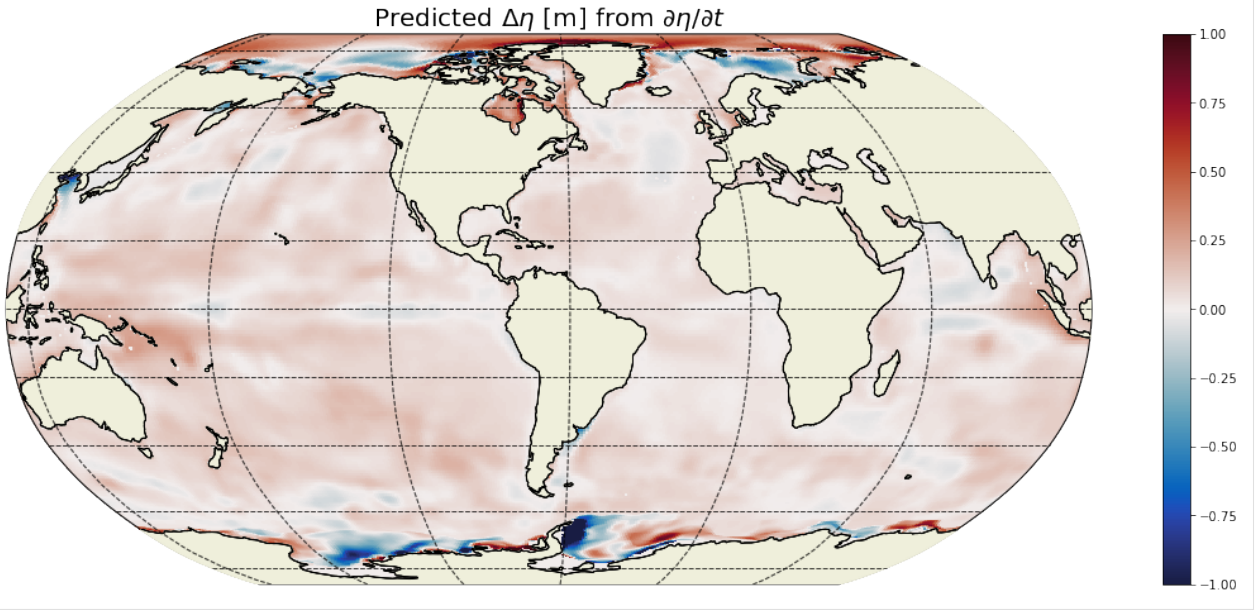
The time average eta tendency is small, about 1 billionth of a meter per second. The ECCO period is coming up to a billion seconds though… How much did `ETAN` change over the analysis period?

[22]:
```
# the number of seconds in the entire period
seconds_in_entire_period = \
    float(ecco_monthly_snaps.time[-1] - ecco_monthly_snaps.time[0])/1e9
print ('seconds in analysis period: ', seconds_in_entire_period)

# which is also the sum of the number of seconds in each month
print('sum of seconds in each month ', secs_per_month.sum().values)
```

```
seconds in analysis period:  694224000.0
sum of seconds in each month  694224000
```

[23]:
```
ETAN_delta = G_total_tendency_mean*seconds_in_entire_period
```

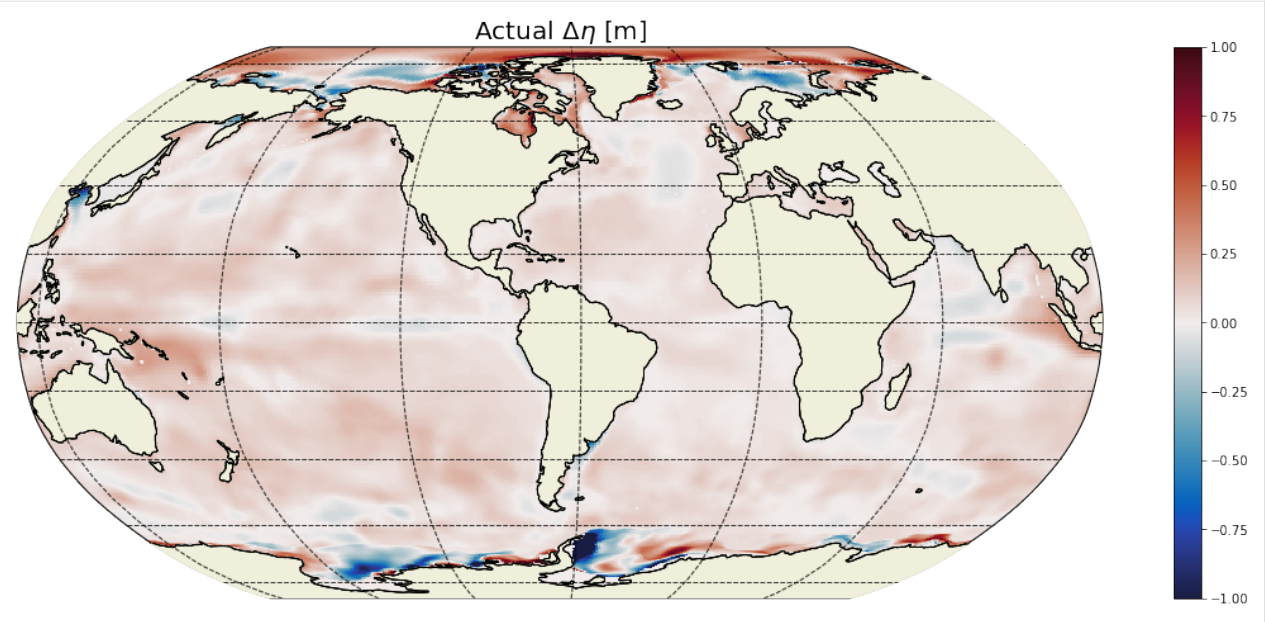
[24]:
```
plt.figure(figsize=(20,8));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, \
                              ETAN_delta,show_colorbar=True,\
                              cmin=-1, cmax=1, \
                              cmap=cmocean.cm.balance, user_lon_0=-67, \
                              dx=map_dx,dy=map_dy);
plt.title('Predicted $\Delta \eta$ [m] from $\partial \eta / \partial t$', \
          fontsize=20);
```



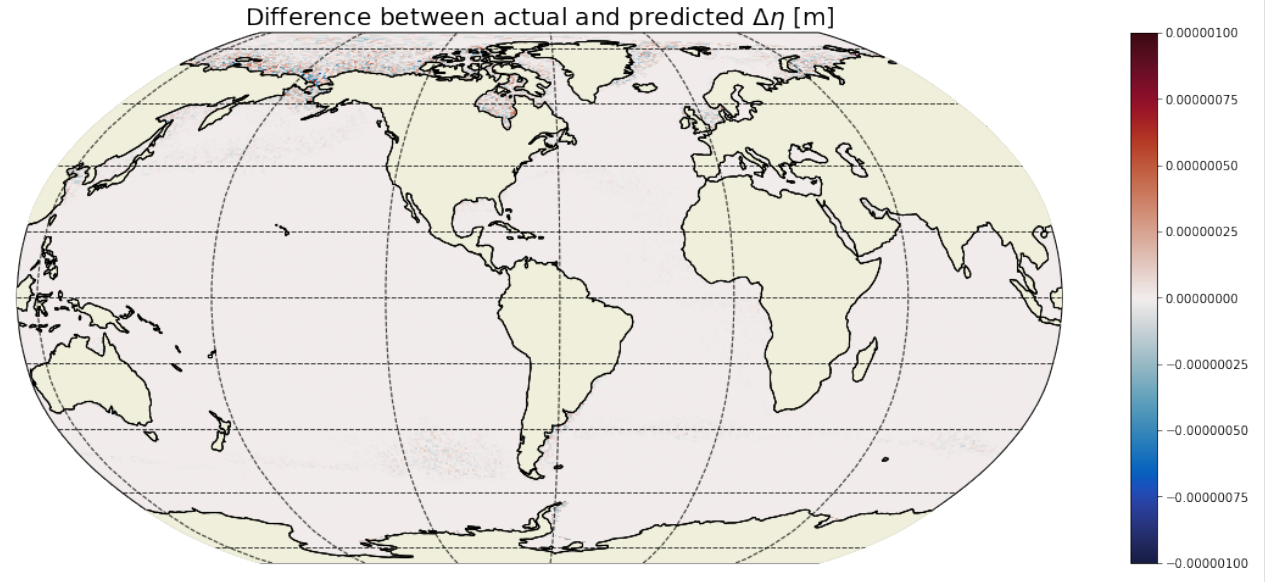
We can sanity check the total ETAN change that we found by multipling the time-mean ETAN tendency with the number of seconds in the simulation by comparing that with the difference in ETAN between the end of the last month and start of the first month.

[25]:
```
ETAN_delta_method_2 = ecco_monthly_snaps.ETAN.isel(time=-1).values - \
    ecco_monthly_snaps.ETAN.isel(time=0).values
```

[26]:
```
plt.figure(figsize=(20,8));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, \
                              ETAN_delta_method_2,
                              show_colorbar=True,\
                              cmin=-1, cmax=1, \
                              cmap=cmocean.cm.balance, user_lon_0=-67,\
                              dx=map_dx,dy=map_dy);

plt.title('Actual $\Delta \eta$ [m]', fontsize=20);
```



[27]:
```
plt.figure(figsize=(20,8));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, \
                              ETAN_delta_method_2-ETAN_delta,\
                              show_colorbar=True,\
                              cmin=-1e-6, cmax=1e-6, \
                              cmap=cmocean.cm.balance, user_lon_0=-67,\
                              dx=map_dx,dy=map_dy);
plt.title('Difference between actual and predicted $\Delta \eta$ [m]', \
          fontsize=20);
```

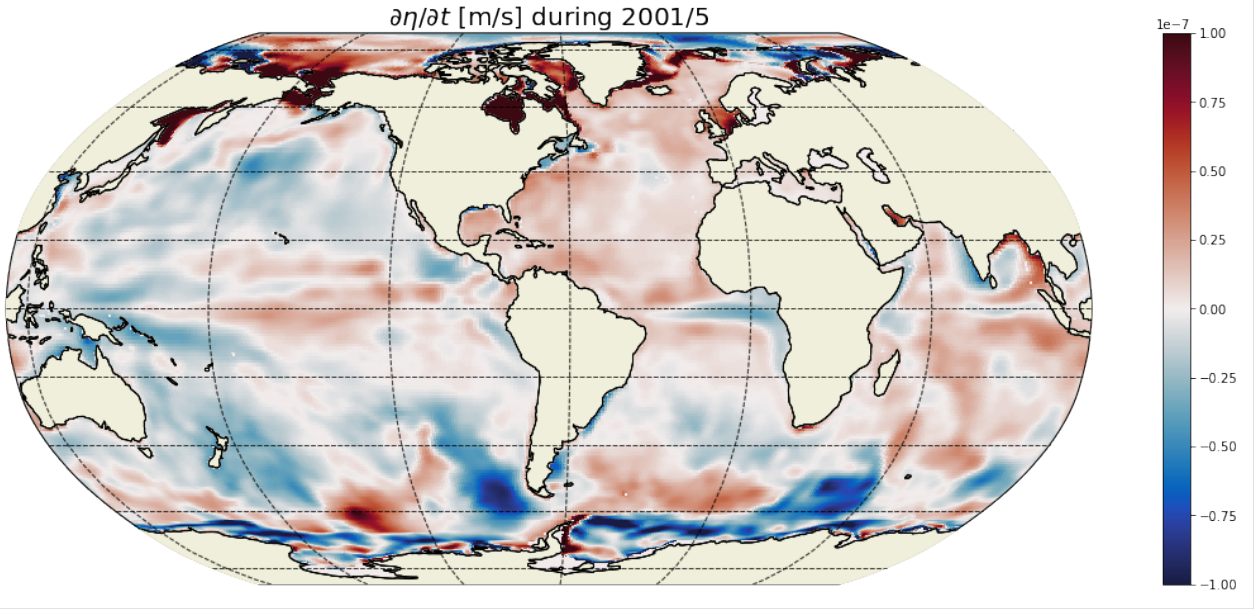


That's a big woo, these are the same to within 10^-6 meters!

Example $\partial\eta/\partial t$ field

[28]:
```
plt.figure(figsize=(20,8));

# get an array of YYYY, MM, DD, HH, MM, SS for
#dETAN_dT_perSec at time index 100
tmp = ecco.extract_yyyy_mm_dd_hh_mm_ss_from_datetime64(G_total_tendency.time[100].values)
print(tmp)
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, \
                              G_total_tendency.isel(time=100),\
                              show_colorbar=True,\
                              cmin=-1e-7, cmax=1e-7,\
                              cmap=cmocean.cm.balance, user_lon_0=-67,\
                              dx=map_dx,dy=map_dy);

plt.title('$\partial \eta / \partial t$ [m/s] during ' +
          str(tmp[0]) +'/' + str(tmp[1]), fontsize=20);
```

```
(2001, 5, 16, 12, 0, 0)
```



For any given month the time rate of change of `ETAN` is two orders of magnitude smaller than the 1993-2015 mean. In the above we are looking at May 2001. We see positive `ETAN` tendency due sea ice melting in the northern hemisphere (e.g., Baffin Bay, Greenland Sea, and Chukchi Sea).

1.20.7 Calculate RHS: η tendency due to surface fluxes, $G_{\text{surface fluxes}}$

Surface mass fluxes are given in `oceFWflx`. Convert surface mass flux to a vertical velocity by dividing by the reference density `rhoConst`= 1029 kg m-3

[29]:
```
# tendency of eta implied by surface volume fluxes (m/s)
G_surf_fluxes = ecco_monthly_mean.oceFWflx/rhoconst
```

1.20.8 Plot the time-mean, total, and one month average of $G_{\text{surface fluxes}}$

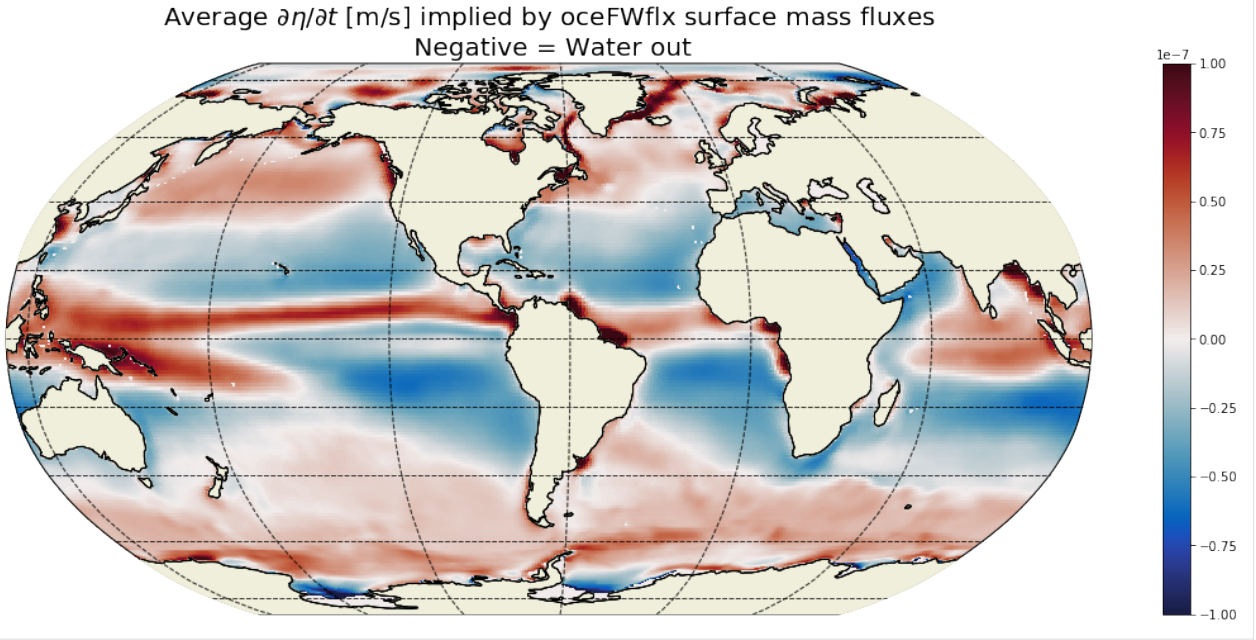
Time-mean $G_{\text{surface fluxes}}$

We calculate the time-mean surface flux η tendency using the same weights as the total η tendency.

[30]:
```
G_surf_fluxes_mean= (G_surf_fluxes*month_length_weights).sum('time')
G_surf_fluxes_mean.shape
```

[30]:
```
(13, 90, 90)
```

[31]:
```
plt.figure(figsize=(20,8));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, \
                              G_surf_fluxes_mean,
                              show_colorbar=True,\
                              cmin=-1e-7, cmax=1e-7, \
                              cmap=cmocean.cm.balance, user_lon_0=-67,
                              dx=map_dx,dy=map_dy)
plt.title('Average $\partial \eta / \partial t$ [m/s] implied by oceFWflx surface mass␣
↪fluxes\n Negative = Water out',
          fontsize=20);
```



Total $\Delta\eta$ due to surface fluxes

If there were no other terms on the RHS to balance surface fluxes, the total change in `ETAN` between 1993 and 2015 would be order of h10s of meters almost everwhere.

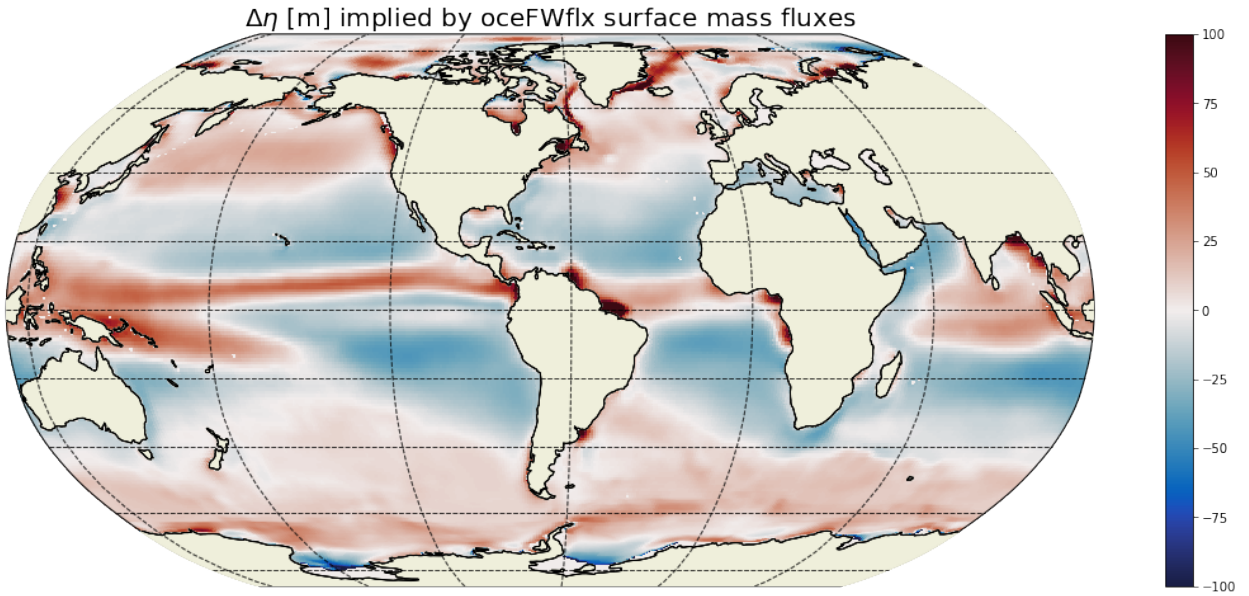
[32]:
```
ETAN_delta_surf_fluxes = G_surf_fluxes_mean*seconds_in_entire_period
```

[33]:
```
plt.figure(figsize=(20,8));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, \
                              ETAN_delta_surf_fluxes,show_colorbar=True,\
```

(continues on next page)

(continued from previous page)

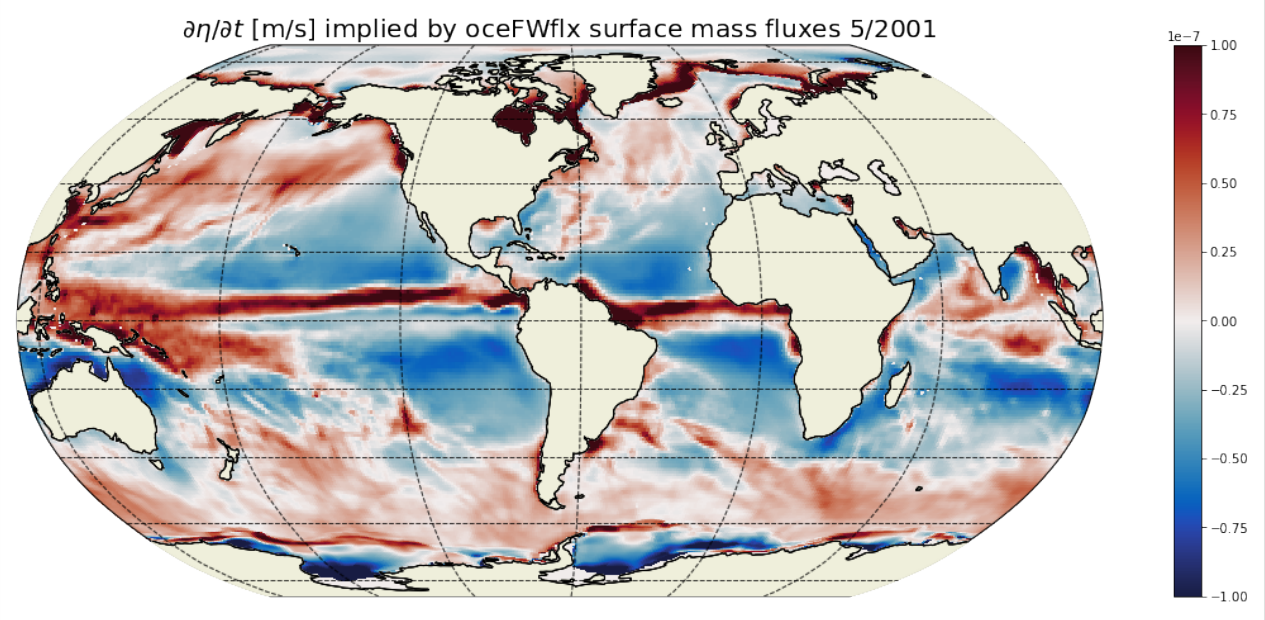
```
                                cmin=-100, cmax=100, \
                                cmap=cmocean.cm.balance, user_lon_0=-67, \
                                dx=map_dx,dy=map_dy);
plt.title('$\Delta \eta$ [m] implied by oceFWflx surface mass fluxes',
          fontsize=20);
```



Example $\partial\eta/\partial t$ impliedy by surface fluxes

[34]:
```
plt.figure(figsize=(20,8));

# get an array of YYYY, MM, DD, HH, MM, SS for
# dETAN_dT_perSec at time index 100
tmp = ecco.extract_yyyy_mm_dd_hh_mm_ss_from_datetime64(G_surf_fluxes.time[100].values)
print(tmp)
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, \
                              G_surf_fluxes.isel(time=100),show_colorbar=True,\
                             cmin=-1e-7, cmax=1e-7, \
                              cmap=cmocean.cm.balance, user_lon_0=-67,\
                              dx=map_dx,dy=map_dy);
plt.title('$\partial \eta / \partial t$ [m/s] implied by oceFWflx surface mass fluxes ' +
          str(tmp[1]) +'/' + str(tmp[0]), fontsize=20);
```

```
(2001, 5, 16, 12, 0, 0)
```



For any given month the time rate of change of ETAN is almost the same as its 22 year mean. Differences are largest in the high latitudes where sea-ice melt and growth during any particular month induce large changes in ETAN.

1.20.9 Calculate RHS: η tendency due to volumetric flux divergence, $G_{\text{volumetric fluxes}}$

First we will look at vertical volumetric flux divergence, then horizontal volumetric flux divergence.

Vertical volumetric flux divergence

It turns out that WVELMASS at k_l=0 (the top face of the top tracer cell) is proportional to the ocean surface mass flux oceFWflx. The vertical velocity of the ocean surface is equal to the rate at which water is being added or removed across the top surface of the uppermost grid cell. This is demonstrated by differencing the velocity at the top 'w' face of the uppermost tracr cell WVELMASS (k_l = 0) and the velocity equivalent of transporting the surface mass flux term oceFWflx through this same face.

First, find the time-mean vertical velocity at the liquid ocean surface

[35]:
```
WVELMASS_surf_mean = \
    (ecco_monthly_mean.WVELMASS.isel(k_l=0)*month_length_weights).sum('time')
```

Next, find the time-mean vertical velocity implied by the oceFWflx at k_l=0:

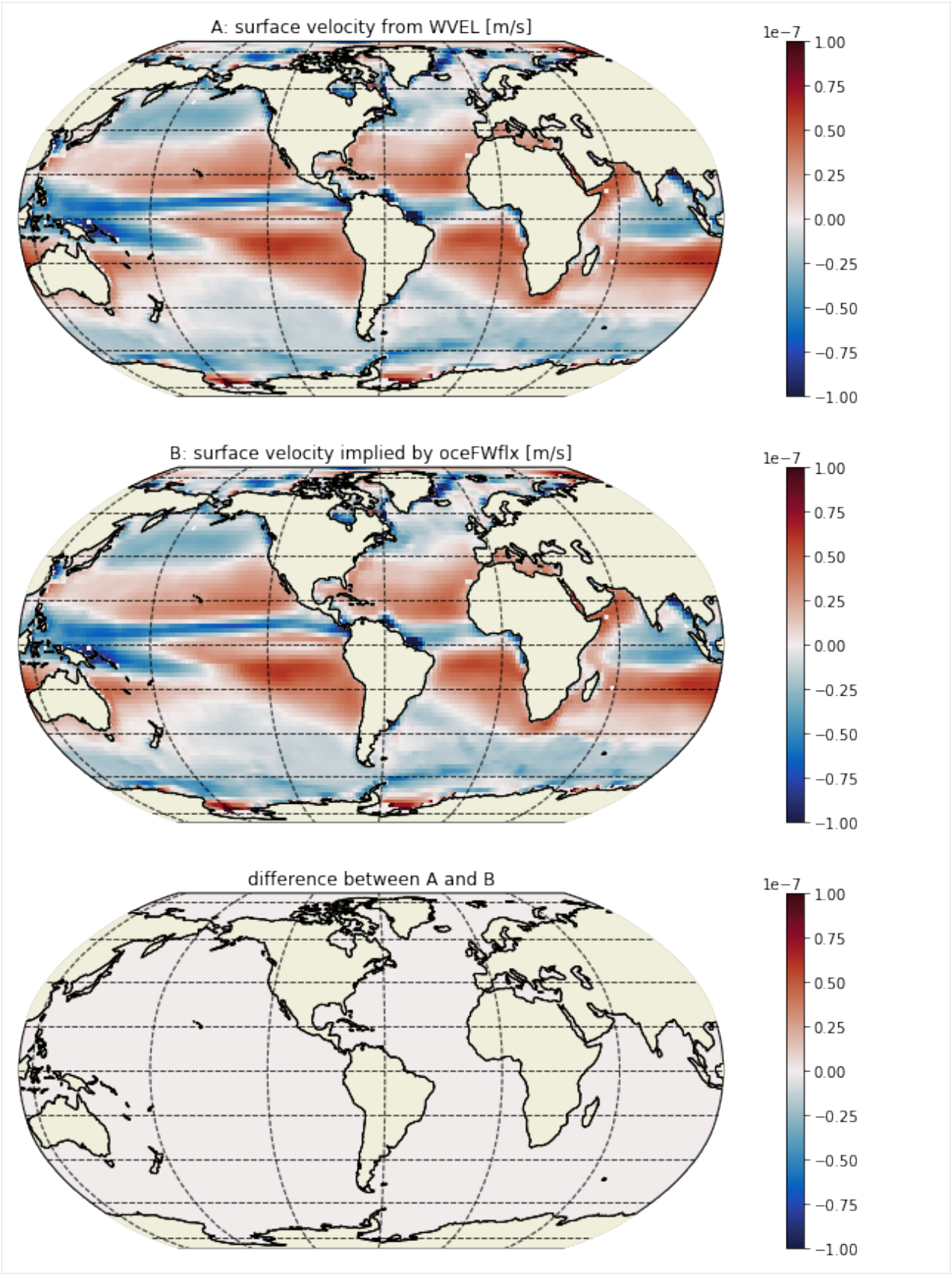
[36]:
```
WVEL_from_oceFWflx_mean = \
    -(ecco_monthly_mean.oceFWflx*month_length_weights).sum('time')/rhoconst
```

[37]:
```
plt.figure(figsize=(15,15))
#plt.sca(axs[0,0])
F=ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, \
                                WVELMASS_surf_mean,\
                                show_colorbar=True,\
                                cmin=-1e-7, cmax=1e-7, \
                                cmap=cmocean.cm.balance, user_lon_0=-67,\
```

(continues on next page)

(continued from previous page)

```
                                dx=2,dy=2, subplot_grid=[3,1,1]);
F[1].set_title('A: surface velocity from WVEL [m/s]')

F=ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, \
                                WVEL_from_oceFWflx_mean,\
                                show_colorbar=True,\
                                cmin=-1e-7, cmax=1e-7, \
                                cmap=cmocean.cm.balance, user_lon_0=-67,\
                                dx=2,dy=2, subplot_grid=[3,1,2])
F[1].set_title("B: surface velocity implied by oceFWflx [m/s]")

F=ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, \
                                WVELMASS_surf_mean-WVEL_from_oceFWflx_mean,\
                                show_colorbar=True,\
                                cmin=-1e-7, cmax=1e-7, \
                                cmap=cmocean.cm.balance, user_lon_0=-67,\
                                dx=2,dy=2, subplot_grid=[3,1,3])
F[1].set_title("difference between A and B");
```


A: surface velocity from WVEL [m/s]
1e−7
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
B: surface velocity implied by oceFWflx [m/s]
1e−7
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
difference between A and B
1e−7
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

WVELMASS at the surface evidently is the same as the surface velocity implied by the surface mass flux oceFWflx. Thus, we do not actually need oceFWflx to close the volume budget. However, to keep the surface forcing term explicitly represented, we will keep oceFWflx and instead zero out the values of WVELMASS at the surface so as to avoid double counting.

Calculate the vertical volume fluxes at each level: (velocity x grid cell area) [m3 s-1]

[38]:
```
vol_transport_z = ecco_monthly_mean.WVELMASS * ecco_grid.rA
```

Set the volume transport at the surface level to be zero because we already counted the fluxes out of the domain with oceFWflx.

[39]:
```
vol_transport_z.isel(k_l=0).values[:] = 0
```

Each grid cell has a top and bottom surface and therefore WEVELMASSshould have 51 vertical levels (one more than the number of tracer cells). For some reason we only have 50 vertical levels, with the bottom of the 50th tracer cell missing. To calculate vertical flux divergence we need to add this 51st WVELMASS which is everywhere zero (no volume flux from the seafloor). The xgcm library handles this in its diff routine by specifying the boundary='fill' and fill_value = 0.

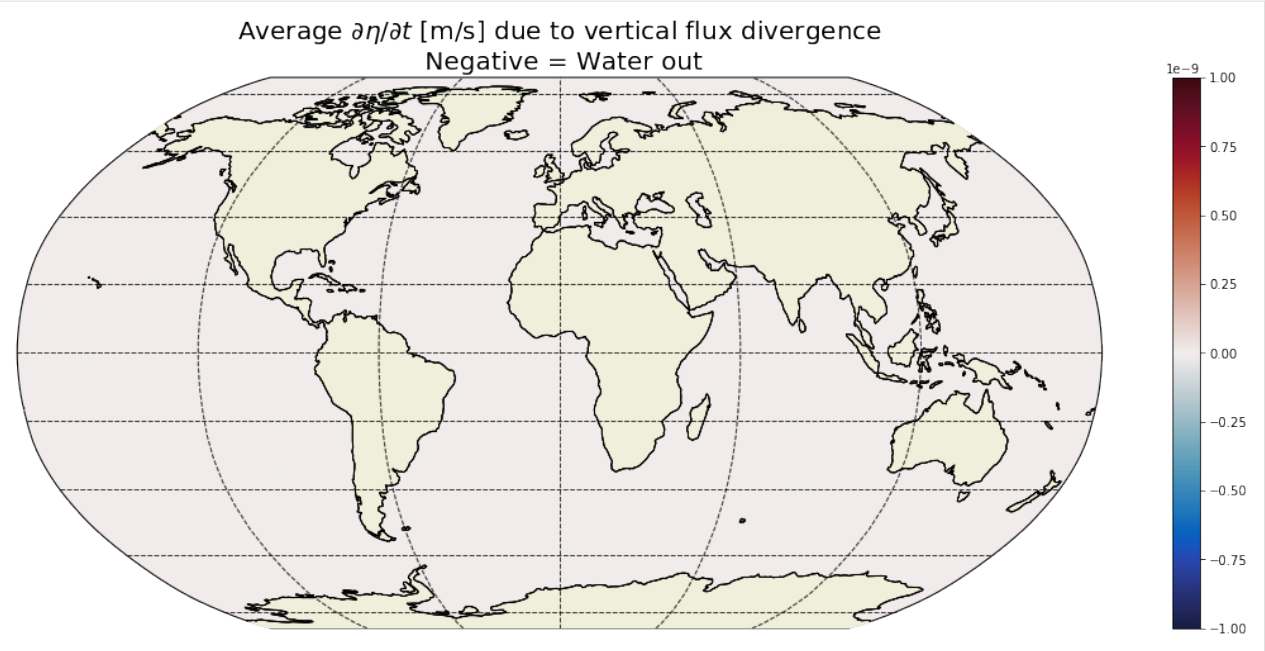
[40]:
```
# volume flux divergence into each grid cell, m^3 / s
vol_vert_divergence = ecco_xgcm_grid.diff(vol_transport_z, 'Z', \
                                          boundary='fill', fill_value=0)

# change in eta per unit time due to volumetric vertical convergence
# at each depth level: m/s
G_vertical_flux_divergence = vol_vert_divergence / ecco_grid.rA;
```

[41]:
```
# change in eta per unit time due to vertical integral of
# volumetric horizonal convergence: m/s
G_vertical_flux_divergence_depth_integrated = G_vertical_flux_divergence.sum('k')
```

[42]:
```
# Calculate the time-mean surface flux $\eta$ tendency using
# the same weights as the total $\eta$ tendency.
G_vertical_flux_divergence_depth_integrated_time_mean = \
    (G_vertical_flux_divergence_depth_integrated * month_length_weights).sum('time')
```

Plot the time-mean $G_{\text{vertical flux divergence}}$

[43]:
```
plt.figure(figsize=(20,8));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC,\
                              G_vertical_flux_divergence_depth_integrated_time_mean,
                              show_colorbar=True,\
                              cmap=cmocean.cm.balance, \
                              cmin=-1e-9, cmax=1e-9,
                              dx=map_dx,dy=map_dy);
plt.title('Average $\partial \eta / \partial t$ [m/s] due to vertical flux divergence\n␣
↪Negative = Water out',
          fontsize=20);
```



These values are everywhere essentially zero (numerical noise). On average, the only vertical flux divergence in the column is across the ocean surface. Below the surface, the sum of vertical flux divergence in all tracer cells in the column must be zero because any divergence in any one particular cell is exactly offset by convergence in another cell. Net convergence into the column manifests as a positive vertical velocity at the surface which is equal to oceFWflux in the time-mean. **Thanks to Hong Zhang for comments that improved this explanation**

Horizontal Volume Flux Divergence

[44]:
```
# Volumetric transports in x and y(m^3/s)
vol_transport_x = ecco_monthly_mean.UVELMASS * ecco_grid.dyG * ecco_grid.drF
vol_transport_y = ecco_monthly_mean.VVELMASS * ecco_grid.dxG * ecco_grid.drF
```

[45]:
```
# Difference of horizontal transports in x and y directions
vol_flux_diff = ecco_xgcm_grid.diff_2d_vector({'X': vol_transport_x, \
                                               'Y': vol_transport_y},\
                                              boundary='fill')

# volume flux divergence into each grid cell, m^3 / s
vol_horiz_divergence = (vol_flux_diff['X'] + vol_flux_diff['Y'])

# change in eta per unit time due to volumetric horizonal
# convergence at each depth level: m/s
# a positive DIVERGENCE leads to negative eta tendency
G_vol_horiz_divergence = -vol_horiz_divergence / ecco_grid.rA

# change in eta in each grid cell per unit time due to horiz. divergence: m/s
G_vol_horiz_divergence_depth_integrated = G_vol_horiz_divergence.sum('k')
```

[46]:
```
# calculate time-mean using the month length weights
G_vol_horiz_divergence_depth_integrated_mean = \
```

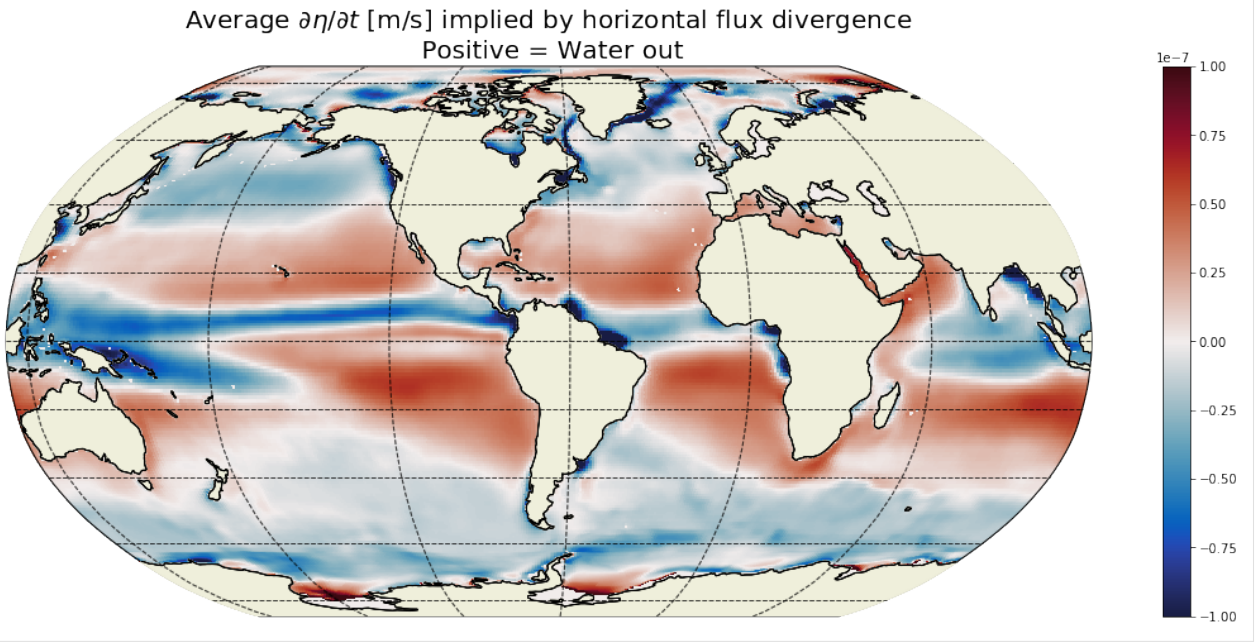
(continues on next page)

(continued from previous page)

```
    (G_vol_horiz_divergence_depth_integrated * month_length_weights).sum('time')
```

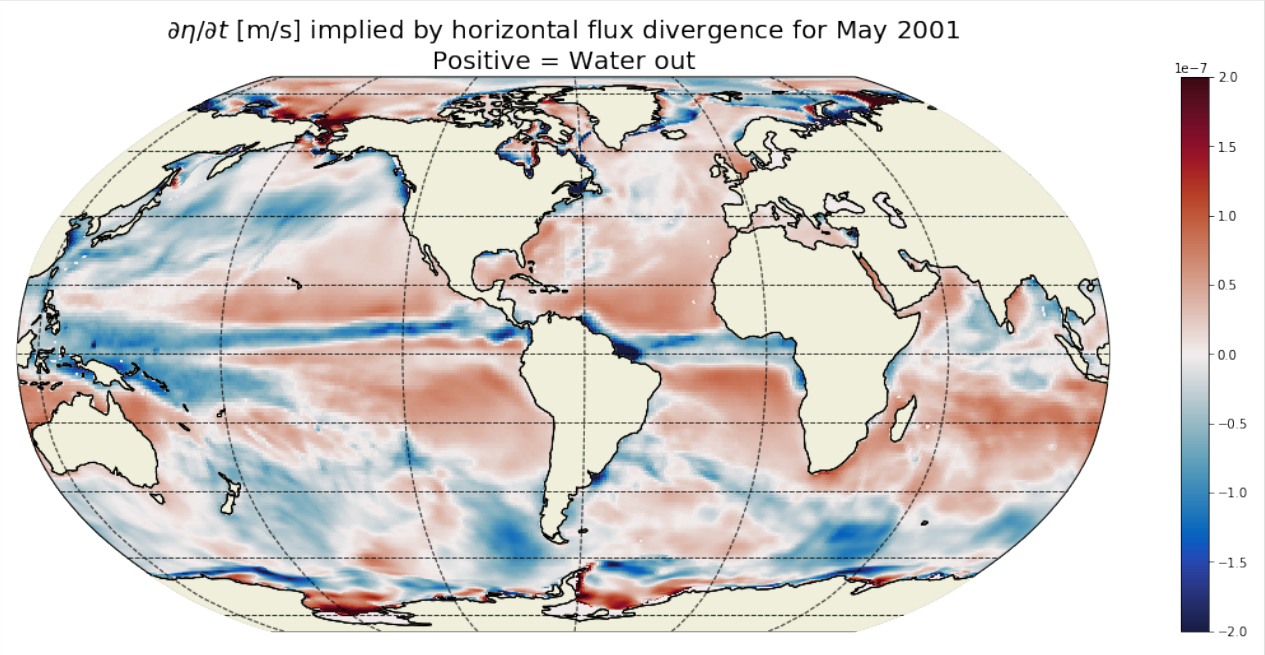
Plot the time-mean $G_{\text{horizontal flux divergence}}$

[47]:
```
plt.figure(figsize=(20,8));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, \
                              G_vol_horiz_divergence_depth_integrated_mean,
                              show_colorbar=True,\
                              cmin=-1e-7, cmax=1e-7, \
                              cmap=cmocean.cm.balance, user_lon_0=-67,\
                              dx=map_dx,dy=map_dy);
plt.title('Average $\partial \eta / \partial t$ [m/s] implied by horizontal flux␣
↪divergence\n Positive = Water out',
          fontsize=20);
```



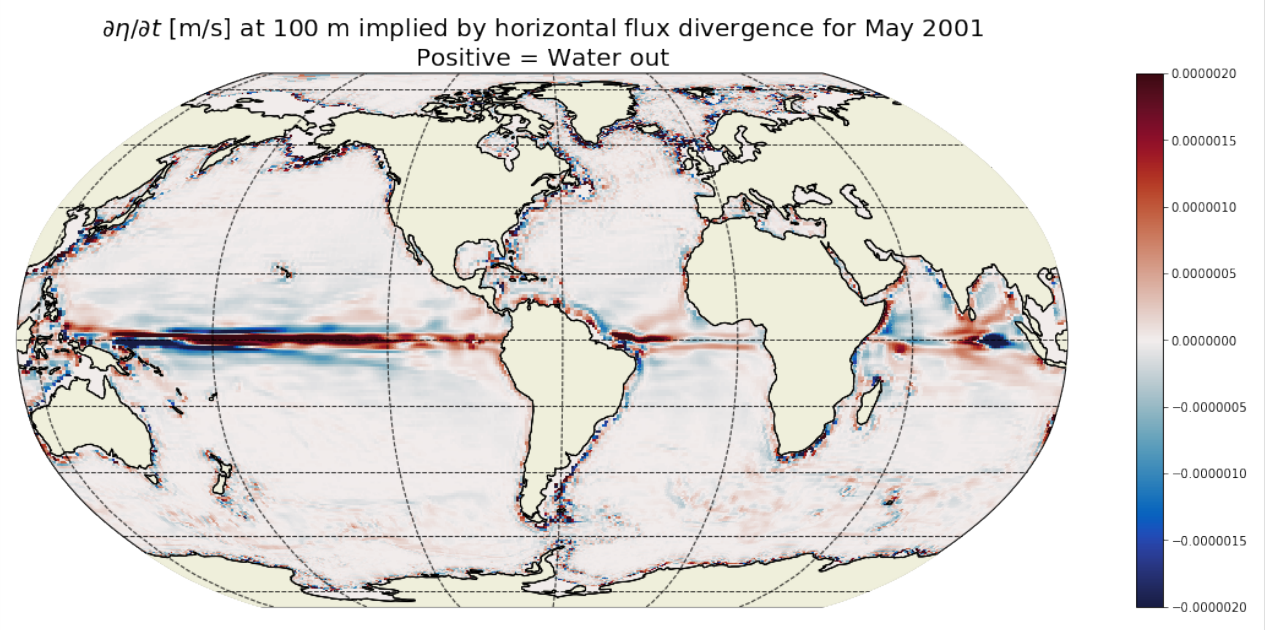
Plot one example $G_{\text{horizontal flux divergence}}$

[48]:
```
plt.figure(figsize=(20,8));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, \
                              G_vol_horiz_divergence_depth_integrated.isel(time=100),
                              show_colorbar=True,\
                              cmin=-2e-7, cmax=2e-7, \
                              cmap=cmocean.cm.balance, user_lon_0=-67,\
                              dx=map_dx,dy=map_dy);
plt.title('$\partial \eta / \partial t$ [m/s] implied by horizontal flux divergence for␣
↪May 2001\nPositive = Water out',
          fontsize=20);
```



Plot $G_{\text{horizontal flux divergence}}$ at 100m depth

[49]:
```
plt.figure(figsize=(20,8));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, \
                              G_vol_horiz_divergence.isel(k=10,time=100),
                              show_colorbar=True,\
                              cmin=-2e-6, cmax=2e-6, \
                              cmap=cmocean.cm.balance, user_lon_0=-67,\
                              dx=map_dx,dy=map_dy);
plt.title('$\partial \eta / \partial t$ [m/s] at 100 m implied by horizontal flux␣
↪divergence for May 2001\nPositive = Water out',
          fontsize=20);
```



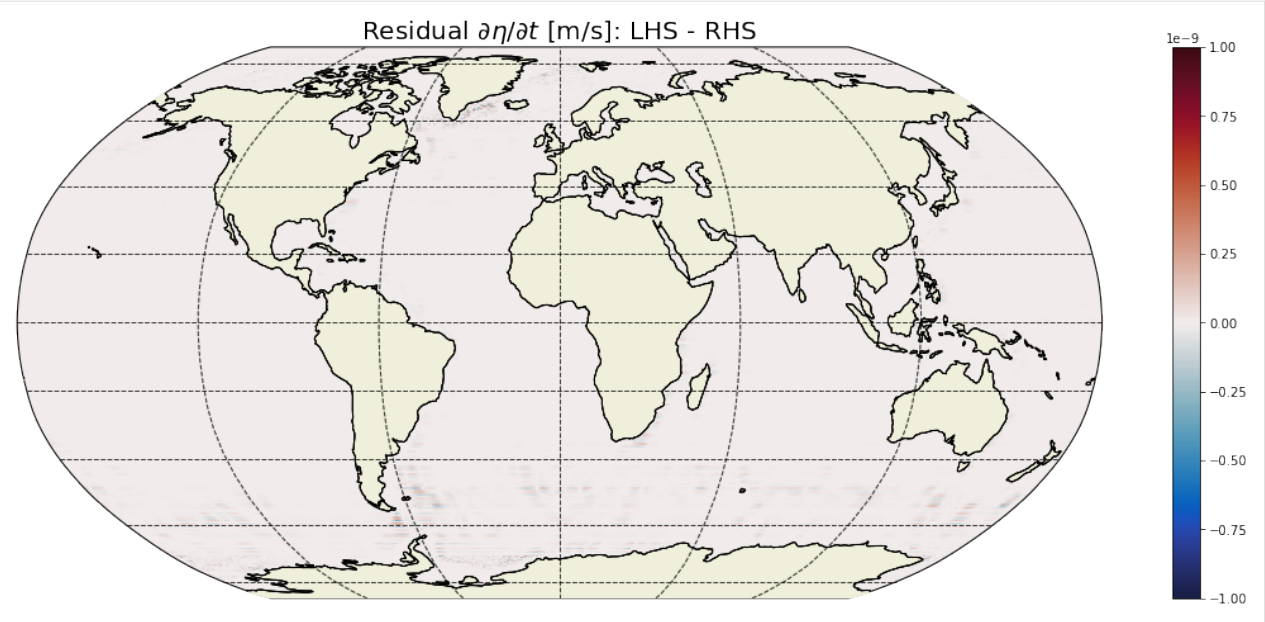
1.20.10 Comparison of LHS and RHS

Time mean difference of LHR and RHS

Plot the time-mean difference between the LHS and RHS of the volume budget equation.

[50]:
```
#LHS ETA TENDENCY
a = G_total_tendency_mean
# RHS ETA TENDENCY FROM VOL DIVERGENCE AND SURFACE FLUXES
b = G_vol_horiz_divergence_depth_integrated_mean
c = G_surf_fluxes_mean

delta = a - b -c
```

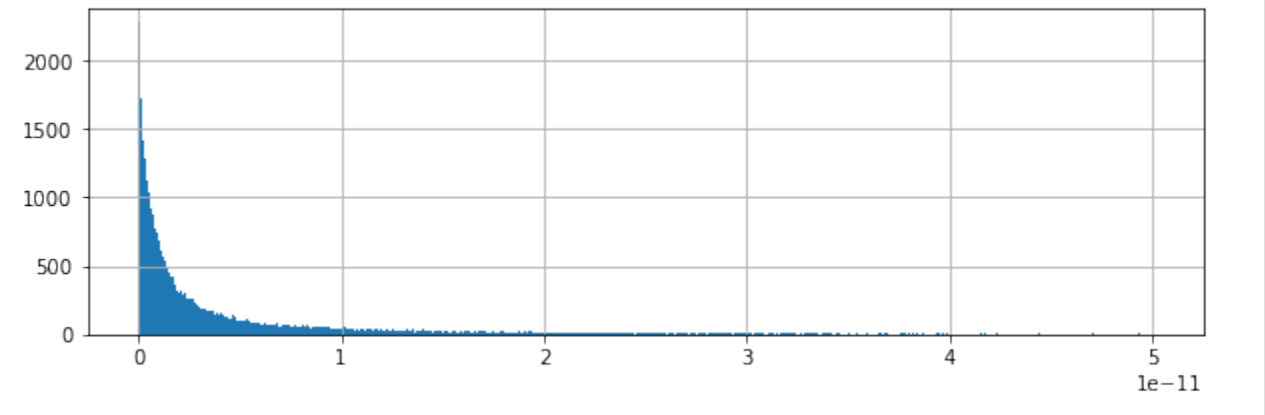
[51]:
```
plt.figure(figsize=(20,8));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, delta, \
                              show_colorbar=True,\
                              cmin=-1e-9, cmax=1e-9, \
                              cmap=cmocean.cm.balance, \
                              dx=map_dx,dy=map_dy);
plt.title('Residual $\partial \eta / \partial t$ [m/s]: LHS - RHS ',
          fontsize=20);
```



The residual of the time-mean surface velocity tendency terms is essentially zero. We can look at the distribution of residuals to get a little more confidence.

Histogram of residuals

[52]:
```
tmp = np.abs( a-b-c).values.ravel();
plt.figure(figsize=(10,3));

plt.hist(tmp[np.nonzero(tmp > 0)],np.linspace(0, .5e-10,1000));
plt.grid()
```



Almost all residuals < 10^{-11} m/s. We can close the ETAN budget using UVELMASS, VVELMASS, `WVELMASS` and `oceFWflx`.

> Note: As stated earlier, we don't actually need `oceFWflx` because the surface velocity of `WVELMASS` is proportional to `oceFWflx` but we kept it so that the surface forcing term is explicit.

1.20.11 ETAN budget closure through time

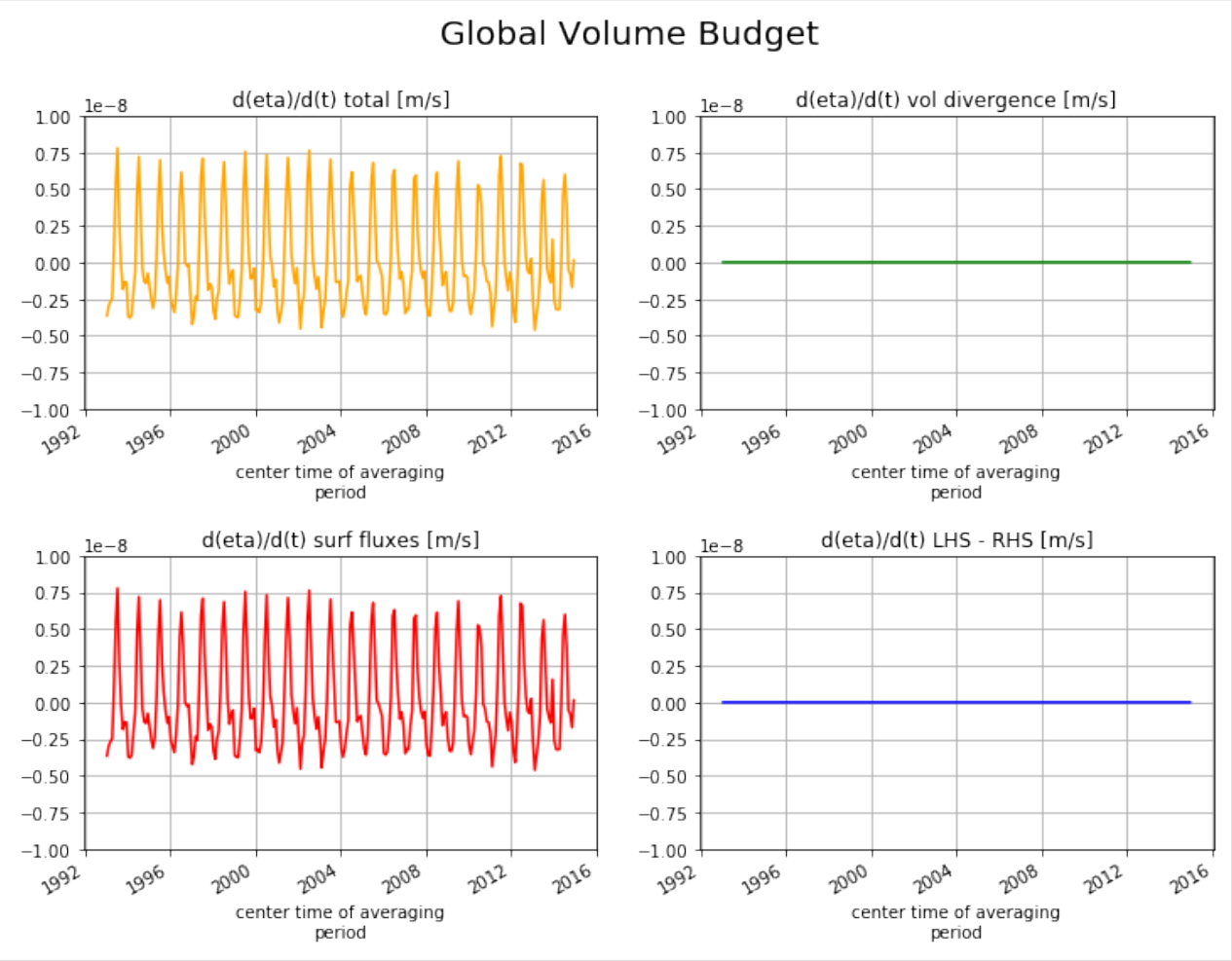
Global average ETAN budget closure

Another way of demonstrating volume budget closure is to show the global spatially-averaged ETAN tendency terms through time

[53]:
```
# LHR and RHS through time
a = G_total_tendency
b = G_vol_horiz_divergence_depth_integrated
c = G_surf_fluxes
# residuals
d = a-b-c

area_masked = ecco_grid.rA.where(ecco_grid.hFacC.isel(k=0)> 0)

# take area-weighted mean of these terms
tmp_a=(a*area_masked).sum(dim=('i','j','tile'))/area_masked.sum()
tmp_b=(b*area_masked).sum(dim=('i','j','tile'))/area_masked.sum()
tmp_c=(c*area_masked).sum(dim=('i','j','tile'))/area_masked.sum()
tmp_d=(d*area_masked).sum(dim=('i','j','tile'))/area_masked.sum()

# result is four time series
tmp_a.dims
```

[53]:
```
('time',)
```

[54]:
```
fig, axs = plt.subplots(2, 2, figsize=(12,8))

plt.sca(axs[0,0])
tmp_a.plot(color='orange')
axs[0,0].set_title("d(eta)/d(t) total [m/s]")
plt.grid()
plt.ylim([-1e-8, 1e-8]);

plt.sca(axs[0,1])
tmp_b.plot(color='g')
axs[0,1].set_title("d(eta)/d(t) vol divergence [m/s]")
plt.grid()
plt.ylim([-1e-8, 1e-8]);

plt.sca(axs[1,0])
tmp_c.plot(color='r')
axs[1,0].set_title("d(eta)/d(t) surf fluxes [m/s]")
plt.grid()
plt.ylim([-1e-8, 1e-8]);

plt.sca(axs[1,1])
tmp_d.plot(color='b')
axs[1,1].set_title("d(eta)/d(t) LHS - RHS [m/s]")
plt.grid()
plt.subplots_adjust(hspace = .5, wspace=.2)
plt.ylim([-1e-8, 1e-8])
plt.suptitle('Global Volume Budget',fontsize=20);
```



When averaged over the entire ocean surface the volumetric divergence has no net impact on $\partial\eta/\partial t$. This makes sence because horizontal flux divergence can only redistributes volume. Globally, η can only change via net surface fluxes.

Local ETAN budget closure

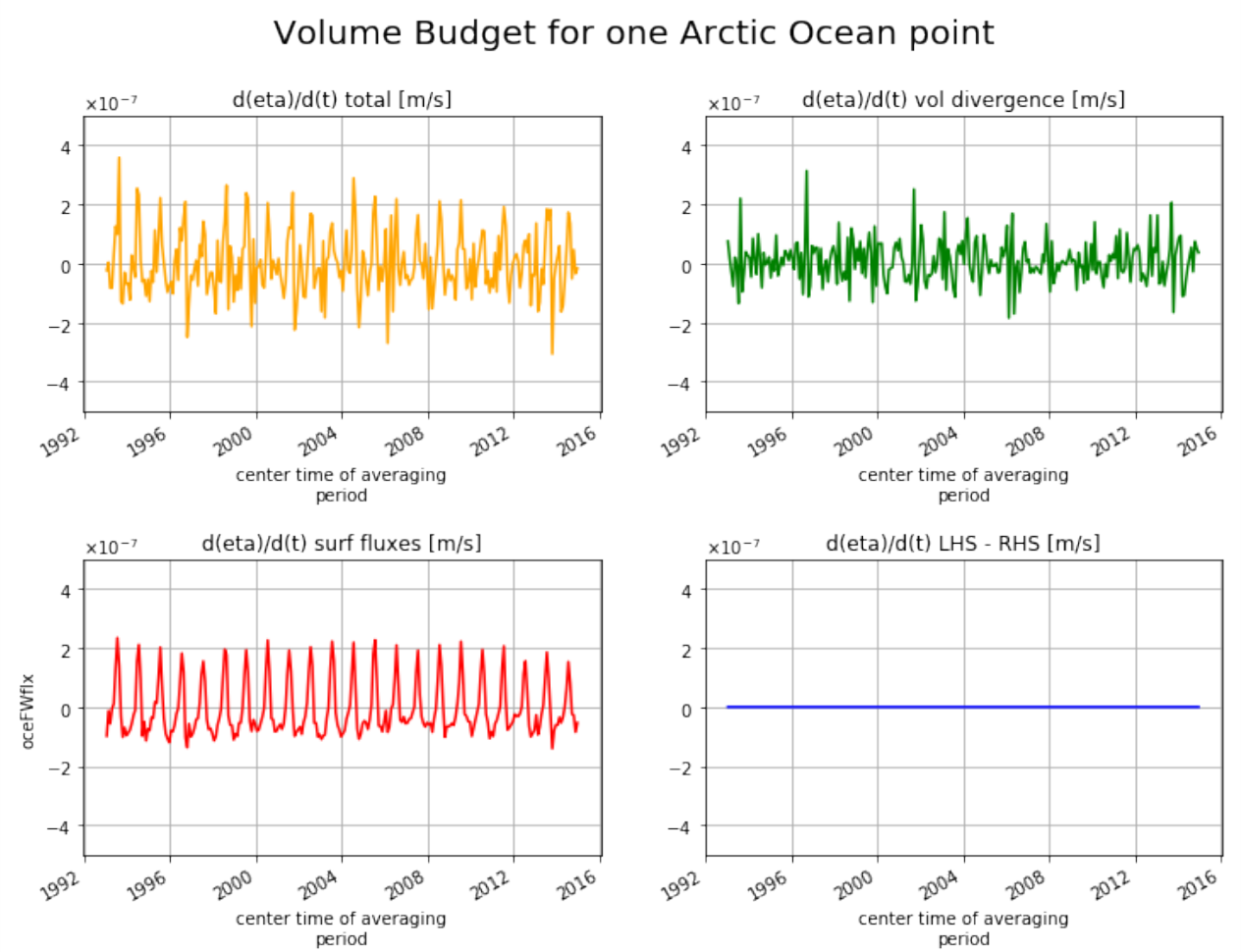
Locally we expect that volume divergence can impact η. This is demonstrated for a single point the Arctic.

[55]:
```
# Recall, from above...
#a = G_total_tendency
#b = G_vol_horiz_divergence_depth_integrated
#c = G_surf_fluxes
#d = a-b-c

tmp_aa = a.isel(tile=6,j=40,i=29)
tmp_bb = b.isel(tile=6,j=40,i=29)
tmp_cc = c.isel(tile=6,j=40,i=29)
tmp_dd = d.isel(tile=6,j=40,i=29)

fig, axs = plt.subplots(2, 2, figsize=(12,8))
plt.sca(axs[0,0])
```

(continues on next page)

(continued from previous page)

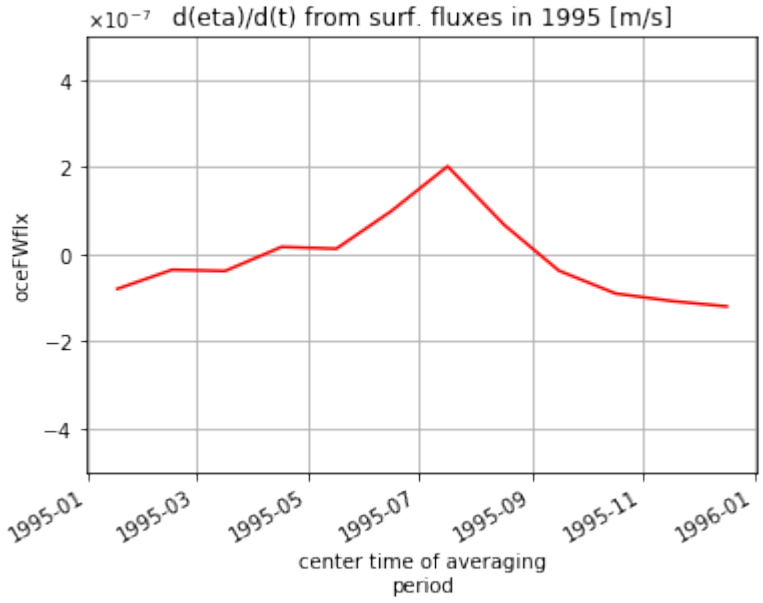
```
tmp_aa.plot(color='orange')
axs[0,0].set_title("d(eta)/d(t) total [m/s]")
plt.ticklabel_format(axis='y', style='sci', useMathText=True)
plt.ylim([-5e-7, 5e-7]);
plt.grid()

plt.sca(axs[0,1])
tmp_bb.plot(color='g')
axs[0,1].set_title("d(eta)/d(t) vol divergence [m/s]")
plt.ticklabel_format(axis='y', style='sci', useMathText=True)
plt.ylim([-5e-7, 5e-7]);
plt.grid()

plt.sca(axs[1,0])
tmp_cc.plot(color='r')
axs[1,0].set_title("d(eta)/d(t) surf fluxes [m/s]")
plt.ticklabel_format(axis='y', style='sci', useMathText=True)
plt.ylim([-5e-7, 5e-7]);
plt.grid()

plt.sca(axs[1,1])
tmp_dd.plot(color='b')
axs[1,1].set_title("d(eta)/d(t) LHS - RHS [m/s]")
plt.ticklabel_format(axis='y', style='sci', useMathText=True)
plt.ylim([-5e-7, 5e-7]);
plt.grid()

plt.subplots_adjust(hspace = .5, wspace=.2)
plt.suptitle('Volume Budget for one Arctic Ocean point',
                 fontsize=20);
```



Indeed, the volume divergence term does contribute to η variations at this one point.

The seasonal cycle of surface fluxes from sea-ice growth and melt can be seen in the surface fluxes term (plotted below for just the year 1995)

[56]:
```
tmp_cc.sel(time='1995').plot(color='r')
plt.ticklabel_format(axis='y', style='sci', useMathText=True)
plt.ylim([-5e-7, 5e-7]);
plt.grid()
plt.title('d(eta)/d(t) from surf. fluxes in 1995 [m/s]');
```



1.20.12 Predicted vs. actual η

As we have shown, in our Boussinesq model the only term that can change global mean model sea level anomaly η is net surface freshwater flux. Let us compare the time-evolution of η implied by surface freshwater fluxes and the actual η from the model output.

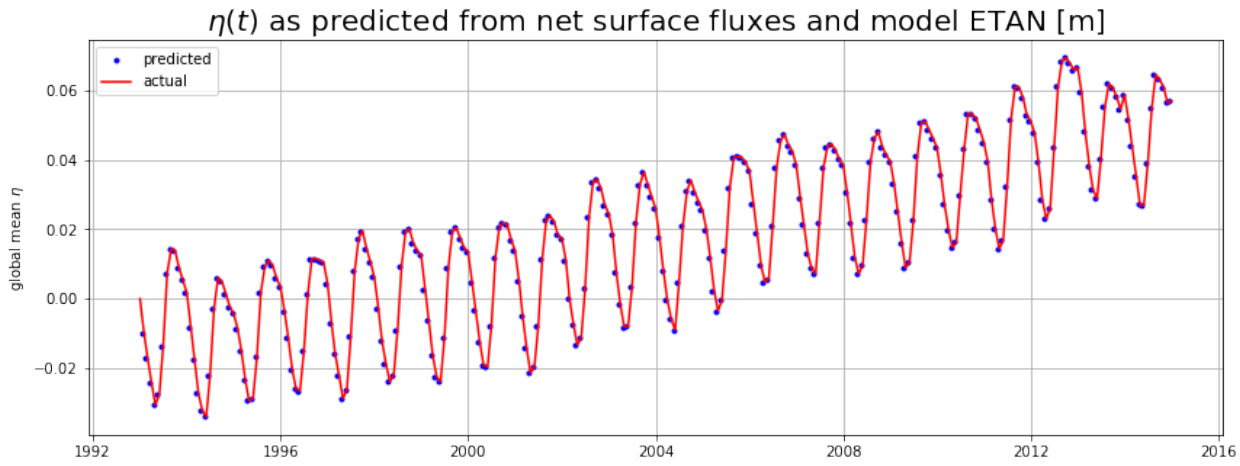
The predicted η time series is calculated by time integrating **:math:`G_{surface fluxes}`**.
This time series is compared against the actual η time series anomaly relative to the $\eta(t = 0)$.

[57]:
```
area_masked = ecco_grid.rA.where(ecco_grid.maskC.isel(k=0) == 1)

dETA_per_month_predicted_from_surf_fluxes = \
    ((G_surf_fluxes * area_masked).sum(dim=('i','j','tile')) /
     area_masked.sum())*secs_per_month

ETA_predicted_by_surf_fluxes = \
    np.cumsum(dETA_per_month_predicted_from_surf_fluxes.values)

ETA_from_ETAN =
    (ecco_monthly_snaps.ETAN * area_masked).sum(dim=('i','j','tile')) /
    area_masked.sum()

# plotting
plt.figure(figsize=(14,5));

plt.plot(dETA_per_month_predicted_from_surf_fluxes.time, \
         ETA_predicted_by_surf_fluxes,'b.')
plt.plot(ETA_from_ETAN.time.values, ETA_from_ETAN-ETA_from_ETAN[0],'r-')
```

(continues on next page)

(continued from previous page)

```
plt.grid()
plt.ylabel('global mean $\eta$');
plt.legend(('predicted', 'actual'));
plt.title('$\eta(t)$ as predicted from net surface fluxes and model ETAN [m]',
          fontsize=20);
```



The first predicted η occurs at the end of the first month (one month time integral of $\partial\eta/\partial t$. The first *actual* η is set to be zero.

The above plot is another way of confirming that in our Boussinesq model the only term that can change global mean model sea level anomaly η is net surface freshwater flux. To account for changes in global mean density we must apply the Greatbatch correction, inverse-barometer correction, and a correction term to account for the fact that sea-ice does not 'float' on top of the ocean but in fact displaces seawater upwards. All of these corrections are made for the term SSH, dynamic sea surface height anomaly (not shown here).

One can compare the sea level rise from mass fluxes in ECCO vs those estimated from GRACE and other datasets published the WCRP Global Sea Level Budget Group: Global sea-level budget 1993-present, Earth Syst. Sci. Data, 10, 1551-1590, https://doi.org/10.5194/essd-10-1551-2018, 2018.

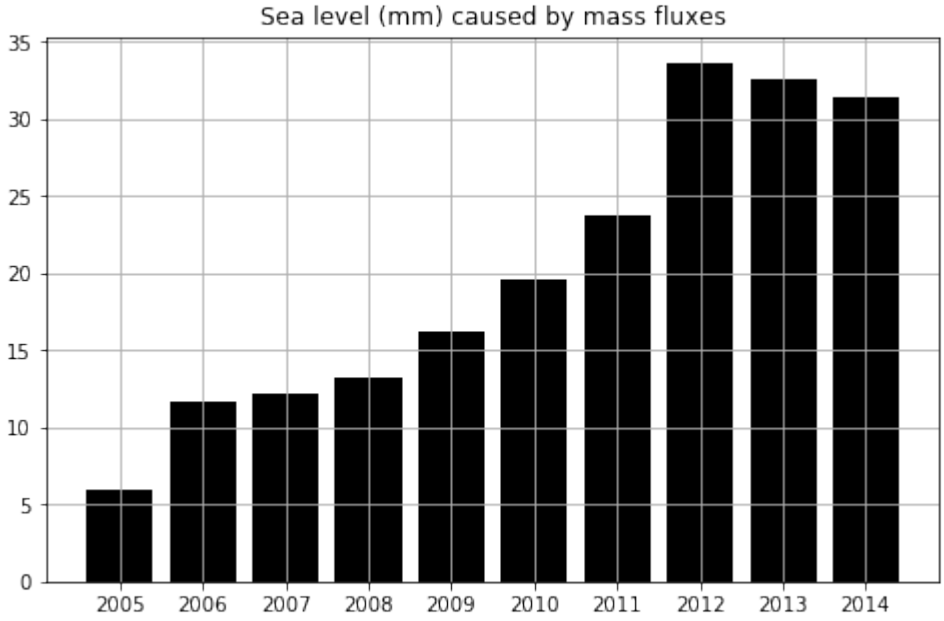
Available here. See Figure 16. https://www.earth-syst-sci-data.net/10/1551/2018/essd-10-1551-2018.pdf

[58]:
```
# Annual mean SL calculation must account for different lengths of each month.

# step 1. weight predicted eta by seconds in each month
tmp1=ETA_predicted_by_surf_fluxes*secs_per_month
# step 2, group records by year and sum across each year
tmp2=tmp1.groupby('time.year').sum()
# step 3, group secs per month by year and sum across each year
secs_per_year = secs_per_month.groupby('time.year').sum()

# step 3, divide time-weighted ETA by seconds per year
annual_mean_GMSL_due_to_mass_fluxes =tmp2/secs_per_year
num_years = len(annual_mean_GMSL_due_to_mass_fluxes.year.values)

plt.figure(figsize=(8,5));
# the -0.13 is to make the starting value comparable with WCRP fig 16.
plt.bar(annual_mean_GMSL_due_to_mass_fluxes.year.values[12:num_years],\
        1000*(annual_mean_GMSL_due_to_mass_fluxes.values[12:num_years]-.017), \
```

(continues on next page)

(continued from previous page)

```
        color='k')
plt.grid()
plt.xticks(np.arange(2005,
                     annual_mean_GMSL_due_to_mass_fluxes.year.values[-1]+1,step=1));
plt.title('Sea level (mm) caused by mass fluxes');
```



1.20.13 Time-mean $\partial\eta/\partial t$ due to net freshwater fluxes

We can easily calculate the time mean rate of global sea level rise due to net freshwater flux.

[59]:
```
x=(G_surf_fluxes_mean * area_masked).sum() / area_masked.sum()
total_GMSLR_due_to_mass_fluxes = x*seconds_in_entire_period
print('total global mean sea level rise due to mass fluxes m: ', \
      total_GMSLR_due_to_mass_fluxes.values)
```

```
total global mean sea level rise due to mass fluxes m:  0.05689936849813988
```

Dividing this by the total number of years in the analysis period gives a average rate per year:

[60]:
```
total_number_of_years = len(secs_per_month)/12
total_number_of_years
```

[60]:
```
22.0
```

[61]:
```
mean_rate_of_GMSLR_due_to_mass_fluxes = \
    total_GMSLR_due_to_mass_fluxes/total_number_of_years

print('mean rate of GMSLR due to mass fluxes [mm/yr] ',  \
      1000*mean_rate_of_GMSLR_due_to_mass_fluxes.values)
```

```
mean rate of GMSLR due to mass fluxes [mm/yr]  2.5863349317336306
```

Compare with other estimates of GMSLR due to mass fluxes.

1.21 Global Heat Budget Closure

Contributors: Jan-Erik Tesdal, Ryan Abernathey and Ian Fenty

A major part of this tutorial is based on "*A Note on Practical Evaluation of Budgets in ECCO Version 4 Release 3"* by Christopher G. Piecuch (https://ecco.jpl.nasa.gov/drive/files/Version4/Release3/doc/v4r3_budgets_howto.pdf). Calculation steps and Python code presented here are converted from the MATLAB code presented in the above reference.

1.21.1 Objectives

Evaluating and closing the heat budget over the global ocean.

1.21.2 Introduction

The ocean heat content (OHC) variability is described here with potential temperature (θ) which is given by the ECCOv4 diagnostic output `THETA`. The budget equation describing the change in θ is evaluated in general as

$$\frac{\partial \theta}{\partial t} = -\nabla \cdot (\theta \mathbf{u}) - \nabla \cdot \mathbf{F}^{\theta}_{\text{diff}} + F^{\theta}_{\text{forc}} \tag{1.10}$$

The heat budget includes the change in temperature over time ($\frac{\partial \theta}{\partial t}$), the convergence of heat advection ($-\nabla \cdot (\theta \mathbf{u})$) and heat diffusion ($-\nabla \cdot \mathbf{F}_{\text{diff}}$), plus downward heat flux from the atmosphere (F_{forc}). Note that in our definition F_{forc} contains both latent and sensible air-sea heat fluxes, longwave and shortwave radiation, as well as geothermal heat flux.

In the special case of ECCOv4, the heat budget is formulated as

$$\underbrace{\frac{\partial (s^* \theta)}{\partial t}}_{G^{\theta}_{\text{total}}} = \underbrace{-\nabla_{z^*} \cdot (s^* \theta \, \mathbf{v}_{res}) - \frac{\partial (\theta \, w_{res})}{\partial z^*}}_{G^{\theta}_{\text{advection}}} \underbrace{-s^* (\nabla \cdot \mathbf{F}^{\theta}_{\text{diff}})}_{G^{\theta}_{\text{diffusion}}} + \underbrace{s^* F^{\theta}_{\text{forc}}}_{G^{\theta}_{\text{forcing}}} \tag{1.11}$$

where $z^* = \frac{z-\eta}{H+\eta} H$ and $\nabla_{z^*} / \frac{\partial}{\partial z^*}$ are horizontal/vertical divergences in the z^* frame. Also note that the advection is now separated into horizontal ($\mathbf{v}_{res}$) and vertical (w_{res}) components, and there is a scaling factor ($s^* = 1 + \frac{\eta}{H}$) applied to the horizontal advection as well as the diffusion term ($G^{\theta}_{\text{diffusion}}$) and forcing term ($G^{\theta}_{\text{forcing}}$). s^* is a function of η which is the displacement of the ocean surface from its resting position of $z = 0$ (i.e., sea height anomaly). H is the ocean depth. s^* comes from the coordinate transformation from z to z^* (Campin and Adcroft, 2004; Campin et al., 2004). See ECCOv4 Global Volume Budget Closure for a more detailed explanation of the z^* coordinate system.

Note that the velocity terms in the ECCOv4 heat budget equation ($\mathbf{v}_{res}$ and w_{res}) are described as the "residual mean" velocities, which contain both the resolved (Eulerian) flow field, as well as the "GM bolus" velocity (i.e., parameterizing unresolved eddy effects):

$$(u_{res}, v_{res}, w_{res}) = (u, v, w) + (u_b, v_b, w_b)$$

Here (u_b, v_b, w_b) is the bolus velocity parameter, taking into account the correlation between velocity and thickness (also known as the eddy induced transportor the eddy advection term).

1.21.3 Evaluating the heat budget

We will evalute each term in the above heat budget

$$G^{\theta}_{\text{total}} = G^{\theta}_{\text{advection}} + G^{\theta}_{\text{diffusion}} + G^{\theta}_{\text{forcing}}$$

The total tendency of θ ($G^{\theta}_{\text{total}}$) is the sum of the θ tendencies from advective heat convergence ($G^{\theta}_{\text{advection}}$), diffusive heat convergence ($G^{\theta}_{\text{diffusion}}$) and total forcing ($G^{\theta}_{\text{forcing}}$).

We present calculation sequentially for each term starting with $G^{\theta}_{\text{total}}$ which will be derived by differencing instantaneous monthly snapshots of θ. The terms on the right hand side of the heat budget are derived from monthly-averaged fields.

1.21.4 Prepare environment and load ECCOv4 diagnostic output

Import relevant Python modules

[1]:
```
import numpy as np
import xarray as xr
```

[2]:
```
# Suppress warning messages for a cleaner presentation
import warnings
warnings.filterwarnings('ignore')
```

[3]:
```
## Import the ecco_v4_py library into Python
## =========================================

## -- If ecco_v4_py is not installed in your local Python library,
##    tell Python where to find it.

#import sys
#sys.path.append('/home/username/ECCOv4-py')

import ecco_v4_py as ecco
```

[4]:
```
# Plotting
import matplotlib.pyplot as plt
%matplotlib inline
```

Add relevant constants

[5]:
```
# Seawater density (kg/m^3)
rhoconst = 1029
## needed to convert surface mass fluxes to volume fluxes

# Heat capacity (J/kg/K)
c_p = 3994

# Constants for surface heat penetration (from Table 2 of Paulson and Simpson, 1977)
R = 0.62
zeta1 = 0.6
zeta2 = 20.0
```

Load ecco_grid

[6]:
```
## Set top-level file directory for the ECCO NetCDF files
## ======================================================================

# Define main directory
base_dir = '/home/username/ECCOv4-release'

# Define ECCO version
ecco_version = 'v4r3'

# Define a high-level directory for ECCO fields
ECCO_dir = base_dir + '/Release3_alt'
```

Note: Change `base_dir` to your own directory path.

[7]:
```
# Load the model grid
grid_dir= ECCO_dir + '/nctiles_grid/'
ecco_grid = ecco.load_ecco_grid_nc(grid_dir, 'ECCOv4r3_grid.nc')
```

Volume

Calculate the volume of each grid cell. This is used when converting advective and diffusive flux convergences and calculating volume-weighted averages.

[8]:
```
# Volume (m^3)
vol = (ecco_grid.rA*ecco_grid.drF*ecco_grid.hFacC).transpose('tile','k','j','i')
```

Load monthly snapshots

[9]:
```
data_dir= ECCO_dir + '/nctiles_monthly_snapshots'

year_start = 1993
year_end = 2017

# Load one extra year worth of snapshots
ecco_monthly_snaps = ecco.recursive_load_ecco_var_from_years_nc(data_dir, \
                         vars_to_load=['ETAN','THETA'],\
                         years_to_load=range(year_start, year_end+1))

num_months = len(ecco_monthly_snaps.time.values)
# Drop the last 11 months so that we have one snapshot at the beginning and end of each␣
↪month within the
# range 1993/1/1 to 2015/1/1

ecco_monthly_snaps = ecco_monthly_snaps.isel(time=np.arange(0, num_months-11))
```

```
loading files of  ETAN
loading files of  THETA
```

[10]:
```
# 1993-01 (beginning of first month) to 2015-01-01 (end of last month, 2014-12)
print(ecco_monthly_snaps.ETAN.time.isel(time=[0, -1]).values)
```

```
['1993-01-01T00:00:00.000000000' '2015-01-01T00:00:00.000000000']
```

[11]:
```
# Find the record of the last snapshot
## This is used to defined the exact period for monthly mean data
last_record_date = ecco.extract_yyyy_mm_dd_hh_mm_ss_from_datetime64(ecco_monthly_snaps.
↪time[-1].values)
print(last_record_date)
```

```
(2015, 1, 1, 0, 0, 0)
```

Load monthly mean data

[12]:
```
data_dir= ECCO_dir + '/nctiles_monthly'

year_end = last_record_date[0]
ecco_monthly_mean = ecco.recursive_load_ecco_var_from_years_nc(data_dir, \
                        vars_to_load=['TFLUX','oceQsw','ADVx_TH','ADVy_TH','ADVr_TH',
                                      'DFxE_TH','DFyE_TH','DFrE_TH','DFrI_TH'],\
                        years_to_load=range(year_start, year_end))
```

```
loading files of  ADVr_TH
loading files of  ADVx_TH
loading files of  ADVy_TH
loading files of  DFrE_TH
loading files of  DFrI_TH
loading files of  DFxE_TH
loading files of  DFyE_TH
loading files of  TFLUX
loading files of  oceQsw
```

[13]:
```
# Print first and last time points of the monthly-mean records
print(ecco_monthly_mean.time.isel(time=[0, -1]).values)
```

```
['1993-01-16T12:00:00.000000000' '2014-12-16T12:00:00.000000000']
```

Each monthly mean record is bookended by a snapshot. We should have one more snapshot than monthly mean record.

[14]:
```
print('Number of monthly mean records: ', len(ecco_monthly_mean.time))
print('Number of monthly snapshot records: ', len(ecco_monthly_snaps.time))
```

```
Number of monthly mean records:  264
Number of monthly snapshot records:  265
```

[15]:
```
# Drop superfluous coordinates (We already have them in ecco_grid)
ecco_monthly_mean = ecco_monthly_mean.reset_coords(drop=True)
```

Merge dataset of monthly mean and snapshots data

Merge the two datasets to put everything into one single dataset

[16]:
```
ds = xr.merge([ecco_monthly_mean,
               ecco_monthly_snaps.rename({'time':'time_snp','ETAN':'ETAN_snp', 'THETA':
↪'THETA_snp'})])
```

Create the xgcm 'grid' object

The `xgcm` 'grid' object is used to calculate the flux divergences across different tiles of the lat-lon-cap grid and the time derivatives from `THETA` snapshots

[17]:
```
# Change time axis of the snapshot variables
ds.time_snp.attrs['c_grid_axis_shift'] = 0.5
```

[18]:
```
grid = ecco.get_llc_grid(ds)
```

Number of seconds in each month

The xgcm `grid` object includes information on the time axis, such that we can use it to get Δt, which is the time span between the beginning and end of each month (in seconds).

[19]:
```
delta_t = grid.diff(ds.time_snp, 'T', boundary='fill', fill_value=np.nan)

# Convert to seconds
delta_t = delta_t.astype('f4') / 1e9
```

1.21.5 Calculate total tendency of θ ($G^{\theta}_{\text{total}}$)

We calculate the monthly-averaged time tendency of `THETA` by differencing monthly `THETA` snapshots. Remember that we need to include a scaling factor due to the nonlinear free surface formulation. Thus, we need to use snapshots of both `ETAN` and `THETA` to evaluate $s^*\theta$.

[20]:
```
# Calculate the s*theta term
sTHETA = ds.THETA_snp*(1+ds.ETAN_snp/ecco_grid.Depth)
```

[21]:
```
# Total tendency (psu/s)
G_total = grid.diff(sTHETA, 'T', boundary='fill', fill_value=0.0)/delta_t
```

> **Note**: Unlike the monthly snapshots `ETAN_snp` and `THETA_snp`, the resulting data array `G_total` has now the same time values as the time-mean fields (middle of the month).

Plot the time-mean $\partial\theta/\partial t$, total $\Delta\theta$, and one example $\partial\theta/\partial t$ field

Time-mean $\partial\theta/\partial t$

The time-mean $\partial\theta/\partial t$ (i.e., $\overline{G^{\theta}_{\text{total}}}$), is given by

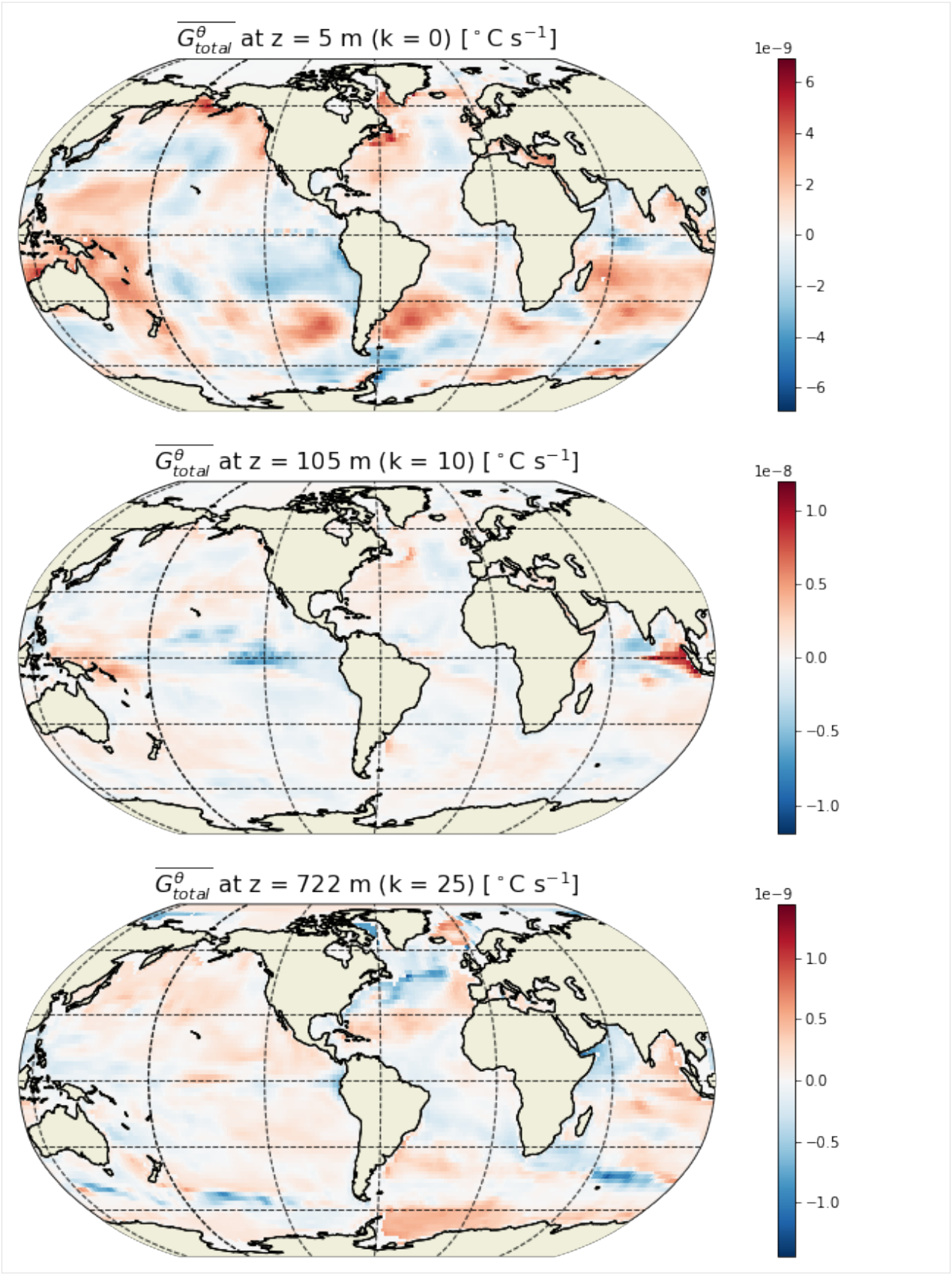
$\overline{G^{\theta}_{\text{total}}} = \sum_{i=1}^{nm} w_i G^{\theta}_{\text{total}}$

with $\sum_{i=1}^{nm} w_i = 1$ and nm=number of months

[22]:
```
# The weights are just the number of seconds per month divided by total seconds
month_length_weights = delta_t / delta_t.sum()
```

[23]:
```
# The weighted mean weights by the length of each month (in seconds)
G_total_mean = (G_total*month_length_weights).sum('time')
```

[24]:
```
plt.figure(figsize=(15,15))

for idx, k in enumerate([0,10,25]):
    p = ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, G_total_mean[:,k],show_
↪colorbar=True,
                                      cmap='RdBu_r', user_lon_0=-67, dx=2, dy=2, subplot_
↪grid=[3,1,idx+1]);
    p[1].set_title(r'$\overline{G^\theta_{total}}$ at z = %i m (k = %i) [$^\circ$C s$^{-
↪1}$]'\
                   %(np.round(-ecco_grid.Z[k].values),k), fontsize=16)
```


$\overline{G^{\theta}_{total}}$ at z = 5 m (k = 0) [$^{\circ}$C s^{-1}]
1e−9
6
4
2
0
−2
−4
−6
$\overline{G^{\theta}_{total}}$ at z = 105 m (k = 10) [$^{\circ}$C s^{-1}]
1e−8
1.0
0.5
0.0
−0.5
−1.0
$\overline{G^{\theta}_{total}}$ at z = 722 m (k = 25) [$^{\circ}$C s^{-1}]
1e−9
1.0
0.5
0.0
−0.5
−1.0

Total $\Delta\theta$

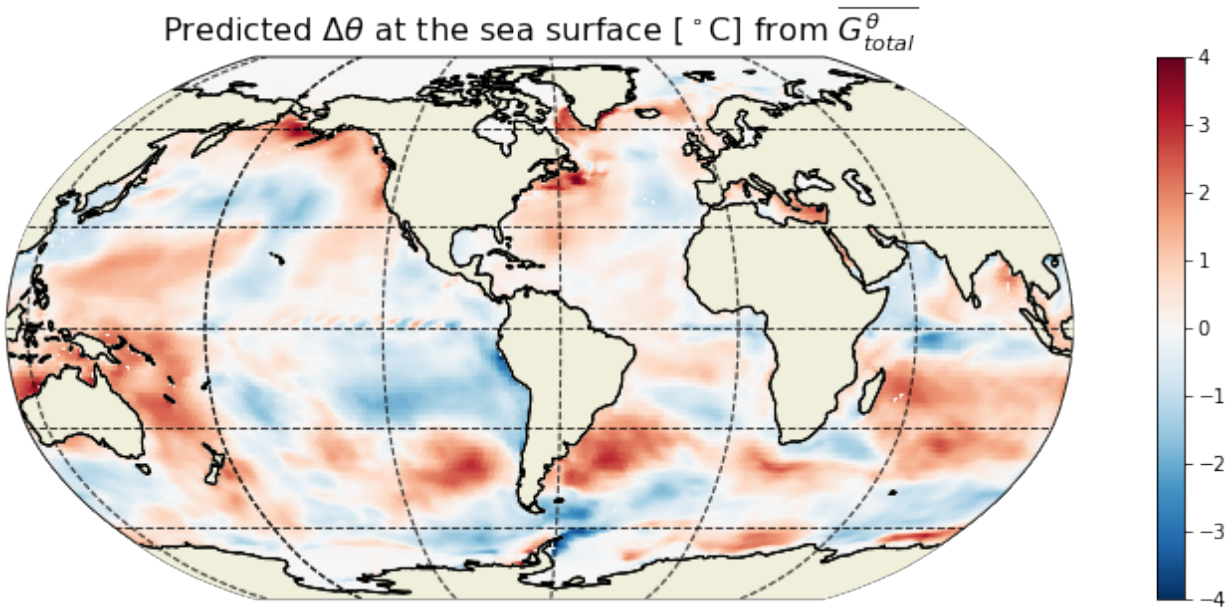
How much did THETA change over the analysis period?

[25]:
```
# The number of seconds in the entire period
seconds_in_entire_period = \
    float(ds.time_snp[-1] - ds.time_snp[0])/1e9
print ('seconds in analysis period: ', seconds_in_entire_period)

# which is also the sum of the number of seconds in each month
print('Sum of seconds in each month ', delta_t.sum().values)
```

```
seconds in analysis period:  694224000.0
Sum of seconds in each month  694224000.0
```

[26]:
```
THETA_delta = G_total_mean*seconds_in_entire_period
```

[27]:
```
plt.figure(figsize=(15,5));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, \
                              THETA_delta[:,0],show_colorbar=True,\
                              cmin=-4, cmax=4, \
                              cmap='RdBu_r', user_lon_0=-67, dx=0.2, dy=0.2);
plt.title(r'Predicted $\Delta \theta$ at the sea surface [$^\circ$C] from $\overline{G^\
↪theta_{total}}$',fontsize=16);
```



We can sanity check the total THETA change that we found by multipling the time-mean THETA tendency with the number of seconds in the simulation by comparing that with the difference in THETA between the end of the last month and start of the first month.

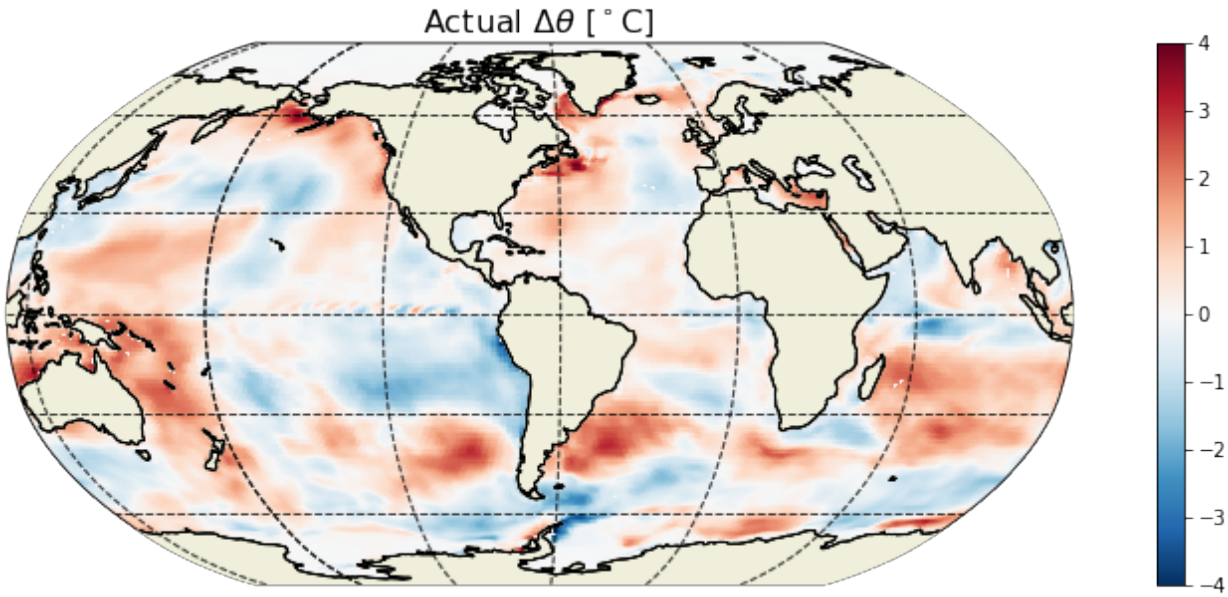
[28]:
```
THETA_delta_method_2 = ds.THETA_snp.isel(time_snp=-1) - ds.THETA_snp.isel(time_snp=0)
```

[29]:
```
plt.figure(figsize=(15,5));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, \
```

(continues on next page)

(continued from previous page)

```
                                THETA_delta_method_2[:,0],show_colorbar=True,\
                                cmin=-4, cmax=4, \
                                cmap='RdBu_r', user_lon_0=-67, dx=0.2, dy=0.2);
plt.title(r'Actual $\Delta \theta$ [$^\circ$C]', fontsize=16);
```

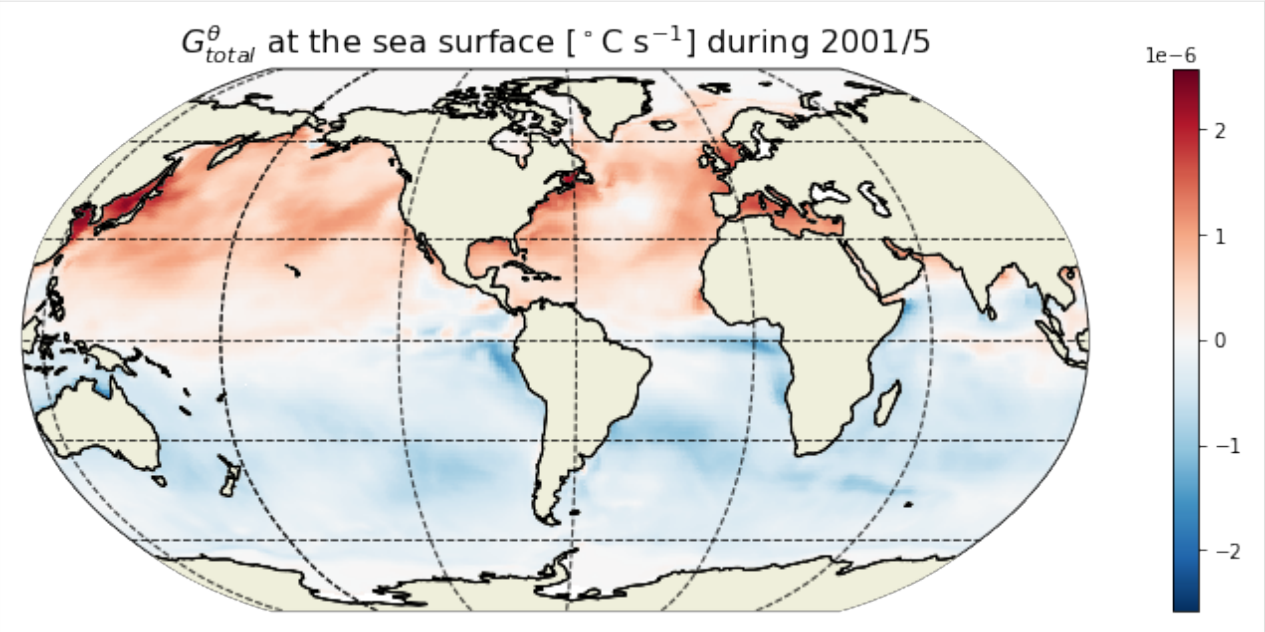


Example G^{θ}_{total} field at a particular time

[30]:
```
# get an array of YYYY, MM, DD, HH, MM, SS for
#dETAN_dT_perSec at time index 100
tmp = ecco.extract_yyyy_mm_dd_hh_mm_ss_from_datetime64(G_total.time[100].values)
print(tmp)
```

```
(2001, 5, 16, 12, 0, 0)
```

[31]:
```
plt.figure(figsize=(15,5));
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, G_total.isel(time=100)[:,0],␣
↪show_colorbar=True,
                                cmap='RdBu_r', user_lon_0=-67, dx=0.2, dy=0.2);

plt.title(r'$G^\theta_{total}$ at the sea surface [$^\circ$C s$^{-1}$] during ' +
          str(tmp[0]) +'/' + str(tmp[1]), fontsize=16);
```



For any given month the time rate of change of `THETA` is strongly dependent on the season. In the above we are looking at May 2001. We see positive `THETA` tendency in the northern hemisphere and cooling in the southern hemisphere.

1.21.6 Calculate tendency due to advective convergence ($G^{\theta}_{\text{advection}}$)

Horizontal convergence of advective heat flux

The relevant fields from the diagnostic output here are - `ADVx_TH`: U Component Advective Flux of Potential Temperature (degC m^3/s) - `ADVy_TH`: V Component Advective Flux of Potential Temperature (degC m^3/s)

The xgcm `grid` object is then used to take the convergence of the horizontal heat advection.

[32]:
```
ADVxy_diff = grid.diff_2d_vector({'X' : ds.ADVx_TH, 'Y' : ds.ADVy_TH}, boundary = 'fill')

# Convergence of horizontal advection (degC m^3/s)
adv_hConvH = (-(ADVxy_diff['X'] + ADVxy_diff['Y']))
```

Vertical convergence of advective heat flux

The relevant field from the diagnostic output is - `ADVr_TH`: Vertical Advective Flux of Potential Temperature (degC m^3/s)

[33]:
```
# Load monthly averages of vertical advective flux
ADVr_TH = ds.ADVr_TH.transpose('time','tile','k_l','j','i')
```

> **Note**: For `ADVr_TH`, `DFrE_TH` and `DFrI_TH`, we need to make sure that sequence of dimensions are consistent. When loading the fields use `.transpose('time','tile','k_l','j','i')`. Otherwise, the divergences will be not correct (at least for `tile = 12`).

[34]:
```
# Convergence of vertical advection (degC m^3/s)
adv_vConvH = grid.diff(ADVr_TH, 'Z', boundary='fill')
```

Note: In case of the volume budget (and salinity conservation), the surface forcing (`oceFWflx`) is already included at the top level (`k_l = 0`) in `WVELMASS`. Thus, to keep the surface forcing term explicitly represented, one needs to zero out the values of `WVELMASS` at the surface so as to avoid double counting (see `ECCO_v4_Volume_budget_closure.ipynb`). This is not the case for the heat budget. `ADVr_TH` does not include the sea surface forcing. Thus, the vertical advective flux (at the air-sea interface) should not be zeroed out.

Total convergence of advective flux ($G^{\theta}_{\text{advection}}$)

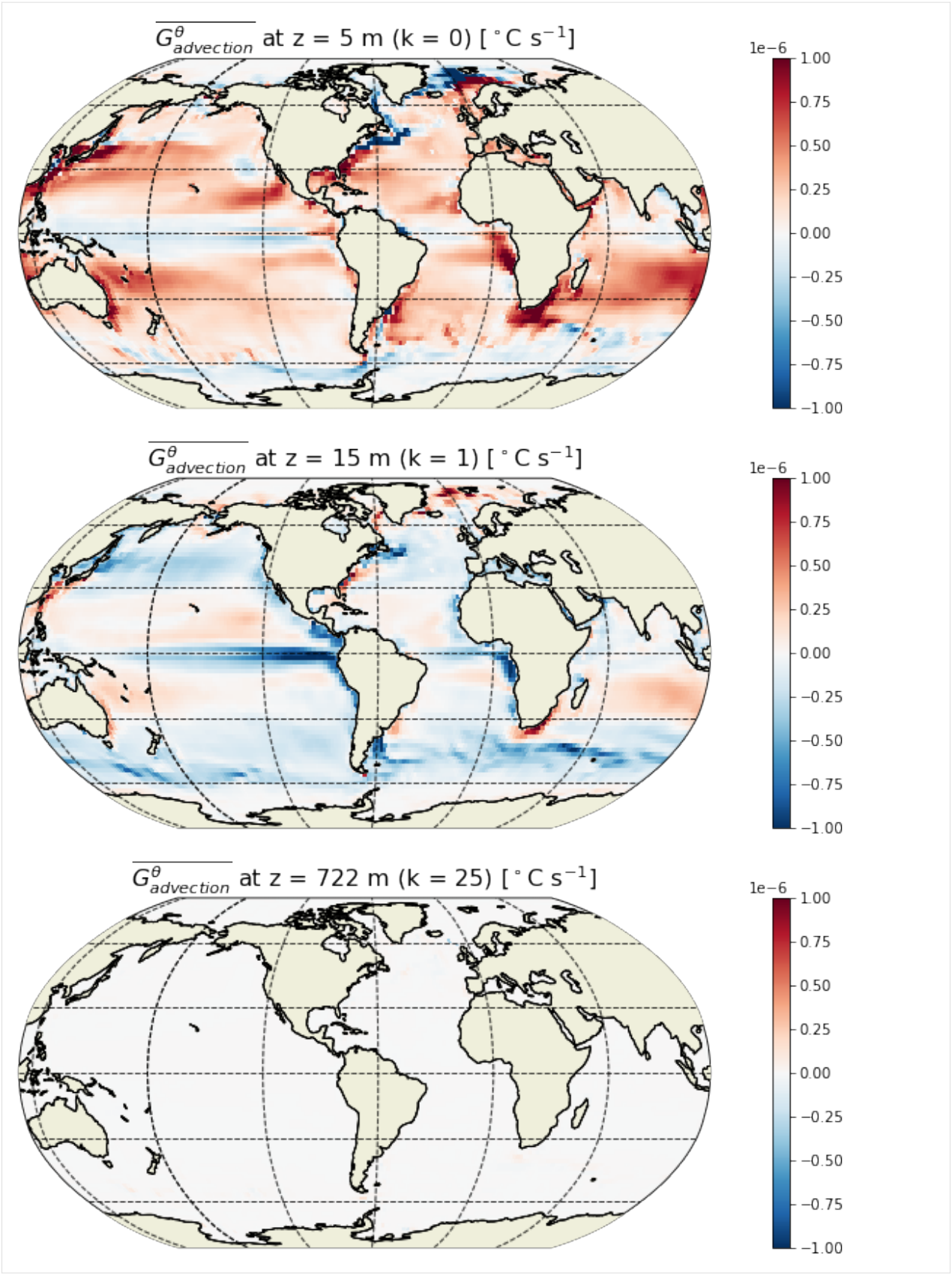
We can get the total convergence by simply adding the horizontal and vertical component.

[35]:
```
# Sum horizontal and vertical convergences and divide by volume (degC/s)
G_advection = (adv_hConvH + adv_vConvH)/vol
```

Plot the time-mean $G^{\theta}_{\text{advection}}$

[36]:
```
G_advection_mean = (G_advection*month_length_weights).sum('time')
```

[37]:
```
plt.figure(figsize=(15,15))

for idx, k in enumerate([0,1,25]):
    p = ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, G_advection_mean[:,k],
↪show_colorbar=True,
                                      cmin=-1e-6, cmax=1e-6, cmap='RdBu_r', user_lon_0=-
↪67, dx=2, dy=2,
                                      subplot_grid=[3,1,idx+1]);
    p[1].set_title(r'$\overline{G^\theta_{advection}}$ at z = %i m (k = %i) [$^\circ$C s
↪$^{-1}$]'\
                   %(np.round(-ecco_grid.Z[k].values),k), fontsize=16)
```

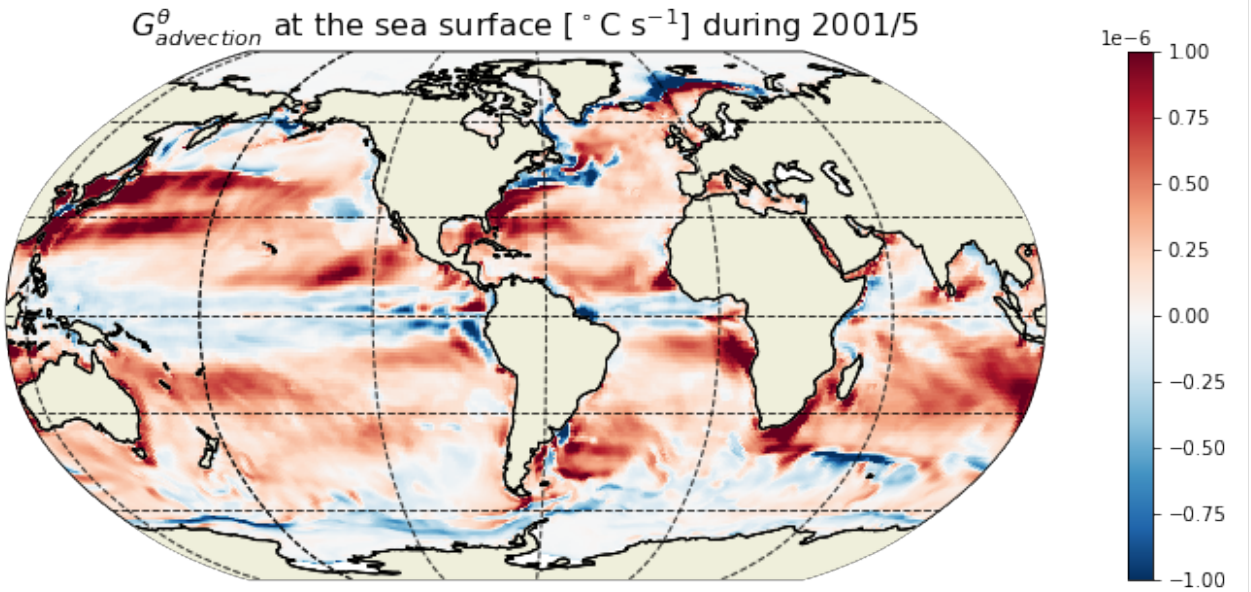

$\overline{G^{\theta}_{advection}}$ at z = 5 m (k = 0) [$^{\circ}$C s^{-1}]
$\overline{G^{\theta}_{advection}}$ at z = 15 m (k = 1) [$^{\circ}$C s^{-1}]
$\overline{G^{\theta}_{advection}}$ at z = 722 m (k = 25) [$^{\circ}$C s^{-1}]
1e−6
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

Example $G^{\theta}_{\text{advection}}$ field at a particular time

[38]:
```
tmp = ecco.extract_yyyy_mm_dd_hh_mm_ss_from_datetime64(G_advection.time[100].values)
print(tmp)
```

```
(2001, 5, 16, 12, 0, 0)
```

[39]:
```
plt.figure(figsize=(15,5));

ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, G_advection.isel(time=100)[:,
↪0], show_colorbar=True,
                              cmin=-1e-6, cmax=1e-6, cmap='RdBu_r', user_lon_0=-67, dx=0.
↪2, dy=0.2)
plt.title(r'$G^\theta_{advection}$ at the sea surface [$^\circ$C s$^{-1}$] during ' +
          str(tmp[0]) +'/' + str(tmp[1]), fontsize=16)
plt.show()
```



1.21.7 Calculate tendency due to diffusive convergence ($G^{\theta}_{\text{diffusion}}$)

Horizontal convergence of advective heat flux

The relevant fields from the diagnostic output here are - `DFxE_TH`: U Component Diffusive Flux of Potential Temperature (degC m^3/s) - `DFyE_TH`: V Component Diffusive Flux of Potential Temperature (degC m^3/s)

As with advective fluxes, we use the xgcm `grid` object to calculate the convergence of horizontal heat diffusion.

[40]:
```
DFxyE_diff = grid.diff_2d_vector({'X' : ds.DFxE_TH, 'Y' : ds.DFyE_TH}, boundary = 'fill')

# Convergence of horizontal diffusion (degC m^3/s)
dif_hConvH = (-(DFxyE_diff['X'] + DFxyE_diff['Y']))
```

Vertical convergence of advective heat flux

The relevant fields from the diagnostic output are - DFrE_TH: Vertical Diffusive Flux of Potential Temperature (Explicit part) (degC m^3/s) - DFrI_TH: Vertical Diffusive Flux of Potential Temperature (Implicit part) (degC m^3/s) > **Note**: Vertical diffusion has both an explicit (DFrE_TH) and an implicit (DFrI_TH) part.

[41]:
```
# Load monthly averages of vertical diffusive fluxes
DFrE_TH = ds.DFrE_TH.transpose('time','tile','k_l','j','i')
DFrI_TH = ds.DFrI_TH.transpose('time','tile','k_l','j','i')

# Convergence of vertical diffusion (degC m^3/s)
dif_vConvH = grid.diff(DFrE_TH, 'Z', boundary='fill') + grid.diff(DFrI_TH, 'Z', boundary=
↪'fill')
```

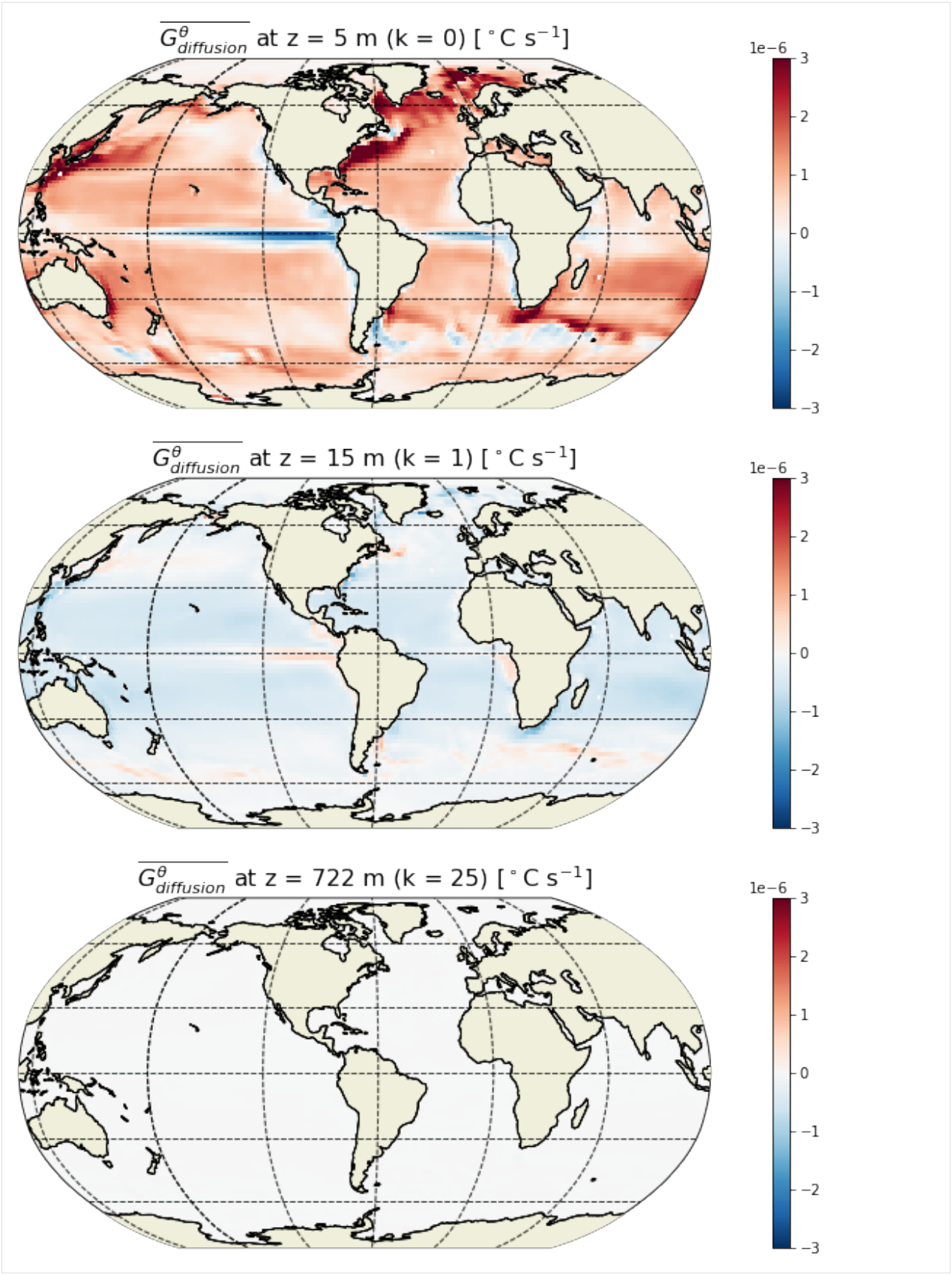
Total convergence of diffusive flux ($G^{\theta}_{\text{diffusion}}$)

[42]:
```
# Sum horizontal and vertical convergences and divide by volume (degC/s)
G_diffusion = (dif_hConvH + dif_vConvH)/vol
```

Plot the time-mean $G^{\theta}_{\text{diffusion}}$

[43]:
```
G_diffusion_mean = (G_diffusion*month_length_weights).sum('time')
```

[44]:
```
plt.figure(figsize=(15,15))

for idx, k in enumerate([0,1,25]):
    p = ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, G_diffusion_mean[:,k],
↪show_colorbar=True,
                                      cmin=-3e-6, cmax=3e-6, cmap='RdBu_r', user_lon_0=-
↪67, dx=2, dy=2,
                                      subplot_grid=[3,1,idx+1]);
    p[1].set_title(r'$\overline{G^\theta_{diffusion}}$ at z = %i m (k = %i) [$^\circ$C s
↪$^{-1}$]'\
                   %(np.round(-ecco_grid.Z[k].values),k), fontsize=16)
```

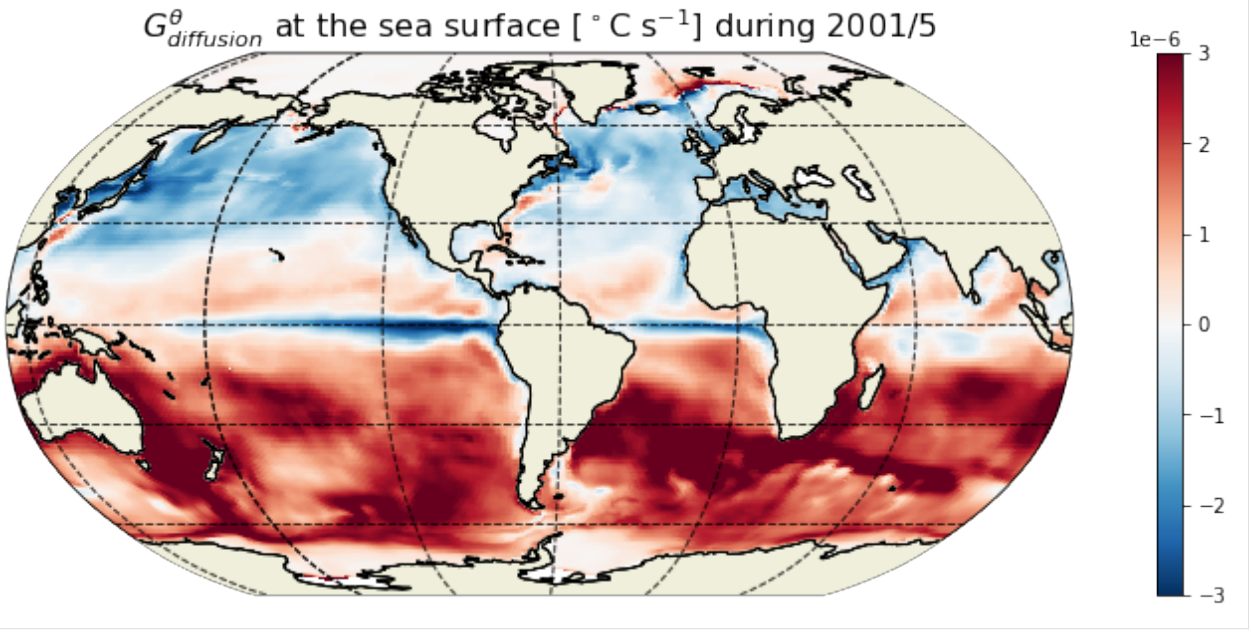

$\overline{G^{\theta}_{diffusion}}$ at z = 5 m (k = 0) [$^{\circ}$C s^{-1}]
1e−6
3
2
1
0
−1
−2
−3
$\overline{G^{\theta}_{diffusion}}$ at z = 15 m (k = 1) [$^{\circ}$C s^{-1}]
1e−6
3
2
1
0
−1
−2
−3
$\overline{G^{\theta}_{diffusion}}$ at z = 722 m (k = 25) [$^{\circ}$C s^{-1}]
1e−6
3
2
1
0
−1
−2
−3

Example $G^{\theta}_{\text{diffusion}}$ field at a particular time

[45]:
```
tmp = ecco.extract_yyyy_mm_dd_hh_mm_ss_from_datetime64(G_diffusion.time[100].values)
print(tmp)
```

```
(2001, 5, 16, 12, 0, 0)
```

[46]:
```
plt.figure(figsize=(15,5));

ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, G_diffusion.isel(time=100)[:,
↪0],show_colorbar=True,
                              cmin=-3e-6, cmax=3e-6, cmap='RdBu_r', user_lon_0=-67, dx=0.
↪2, dy=0.2)
plt.title(r'$G^\theta_{diffusion}$ at the sea surface [$^\circ$C s$^{-1}$] during ' +
          str(tmp[0]) +'/' + str(tmp[1]), fontsize=16)
plt.show()
```



1.21.8 Calculate tendency due to forcing ($G^{\theta}_{\text{forcing}}$)

Finally, we evaluate the local forcing term due to surface heat and geothermal fluxes.

Surface heat flux

For the surface contribution, there are two relevant model diagnostics: - `TFLUX`: total heat flux (match heat-content variations) (W/m^2) - `oceQsw`: net Short-Wave radiation (+=down) (W/m^2)

Defining terms needed for evaluating surface heat forcing

[47]:
```
Z = ecco_grid.Z.load()
RF = np.concatenate([ecco_grid.Zp1.values[:-1],[np.nan]])
```

Note: Z and Zp1 are used in deriving surface heat penetration. MATLAB code uses RF from mygrid structure.

[48]:
```
q1 = R*np.exp(1.0/zeta1*RF[:-1]) + (1.0-R)*np.exp(1.0/zeta2*RF[:-1])
q2 = R*np.exp(1.0/zeta1*RF[1:]) + (1.0-R)*np.exp(1.0/zeta2*RF[1:])
```

[49]:
```
# Correction for the 200m cutoff
zCut = np.where(Z < -200)[0][0]
q1[zCut:] = 0
q2[zCut-1:] = 0
```

[50]:
```
# Save q1 and q2 as xarray data arrays
q1 = xr.DataArray(q1,coords=[Z.k],dims=['k'])
q2 = xr.DataArray(q2,coords=[Z.k],dims=['k'])
```

Compute vertically penetrating flux

Given the penetrating nature of the shortwave term, to properly evaluate the local forcing term, oceQsw must be removed from TFLUX (which contains the net latent, sensible, longwave, and shortwave contributions) and redistributed vertically.

[51]:
```
## Land masks
# Make copy of hFacC
mskC = ecco_grid.hFacC.copy(deep=True).load()

# Change all fractions (ocean) to 1. land = 0
mskC.values[mskC.values>0] = 1
```

[52]:
```
# Shortwave flux below the surface (W/m^2)
forcH_subsurf = ((q1*(mskC==1)-q2*(mskC.shift(k=-1)==1))*ds.oceQsw).transpose('time',
↪'tile','k','j','i')
```

[53]:
```
# Surface heat flux (W/m^2)
forcH_surf = ((ds.TFLUX - (1-(q1[0]-q2[0]))*ds.oceQsw)\
              *mskC[0]).transpose('time','tile','j','i').assign_coords(k=0).expand_dims(
↪'k')
```

[54]:
```
# Full-depth sea surface forcing (W/m^2)
forcH = xr.concat([forcH_surf,forcH_subsurf[:,:,1:]], dim='k').transpose('time','tile','k
↪','j','i')
```

Geothermal flux

The geothermal flux contribution is not accounted for in any of the standard model diagnostics provided as output. Rather, this term, which is time invariant, is provided in the input file `geothermalFlux.bin` and can be downloaded from the PO.DAAC drive (https://ecco.jpl.nasa.gov/drive/files/Version4/Release3/input_init/geothermalFlux.bin). > **Note**: Here, `geothermalFlux.bin` has been placed in `base_dir`.

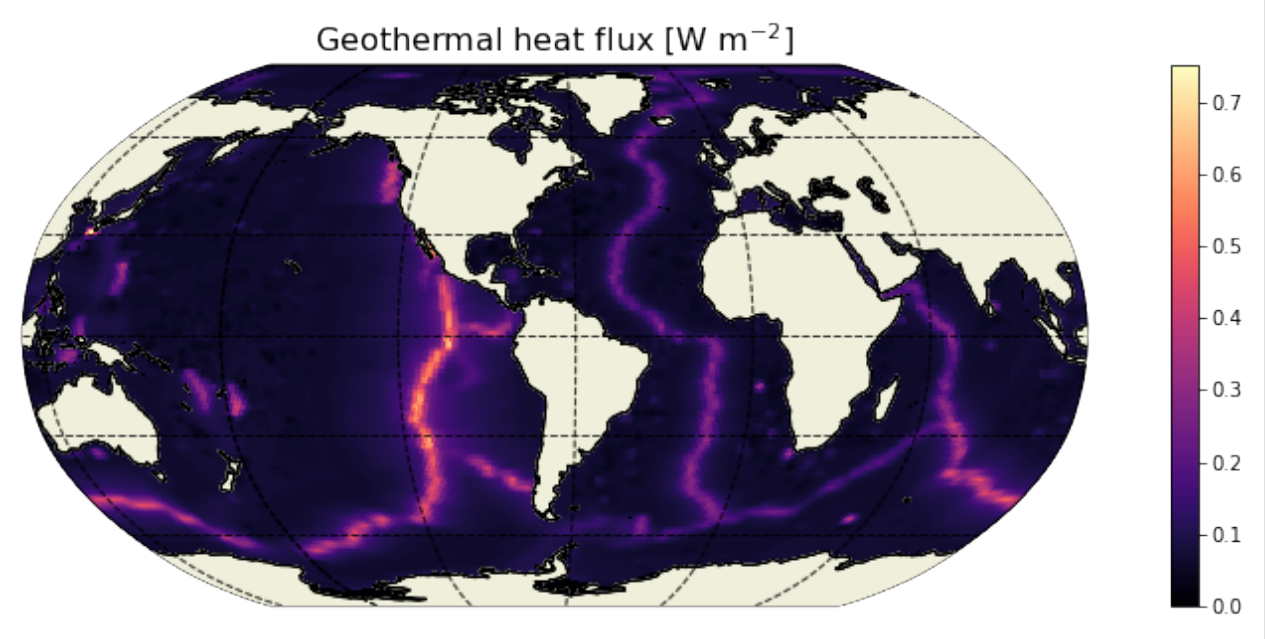
[55]:
```
# Load the geothermal heat flux using the routine 'read_llc_to_tiles'
geoflx = ecco.read_llc_to_tiles(base_dir, 'geothermalFlux.bin')
```

```
load_binary_array: loading file /work/noaa/gfdlscr/jtesdal/ECCOv4-release/geothermalFlux.
↪bin
load_binary_array: data array shape  (1170, 90)
load_binary_array: data array type  >f4
llc_compact_to_faces: dims, llc  (1170, 90) 90
llc_compact_to_faces: data_compact array type  >f4
llc_faces_to_tiles: data_tiles shape  (13, 90, 90)
llc_faces_to_tiles: data_tiles dtype  >f4
```

The geothermal flux dataset needs to be saved as an xarray data array with the same format as the model output.

[56]:
```
# Convert numpy array to an xarray DataArray with matching dimensions as the monthly␣
↪mean fields
geoflx_llc = xr.DataArray(geoflx,coords={'tile': ecco_monthly_mean.tile.values,
                                         'j': ecco_monthly_mean.j.values,
                                         'i': ecco_monthly_mean.i.values},dims=['tile','j
↪','i'])
```

[57]:
```
plt.figure(figsize=(15,5));

ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, geoflx_llc,show_colorbar=True,
↪cmap='magma',
                              user_lon_0=-67, dx=0.2, dy=0.2)
plt.title(r'Geothermal heat flux [W m$^{-2}$]', fontsize=16)
plt.show()
```



Geothermal flux needs to be a three dimensional field since the sources are distributed along the ocean floor at various depths. This requires a three dimensional mask.

[58]:
```
# Create 3d bathymetry mask
mskC_shifted = mskC.shift(k=-1)

mskC_shifted.values[-1,:,:,:] = 0
mskb = mskC - mskC_shifted

# Create 3d field of geothermal heat flux
geoflx3d = geoflx_llc * mskb.transpose('k','tile','j','i')
GEOFLX = geoflx3d.transpose('k','tile','j','i')
GEOFLX.attrs = {'standard_name': 'GEOFLX','long_name': 'Geothermal heat flux','units':
↪'W/m^2'}
```

Total forcing ($G^{\theta}_{\text{forcing}}$)

[59]:
```
# Add geothermal heat flux to forcing field and convert from W/m^2 to degC/s
G_forcing = ((forcH + GEOFLX)/(rhoconst*c_p))/(ecco_grid.hFacC*ecco_grid.drF)
```

Plot the time-mean $G^{\theta}_{\text{forcing}}$

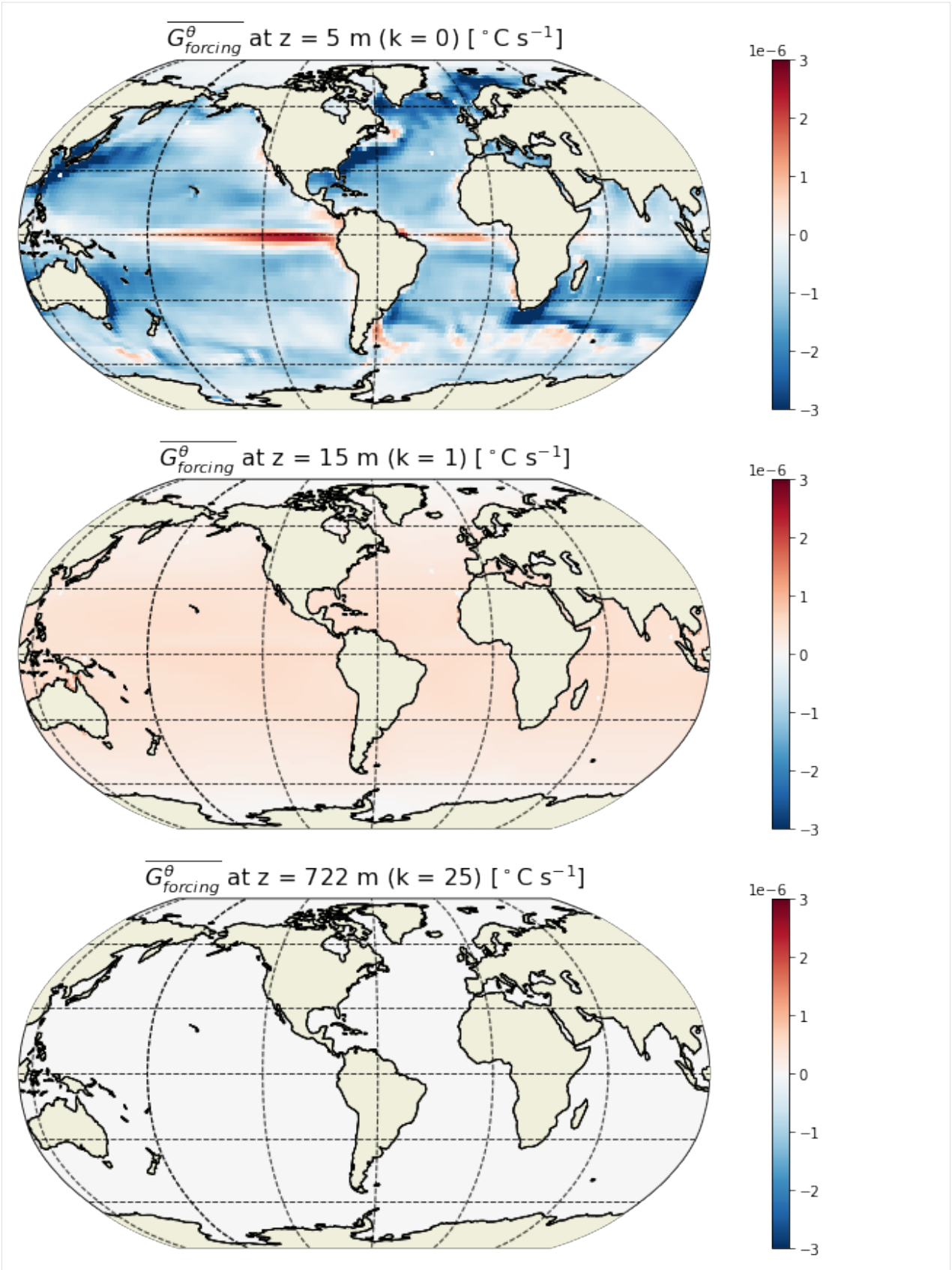
[60]:
```
G_forcing_mean = (G_forcing*month_length_weights).sum('time')
```

[61]:
```
plt.figure(figsize=(15,15))

for idx, k in enumerate([0,1,25]):
    p = ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, G_forcing_mean[:,k],
↪show_colorbar=True,
```

(continues on next page)

(continued from previous page)

```
                                        cmin=-3e-6, cmax=3e-6, cmap='RdBu_r', user_lon_0=-
↪67, dx=2, dy=2,
                                        subplot_grid=[3,1,idx+1]);
    p[1].set_title(r'$\overline{G^\theta_{forcing}}$ at z = %i m (k = %i) [$^\circ$C s$^
↪{-1}$]'\
                   %(np.round(-ecco_grid.Z[k].values),k), fontsize=16)
```


$\overline{G^{\theta}_{forcing}}$ at z = 5 m (k = 0) [$^{\circ}$C s^{-1}]
1e−6
3
2
1
0
−1
−2
−3
$\overline{G^{\theta}_{forcing}}$ at z = 15 m (k = 1) [$^{\circ}$C s^{-1}]
1e−6
3
2
1
0
−1
−2
−3
$\overline{G^{\theta}_{forcing}}$ at z = 722 m (k = 25) [$^{\circ}$C s^{-1}]
1e−6
3
2
1
0
−1
−2
−3

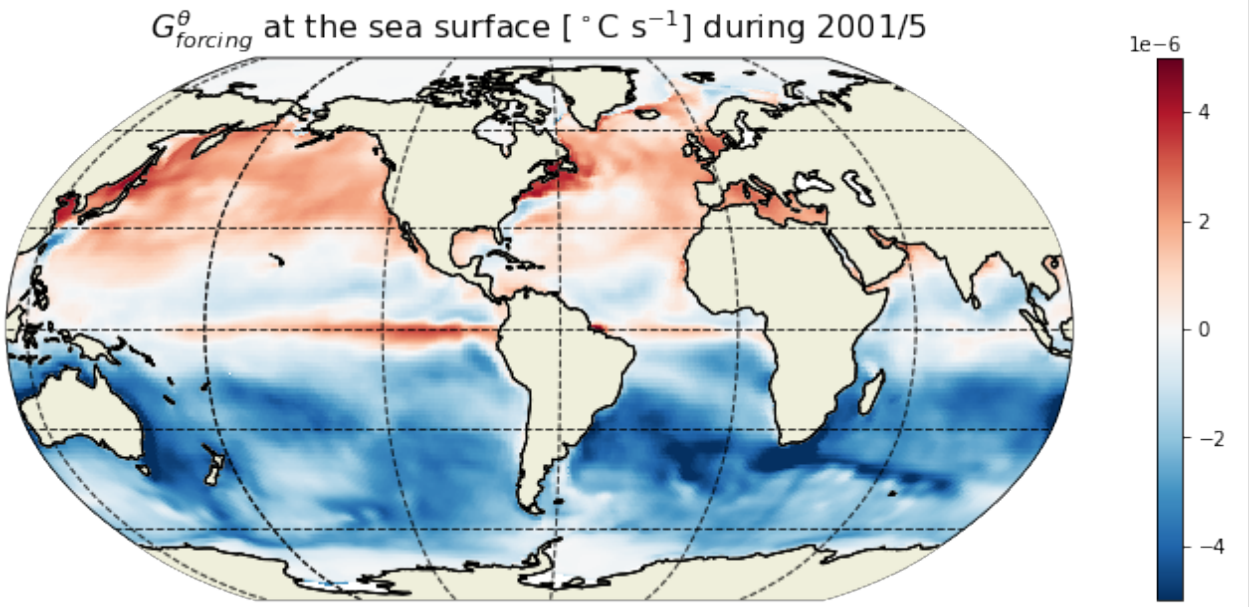
$\overline{G^{\theta}_{forcing}}$ is focused at the sea surface and much smaller (essentially zero) at depth. $\overline{G^{\theta}_{forcing}}$ is negative for most of the ocean (away from the equator). The spatial pattern in the surface forcing is the same as for diffusion but with opposite sign (see maps for $\overline{G^{\theta}_{diffusion}}$ above). This makes sense as forcing is to a large extent balanced by diffusion within the mixed layer.

Example $G^{\theta}_{\text{forcing}}$ field at a particular time

[62]:
```
tmp = ecco.extract_yyyy_mm_dd_hh_mm_ss_from_datetime64(G_forcing.time[100].values)
print(tmp)
```

```
(2001, 5, 16, 12, 0, 0)
```

[63]:
```
plt.figure(figsize=(15,5));

ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, G_forcing.isel(time=100)[:,0],
↪show_colorbar=True,
                              cmin=-5e-6, cmax=5e-6, cmap='RdBu_r', user_lon_0=-67, dx=0.
↪2, dy=0.2)
plt.title(r'$G^\theta_{forcing}$ at the sea surface [$^\circ$C s$^{-1}$] during ' +
          str(tmp[0]) +'/' + str(tmp[1]), fontsize=16)
plt.show()
```



1.21.9 Save to dataset

Now that we have all the terms evaluated, let's save them to a dataset. Here are two examples: - Zarr is a new format that is used for cloud storage. - Netcdf is the more traditional format that most people are familiar with.

Add all variables to a new dataset

[65]:
```
varnames = ['G_total','G_advection','G_diffusion','G_forcing']

ds = xr.Dataset(data_vars={})
for varname in varnames:
    ds[varname] = globals()[varname].chunk(chunks={'time':1,'tile':13,'k':50,'j':90,'i':
↪90})
```

[66]:
```
# Add surface forcing (degC/s)
ds['Qnet'] = ((forcH /(rhoconst*c_p))\
              /(ecco_grid.hFacC*ecco_grid.drF)).chunk(chunks={'time':1,'tile':13,'k':50,
↪'j':90,'i':90})
```

[67]:
```
# Add shortwave penetrative flux (degC/s)
#Since we only are interested in the subsurface heat flux we need to zero out the top␣
↪cell
SWpen = ((forcH_subsurf /(rhoconst*c_p))/(ecco_grid.hFacC*ecco_grid.drF)).where(forcH_
↪subsurf.k>0).fillna(0.)
ds['SWpen'] = SWpen.where(ecco_grid.hFacC>0).chunk(chunks={'time':1,'tile':13,'k':50,'j':
↪90,'i':90})
```

Note: `Qnet` and `SWpen` are included in `G_forcing` and are not necessary to close the heat budget.

[68]:
```
ds.time.encoding = {}
ds = ds.reset_coords(drop=True)
```

Save to zarr

[69]:
```
from dask.diagnostics import ProgressBar
```

[70]:
```
with ProgressBar():
    ds.to_zarr(base_dir + '/eccov4r3_budg_heat')
```

```
[########################################] | 100% Completed |  2min 40.7s
```

Save to netcdf

[70]:
```
with ProgressBar():
    ds.to_netcdf(base_dir + '/eccov4r3_budg_heat.nc', format='NETCDF4')
```

```
[########################################] | 100% Completed | 14min 17.5s
```

1.21.10 Load budget variables from file

After having saved the budget terms to file, we can load the dataset like this

[64]:
```
# Load terms from zarr dataset
G_total = xr.open_zarr(base_dir + '/eccov4r3_budg_heat').G_total
G_advection = xr.open_zarr(base_dir + '/eccov4r3_budg_heat').G_advection
G_diffusion = xr.open_zarr(base_dir + '/eccov4r3_budg_heat').G_diffusion
G_forcing = xr.open_zarr(base_dir + '/eccov4r3_budg_heat').G_forcing
Qnet = xr.open_zarr(base_dir + '/eccov4r3_budg_heat').Qnet
SWpen = xr.open_zarr(base_dir + '/eccov4r3_budg_heat').SWpen
```

Or if you saved it as a netcdf file:

```
# Load terms from netcdf file
G_total_tendency = xr.open_dataset(base_dir + '/eccov4r3_budg_heat.nc').G_total_tendency
G_advection = xr.open_dataset(base_dir + '/eccov4r3_budg_heat.nc').G_advection
G_diffusion = xr.open_dataset(base_dir + '/eccov4r3_budg_heat.nc').G_diffusion
G_forcing = xr.open_dataset(base_dir + '/eccov4r3_budg_heat.nc').G_forcing
Qnet = xr.open_dataset(base_dir + '/eccov4r3_budg_heat.nc').Qnet
```

1.21.11 Comparison between LHS and RHS of the budget equation

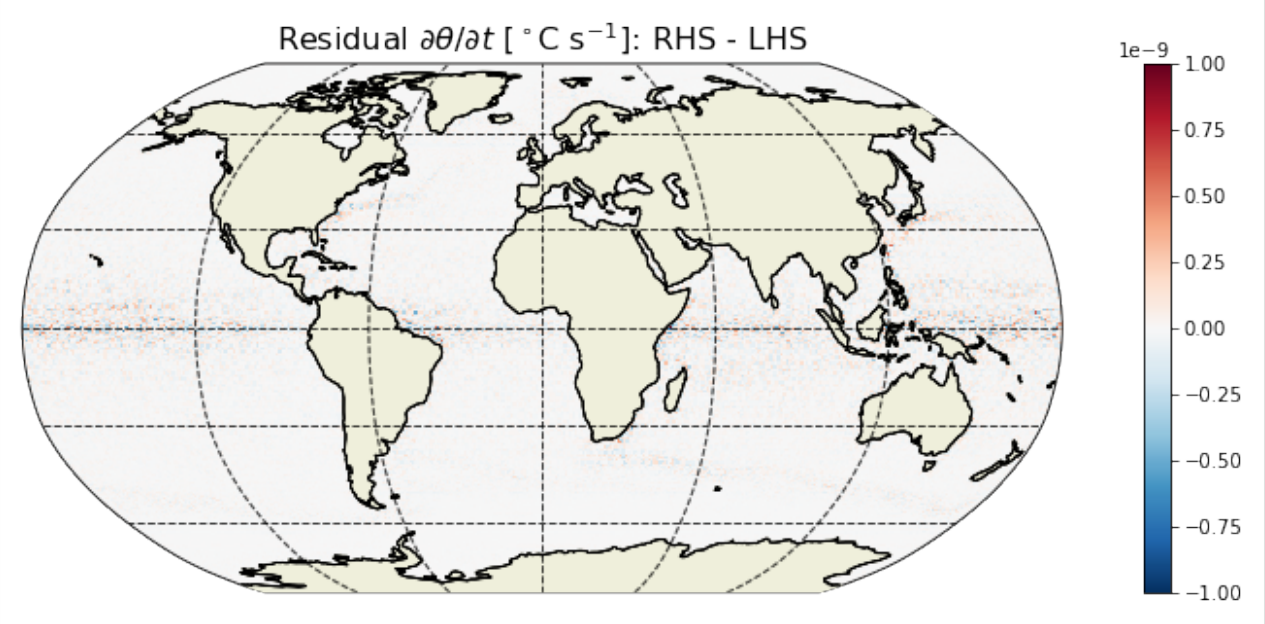
[65]:
```
# Total convergence
ConvH = G_advection + G_diffusion
```

[66]:
```
# Sum of terms in RHS of equation
rhs = ConvH + G_forcing
```

Map of residuals

[67]:
```
res = (rhs-G_total).sum(dim='k').sum(dim='time').compute()
```

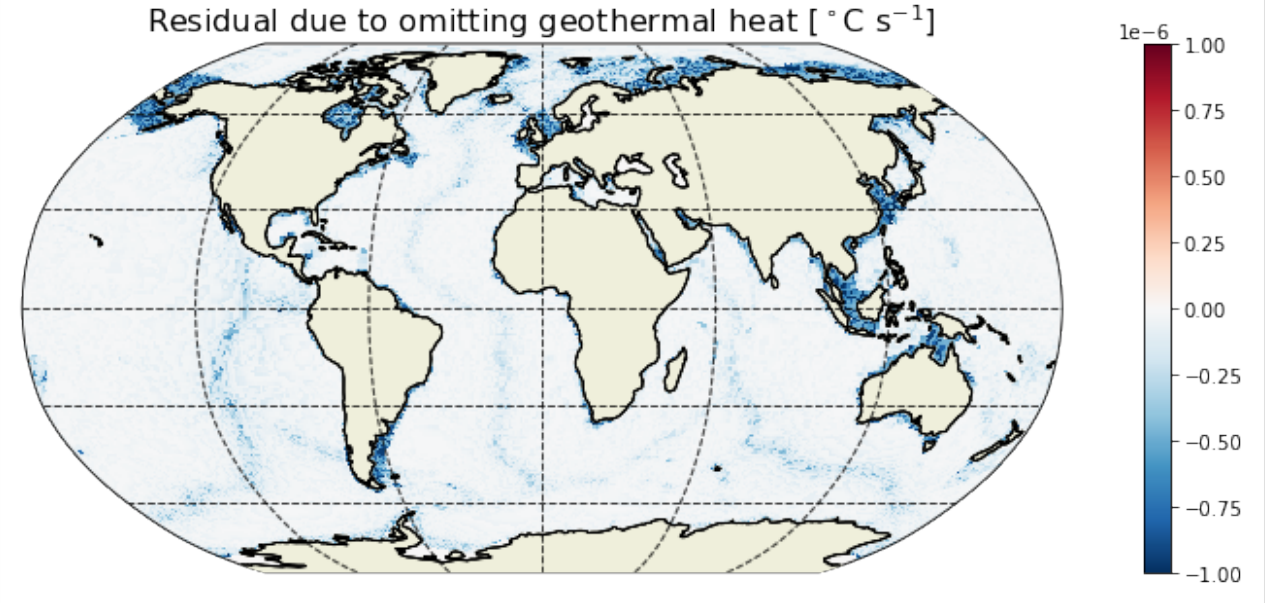
[68]:
```
plt.figure(figsize=(15,5))
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, res,
                              cmin=-1e-9, cmax=1e-9, show_colorbar=True, cmap='RdBu_r',
↪dx=0.2, dy=0.2)
plt.title(r'Residual $\partial \theta / \partial t$ [$^\circ$C s$^{-1}$]: RHS - LHS',␣
↪fontsize=16)
plt.show()
```



The residual (summed over depth and time) is essentially zero everywhere. What if we omit the geothermal heat flux?

[69]:
```
# Residual when omitting geothermal heat flux
res_geo = (ConvH + Qnet - G_total).sum(dim='k').sum(dim='time').compute()
```

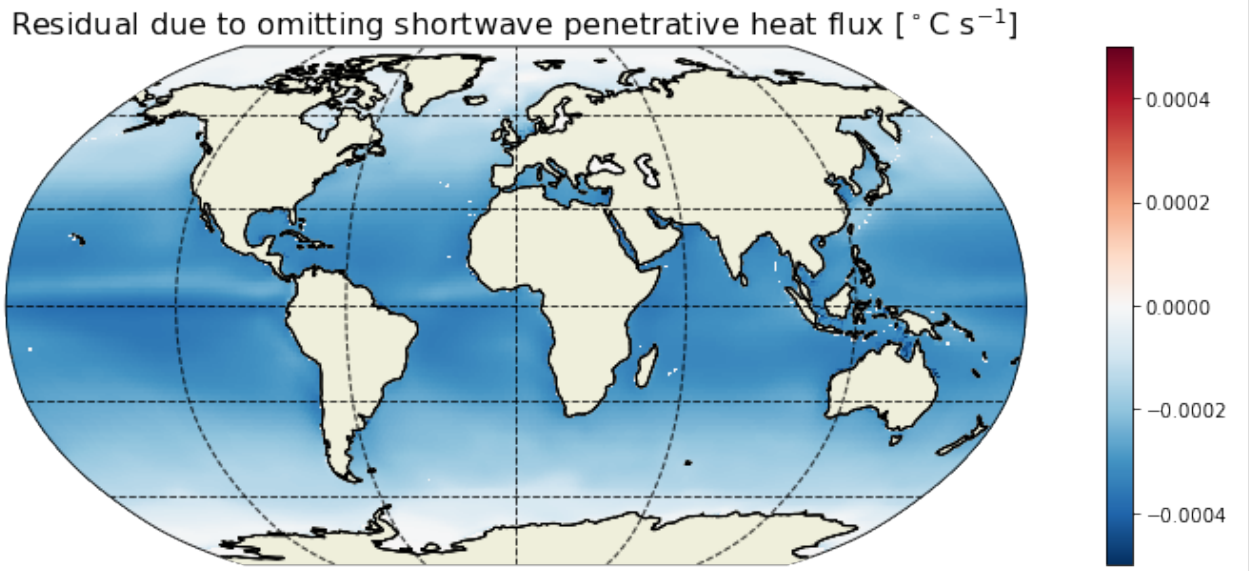
[70]:
```
plt.figure(figsize=(15,5))
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, res_geo,
                              cmin=-1e-6, cmax=1e-6, show_colorbar=True, cmap='RdBu_r',␣
↪dx=0.2, dy=0.2)
plt.title(r'Residual due to omitting geothermal heat [$^\circ$C s$^{-1}$] ', fontsize=16)
plt.show()
```



We see that the contribution from geothermal flux in the heat budget is well above the residual (by *three orders of magnitude*).

[71]:
```
# Residual when omitting shortwave penetrative heat flux
res_sw = (rhs-SWpen-G_total).sum(dim='k').sum(dim='time').compute()
```

[72]:
```
plt.figure(figsize=(15,5))
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, res_sw,
                              cmin=-5e-4, cmax=5e-4, show_colorbar=True, cmap='RdBu_r',␣
↪dx=0.2, dy=0.2)
plt.title(r'Residual due to omitting shortwave penetrative heat flux [$^\circ$C s$^{-1}
↪$] ', fontsize=16)
plt.show()
```



In terms of subsurface heat fluxes, shortwave penetration represents a much larger heat flux compared to geothermal heat flux (by around *three orders of magnitude*).

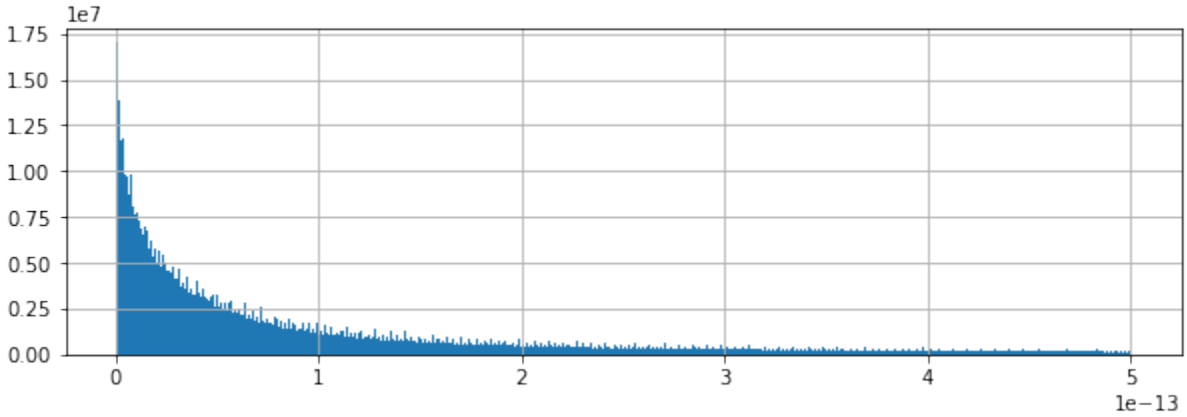
Histogram of residuals

We can look at the distribution of residuals to get a little more confidence.

[76]:
```
from xhistogram.xarray import histogram
```

[74]:
```
tmp = np.abs(rhs-G_total).values.ravel()
```

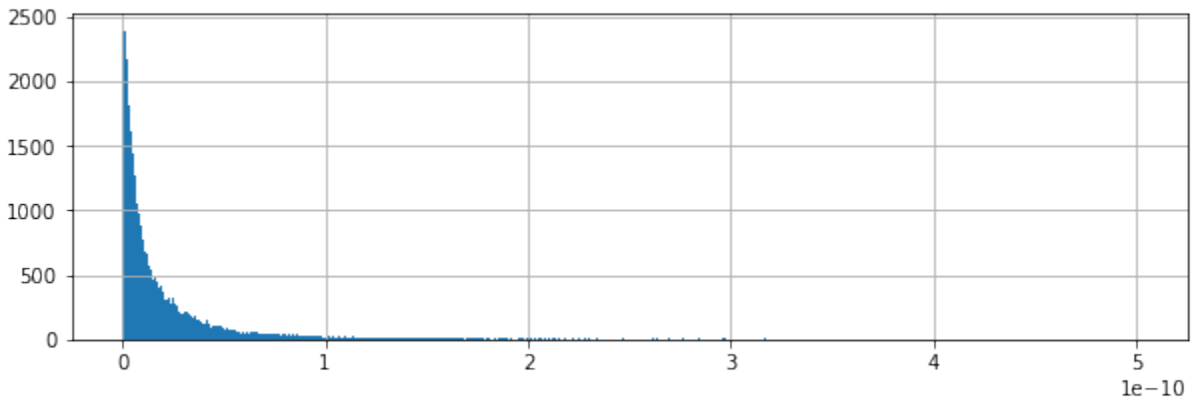
[87]:
```
plt.figure(figsize=(10,3));

plt.hist(tmp[np.nonzero(tmp > 0)],np.linspace(0, .5e-12,501));
plt.grid()
```



Almost all residuals < 10^{-13} °C s^{-1}.

[88]:
```
tmp = np.abs(res).values.ravel()
```

[89]:
```
plt.figure(figsize=(10,3));

plt.hist(tmp[np.nonzero(tmp > 0)],np.linspace(0, .5e-9, 1000));
plt.grid()
```



Summing residuals vertically and temporally yields < 10^{-10} °C s^{-1} for most grid points.

1.21.12 Heat budget closure through time

Global average budget closure

Another way of demonstrating heat budget closure is to show the global spatially-averaged `THETA` tendency terms

[91]:
```
# Volume (m^3)
vol = (ecco_grid.rA*ecco_grid.drF*ecco_grid.hFacC).transpose('tile','k','j','i')

# Take volume-weighted mean of these terms
tmp_a=(G_total*vol).sum(dim=('k','i','j','tile'))/vol.sum()
tmp_b=(G_advection*vol).sum(dim=('k','i','j','tile'))/vol.sum()
```

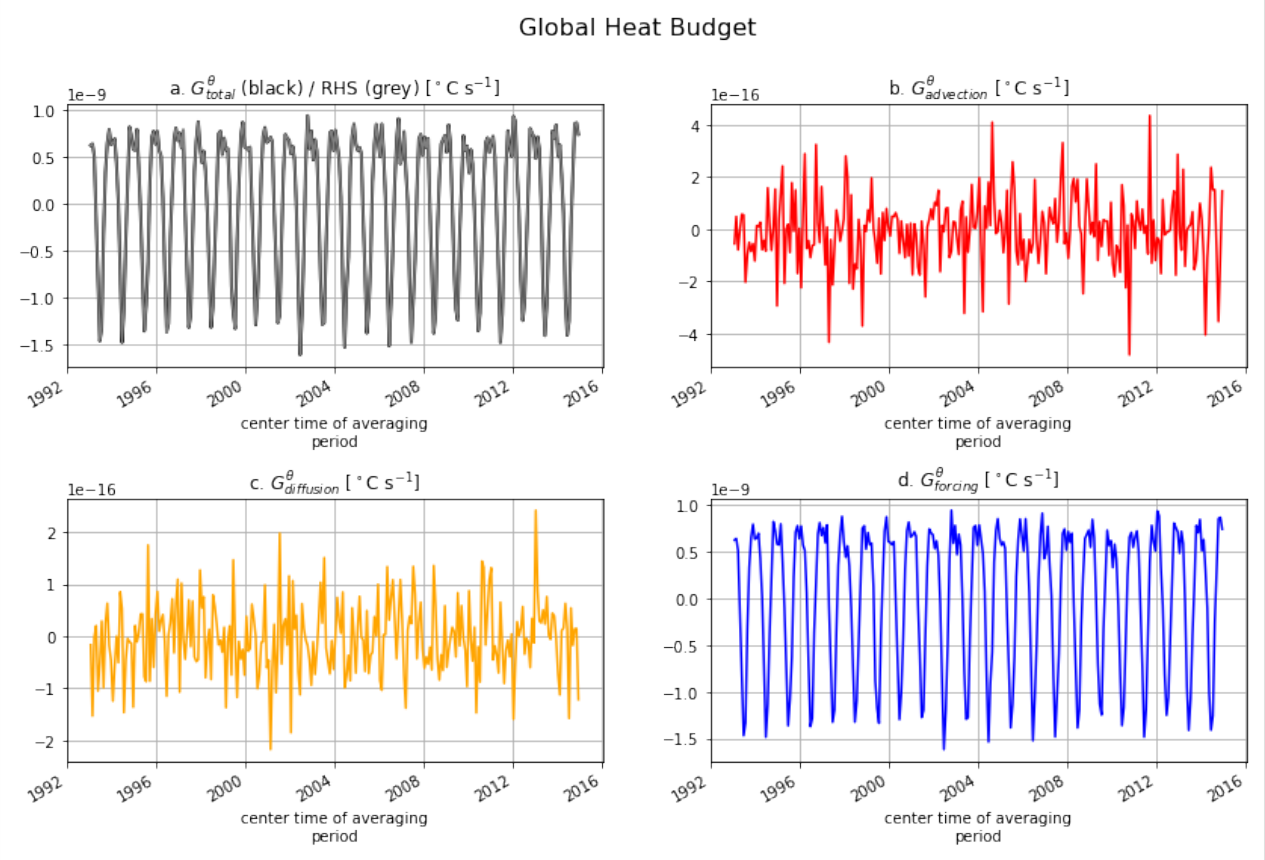
(continues on next page)

(continued from previous page)

```
tmp_c=(G_diffusion*vol).sum(dim=('k','i','j','tile'))/vol.sum()
tmp_d=(G_forcing*vol).sum(dim=('k','i','j','tile'))/vol.sum()
tmp_e=(rhs*vol).sum(dim=('k','i','j','tile'))/vol.sum()

# Result is five time series
tmp_a.dims
```

[91]:
```
('time',)
```

[92]:
```
fig, axs = plt.subplots(2, 2, figsize=(14,8))

plt.sca(axs[0,0])
tmp_a.plot(color='k',lw=2)
tmp_e.plot(color='grey')
axs[0,0].set_title(r'a. $G^\theta_{total}$ (black) / RHS (grey) [$^\circ$C s$^{-1}$]',␣
↪fontsize=12)
plt.grid()

plt.sca(axs[0,1])
tmp_b.plot(color='r')
axs[0,1].set_title(r'b. $G^\theta_{advection}$ [$^\circ$C s$^{-1}$]', fontsize=12)
plt.grid()

plt.sca(axs[1,0])
tmp_c.plot(color='orange')
axs[1,0].set_title(r'c. $G^\theta_{diffusion}$ [$^\circ$C s$^{-1}$]', fontsize=12)
plt.grid()

plt.sca(axs[1,1])
tmp_d.plot(color='b')
axs[1,1].set_title(r'd. $G^\theta_{forcing}$ [$^\circ$C s$^{-1}$]', fontsize=12)
plt.grid()
plt.subplots_adjust(hspace = .5, wspace=.2)
plt.suptitle('Global Heat Budget', fontsize=16);
```



When averaged over the entire ocean the ocean heat transport terms ($G^{\theta}_{\text{advection}}$ and $G^{\theta}_{\text{diffusion}}$) have no net impact on $G^{\theta}_{\text{total}}$ (i.e., $\partial\theta/\partial t$). This makes sence because $G^{\theta}_{\text{advection}}$ and $G^{\theta}_{\text{diffusion}}$ can only redistributes heat. Globally, θ can only change via $G^{\theta}_{\text{forcing}}$.

Local heat budget closure

Locally we expect that heat divergence can impact θ. This is demonstrated for a single grid point.

[93]:
```
# Pick any set of indices (tile, k, j, i) corresponding to an ocean grid point
t,k,j,i = (6,10,40,29)
print(t,k,j,i)
```

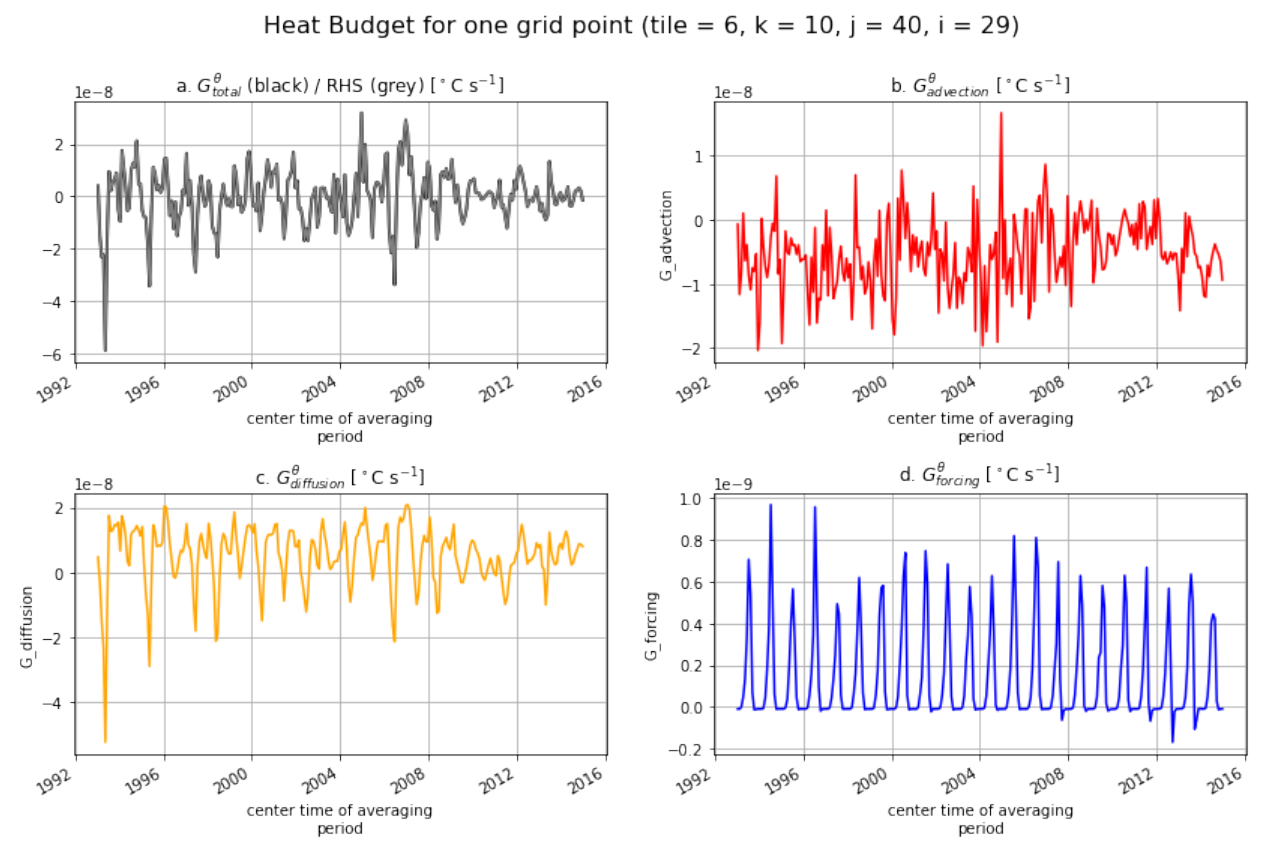
```
6 10 40 29
```

[94]:
```
tmp_a = G_total.isel(tile=t,k=k,j=j,i=i)
tmp_b = G_advection.isel(tile=t,k=k,j=j,i=i)
tmp_c = G_diffusion.isel(tile=t,k=k,j=j,i=i)
tmp_d = G_forcing.isel(tile=t,k=k,j=j,i=i)
tmp_e = rhs.isel(tile=t,k=k,j=j,i=i)

fig, axs = plt.subplots(2, 2, figsize=(14,8))

plt.sca(axs[0,0])
tmp_a.plot(color='k',lw=2)
tmp_e.plot(color='grey')
```

(continues on next page)

(continued from previous page)

```
axs[0,0].set_title(r'a. $G^\theta_{total}$ (black) / RHS (grey) [$^\circ$C s$^{-1}$]',␣
↪fontsize=12)
plt.grid()

plt.sca(axs[0,1])
tmp_b.plot(color='r')
axs[0,1].set_title(r'b. $G^\theta_{advection}$ [$^\circ$C s$^{-1}$]', fontsize=12)
plt.grid()

plt.sca(axs[1,0])
tmp_c.plot(color='orange')
axs[1,0].set_title(r'c. $G^\theta_{diffusion}$ [$^\circ$C s$^{-1}$]', fontsize=12)
plt.grid()

plt.sca(axs[1,1])
tmp_d.plot(color='b')
axs[1,1].set_title(r'd. $G^\theta_{forcing}$ [$^\circ$C s$^{-1}$]', fontsize=12)
plt.grid()
plt.subplots_adjust(hspace = .5, wspace=.2)
plt.suptitle('Heat Budget for one grid point (tile = %i, k = %i, j = %i, i = %i)'%(t,k,j,
↪i), fontsize=16);
```

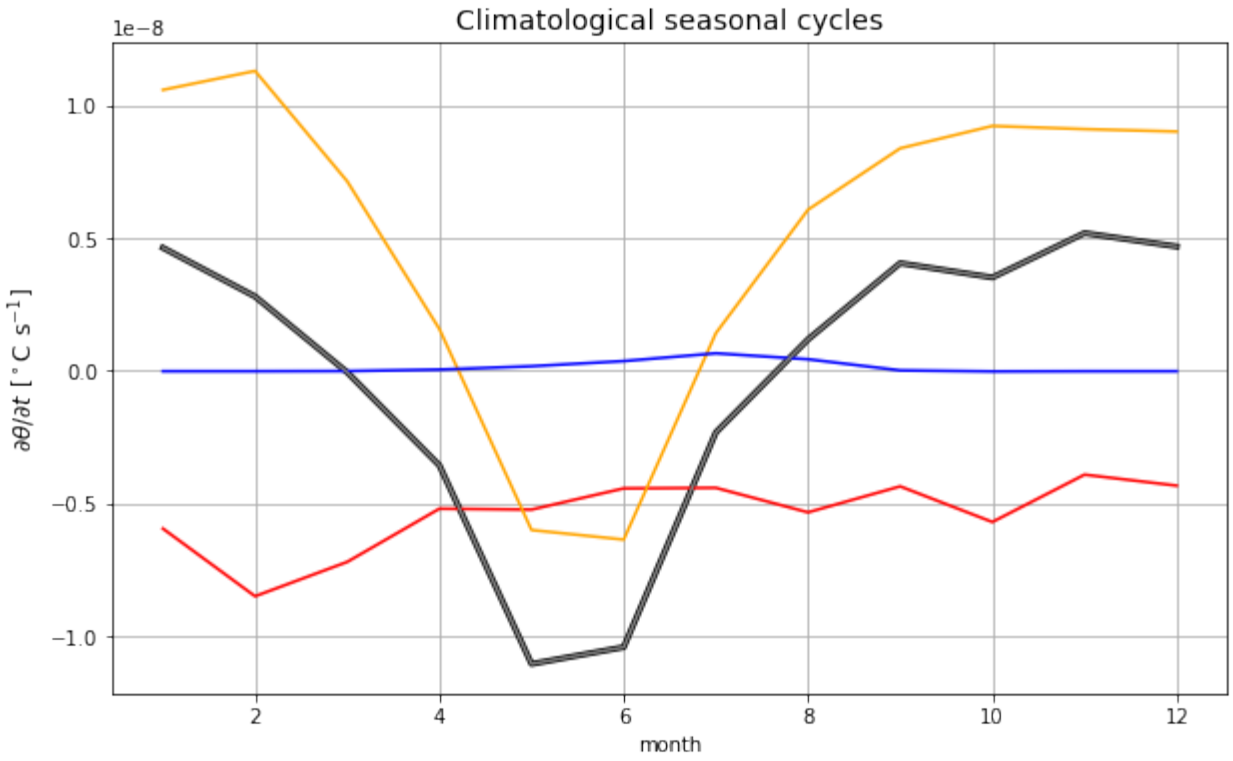


Indeed, the heat divergence terms do contribute to θ variations at a single point. Local heat budget closure is also confirmed at this grid point as we see that the sum of terms on the RHS (grey line) equals the LHS (black line).

For the Arctic grid point, there is a clear seasonal cycles in both G^θ_{total}, $G^\theta_{\text{diffusion}}$ and $G^\theta_{\text{forcing}}$. The seasonal cycle in

$G^{\theta}_{\text{forcing}}$ seems to be the reverse of $G^{\theta}_{\text{total}}$ and $G^{\theta}_{\text{diffusion}}$.

[95]:
```
plt.figure(figsize=(10,6));
tmp_a.groupby('time.month').mean('time').plot(color='k',lw=3)
tmp_b.groupby('time.month').mean('time').plot(color='r')
tmp_c.groupby('time.month').mean('time').plot(color='orange')
tmp_d.groupby('time.month').mean('time').plot(color='b')
tmp_e.groupby('time.month').mean('time').plot(color='grey')
plt.ylabel(r'$\partial\theta$/$\partial t$ [$^\circ$C s$^{-1}$]', fontsize=12)
plt.grid()
plt.title('Climatological seasonal cycles', fontsize=14)
plt.show()
```



The mean seasonal cycle of the total is driven by seasonality in diffusion. However, this is likely depth-dependent. How does the balance look across the upper 200 meter at that location?

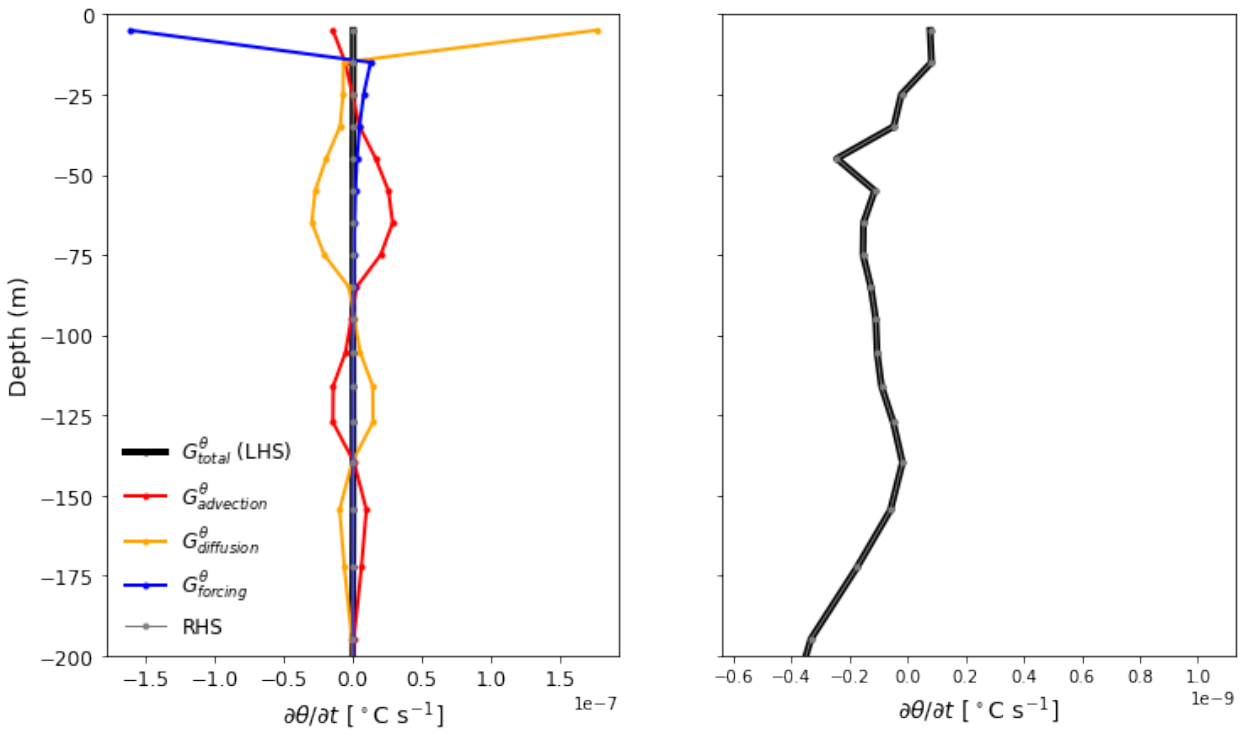
1.21.13 Time-mean vertical profiles

[97]:
```
fig = plt.subplots(1, 2, sharey=True, figsize=(12,7))

plt.subplot(1, 2, 1)
plt.plot(G_total.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z,
         lw=4, color='black', marker='.', label=r'$G^\theta_{total}$ (LHS)')

plt.plot(G_advection.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z,
         lw=2, color='red', marker='.', label=r'$G^\theta_{advection}$')
```

(continues on next page)

(continued from previous page)

```
plt.plot(G_diffusion.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z,
         lw=2, color='orange', marker='.', label=r'$G^\theta_{diffusion}$')

plt.plot(G_forcing.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z,
         lw=2, color='blue', marker='.', label=r'$G^\theta_{forcing}$')
plt.plot(rhs.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z, lw=1, color='grey', marker=
↪'.', label='RHS')
plt.xlabel(r'$\partial\theta$/$\partial t$ [$^\circ$C s$^{-1}$]', fontsize=14)
plt.ylim([-200,0])
plt.ylabel('Depth (m)', fontsize=14)
plt.gca().tick_params(axis='both', which='major', labelsize=12)
plt.legend(loc='lower left', frameon=False, fontsize=12)

plt.subplot(1, 2, 2)
plt.plot(G_total.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z,
         lw=4, color='black', marker='.', label=r'$G^\theta_{total}$ (LHS)')
plt.plot(rhs.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z, lw=1, color='grey', marker=
↪'.', label='RHS')
plt.setp(plt.gca(), 'yticklabels',[])
plt.xlabel(r'$\partial\theta$/$\partial t$ [$^\circ$C s$^{-1}$]', fontsize=14)
plt.ylim([-200,0])
plt.show()
```



Balance between surface forcing and diffusion in the top layers. Balance between advection and diffusion at depth.

[ ]:

1.22 Salt, Salinity and Freshwater Budgets

Contributors: Jan-Erik Tesdal, Ryan Abernathey and Ian Fenty

A major part of this tutorial is based on "*A Note on Practical Evaluation of Budgets in ECCO Version 4 Release 3"* by Christopher G. Piecuch (https://ecco.jpl.nasa.gov/drive/files/Version4/Release3/doc/v4r3_budgets_howto.pdf). Calculation steps and Python code presented here are converted from the MATLAB code presented in the above reference.

1.22.1 Objectives

This tutorial will go over three main budgets which are all related:

1. Salt budget
2. Salinty budget
3. Freshwater budget

We will describe the governing equations for the conservation for both salt, salinity and freshwater content and discuss the subtle differences one needs to be aware of when closing budgets of salt and freshwater content (extensive quantities) versus the budget of salinity (an intensive quantity) in ECCOv4.

1.22.2 Introduction

The general form for the salt/salinity budget can be formulated in the same way as with the heat budget, where instead of potential temperature (θ), the budget is described with salinity (S).

$$\frac{\partial S}{\partial t} = -\nabla \cdot (S\mathbf{u}) - \nabla \cdot \mathbf{F}^S_{\text{diff}} + F^S_{\text{forc}} \tag{1.12}$$

The total tendency ($\frac{\partial S}{\partial t}$) is equal to advective convergence ($-\nabla \cdot (S\mathbf{u})$), diffusive flux convergence ($-\nabla \cdot \mathbf{F}^S_{\text{diff}}$) and a forcing term F^S_{forc}.

In the case of ECCOv4, salt is strictly a conserved mass and can be described as

$$\underbrace{\frac{\partial (s^* S)}{\partial t}}_{G^{Slt}_{\text{total}}} = \underbrace{-\nabla_{z^*} \cdot (s^* S\, \mathbf{v}_{res}) - \frac{\partial (S\, w_{res})}{\partial z^*}}_{G^{Slt}_{\text{advection}}} \underbrace{-s^* (\nabla \cdot \mathbf{F}^S_{\text{diff}})}_{G^{Slt}_{\text{diffusion}}} + \underbrace{s^* F^S_{\text{forc}}}_{G^{Slt}_{\text{forcing}}} \tag{1.13}$$

The change in salt content over time (G^{Slt}_{total}) is equal to the convergence of the advective flux ($G^{Slt}_{\text{advection}}$) and diffusive flux ($G^{Slt}_{\text{diffusion}}$) plus a forcing term associated with surface salt exchanges (G^{Slt}_{forcing}). As with the heat budget, we present both the horizontal ($\mathbf{v}_{res}$) and vertical (w_{res}) components of the advective term. Again, we have $\mathbf{u}_{res}$ as the "residual mean" velocities, which contain both the resolved (Eulerian) and parameterizing "GM bolus" velocities. Also note the use of the rescaled height coordinate z^* and the scale factor s^* which have been described in the volume and heat budget tutorials.

The salt budget in ECCOv4 only considers the mass of salt in the ocean. Thus, the convergence of freshwater and surface freshwater exchanges are not formulated specifically. An important point here is that, given the nonlinear free surface condition in ECCOv4, budgets for salt content (an extensive quantity) are not the same as budgets for salinity (an intensive quantity). In order to accurately describe variation in salinity, we need to take into account the variation of both salt and volume. Using the product rule, G^{Slt}_{total} (i.e., the left side of the salt budget equation) can be extended as follows

$$\frac{\partial (s^* S)}{\partial t} = s^* \frac{\partial S}{\partial t} + S \frac{\partial s^*}{\partial t} \tag{1.14}$$

When substituting G^{Slt}_{total} with the right hand side of the above equation, we can solve for the salinity tendency ($\frac{\partial S}{\partial t}$):

$$\frac{\partial S}{\partial t} = -\frac{1}{s^*}\left[S\frac{\partial s^*}{\partial t} + \nabla_{z^*}\cdot(s^* S\,\mathbf{v}_{res}) + \frac{\partial(S\,w_{res})}{\partial z^*}\right] - \nabla\cdot\mathbf{F}^S_{\text{diff}} + F^S_{\text{forc}}$$

Since $s^* = 1 + \frac{\eta}{H}$ we can define the temporal change in s^* as

$$\frac{\partial s^*}{\partial t} = \frac{1}{H}\frac{\partial \eta}{\partial t} \tag{1.15}$$

This constitutes the conservation of volume in ECCOv4, which can be formulated as

$$\frac{1}{H}\frac{\partial \eta}{\partial t} = -\nabla_{z^*}\cdot(s^*\mathbf{v}) - \frac{\partial w}{\partial z^*} + \mathcal{F} \tag{1.16}$$

You can read more about volume conservation and the z^* coordinate system in another tutorial. $\mathcal{F}$ denotes the volumetric surface fluxes and can be decomposed into net atmospheric freshwater fluxes (i.e., precipitation minus evaporation, $P-E$), continental runoff (R) and exchanges due to sea ice melting/formation (I). Here $\mathbf{v} = (u, v)$ and w are the resolved horizontal and vertical velocities, respectively.

Thus, the conservation of salinity in ECCOv4 can be described as

$$\underbrace{\frac{\partial S}{\partial t}}_{G^{Sln}_{\text{total}}} = \underbrace{\frac{1}{s^*}\left[S\left(\nabla_{z^*}\cdot(s^*\mathbf{v}) + \frac{\partial w}{\partial z^*}\right) - \nabla_{z^*}\cdot(s^* S\,\mathbf{v}_{res}) - \frac{\partial(S\,w_{res})}{\partial z^*}\right]}_{G^{Sln}_{\text{advection}}} \underbrace{-\nabla\cdot\mathbf{F}^S_{\text{diff}}}_{G^{Sln}_{\text{diffusion}}} + \underbrace{F^S_{\text{forc}} - S\,\mathcal{F}}_{G^{Sln}_{\text{forcing}}} \tag{1.17}$$

Notice here that, in contrast to the salt budget equation, the salinity equation explicitly includes the surface forcing ($S\,\mathcal{F}$). $\mathcal{F}$ represents surface freshwater exchanges ($P - E + R - I$) and F^S_{forc} represents surface salt fluxes (i.e., addition/removal of salt). Besides the convergence of the advective flux ($\nabla\cdot(S\mathbf{u}_{res})$), the salinity equation also includes the convergence of the volume flux multiplied by the salinity ($S\,(\nabla\cdot\mathbf{u})$), which accounts for the concentration/dilution effect of convergent/divergent volume flux.

The (liquid) freshwater content is defined here as the volume of freshwater (i.e., zero-salinity water) that needs to be added (or subtracted) to account for the deviation between salinity S from a given reference salinity S_{ref}. Thus, within a control volume V the freshwater content is defined as a volume (V_{fw}):

$$V_{fw} = \iiint_V \frac{S_{ref} - S}{S_{ref}} dV \tag{1.18}$$

Similar to the salt and salinity budgets, the total tendency (i.e., change in freshwater content over time) can be expressed as the sum of the tendencies due to advective convergence, diffusive convergence, and forcing:

$$\underbrace{\frac{\partial V_{fw}}{\partial t}}_{G^{fw}_{\text{total}}} = \underbrace{-\nabla\cdot\mathbf{F}^{fw}_{\text{adv}}}_{G^{fw}_{\text{advection}}} \underbrace{-\nabla\cdot\mathbf{F}^{fw}_{\text{diff}}}_{G^{fw}_{\text{diffusion}}} + \underbrace{\mathcal{F}}_{G^{fw}_{\text{forcing}}} \tag{1.19}$$

1.22.3 Prepare environment and load ECCOv4 diagnostic output

Import relevant Python modules

[1]:
```
import numpy as np
import xarray as xr
```

[2]:
```
# Suppress warning messages for a cleaner presentation
import warnings
warnings.filterwarnings('ignore')
```

[3]:
```
## Import the ecco_v4_py library into Python
## =========================================

## -- If ecco_v4_py is not installed in your local Python library,
##    tell Python where to find it.

#import sys
#sys.path.append('/Users/jet/git/ECCOv4-py')

import ecco_v4_py as ecco
```

[4]:
```
# Plotting
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import AxesGrid
from cartopy.mpl.geoaxes import GeoAxes
import cartopy
%matplotlib inline
```

Add relevant constants

[5]:
```
# Seawater density (kg/m^3)
rhoconst = 1029
## needed to convert surface mass fluxes to volume fluxes
```

Load ecco_grid

[6]:
```
## Set top-level file directory for the ECCO NetCDF files
## ======================================================

# Define main directory
base_dir = '/work/noaa/gfdlscr/jtesdal/ECCOv4-release'

# Define a high-level directory for ECCO fields
ECCO_dir = base_dir + '/Release3_alt'
```

Note: Change `base_dir` to your own directory path.

[7]:
```
# Load the model grid
grid_dir = ECCO_dir + '/nctiles_grid/'
ecco_grid = ecco.load_ecco_grid_nc(grid_dir, 'ECCOv4r3_grid.nc')
```

Volume

Calculate the volume of each grid cell. This is used when converting advective and diffusive flux convergences and calculating volume-weighted averages.

[8]:
```
# Volume (m^3)
vol = (ecco_grid.rA*ecco_grid.drF*ecco_grid.hFacC).transpose('tile','k','j','i')
```

Load monthly snapshots

[9]:
```
data_dir= ECCO_dir + '/nctiles_monthly_snapshots'

year_start = 1993
year_end = 2017

# Load one extra year worth of snapshots
ecco_monthly_snaps = ecco.recursive_load_ecco_var_from_years_nc(data_dir, \
                         vars_to_load=['ETAN','SALT'],\
                         years_to_load=range(year_start, year_end+1))

num_months = len(ecco_monthly_snaps.time.values)
# Drop the last 11 months so that we have one snapshot at the beginning and end of each␣
↪month within the
# range 1993/1/1 to 2015/1/1

ecco_monthly_snaps = ecco_monthly_snaps.isel(time=np.arange(0, num_months-11))
```

```
loading files of  ETAN
loading files of  SALT
```

[10]:
```
# Drop superfluous coordinates (We already have them in ecco_grid)
ecco_monthly_snaps = ecco_monthly_snaps.reset_coords(drop=True)
```

Load monthly mean data

[11]:
```
data_dir= ECCO_dir + '/nctiles_monthly'

# Find the record of the last snapshot
## This is used to defined the exact period for monthly mean data
year_end = ecco_monthly_snaps.time.dt.year.values[-1]

ecco_monthly_mean = ecco.recursive_load_ecco_var_from_years_nc(data_dir, \
                        vars_to_load=['SFLUX', 'oceSPtnd', 'ADVx_SLT', 'ADVy_SLT', 'ADVr_
↪SLT',
                                      'DFxE_SLT', 'DFyE_SLT', 'DFrE_SLT', 'DFrI_SLT',
```

(continues on next page)

(continued from previous page)

```
↪'oceFWflx',
                                        'UVELMASS', 'VVELMASS', 'WVELMASS', 'GM_PsiX', 'GM_
↪PsiY',
                                        'ETAN', 'SALT'], years_to_load=range(year_start,␣
↪year_end))
```

```
loading files of  ADVr_SLT
loading files of  ADVx_SLT
loading files of  ADVy_SLT
loading files of  DFrE_SLT
loading files of  DFrI_SLT
loading files of  DFxE_SLT
loading files of  DFyE_SLT
loading files of  ETAN
loading files of  GM_PsiX
loading files of  GM_PsiY
loading files of  SALT
loading files of  SFLUX
loading files of  UVELMASS
loading files of  VVELMASS
loading files of  WVELMASS
loading files of  oceFWflx
loading files of  oceSPtnd
```

[12]:
```
# Drop superfluous coordinates (We already have them in ecco_grid)
ecco_monthly_mean = ecco_monthly_mean.reset_coords(drop=True)
```

Merge dataset of monthly mean and snapshots data

Merge the two datasets to put everything into one single dataset

[13]:
```
ds = xr.merge([ecco_monthly_mean,
               ecco_monthly_snaps.rename({'time':'time_snp','ETAN':'ETAN_snp', 'SALT':
↪'SALT_snp'})])
```

Predefine coordinates for global regridding of the ECCO output (used in `resample_to_latlon`)

[14]:
```
new_grid_delta_lat = 1
new_grid_delta_lon = 1

new_grid_min_lat = -90+new_grid_delta_lat/2
new_grid_max_lat = 90-new_grid_delta_lat/2

new_grid_min_lon = -180+new_grid_delta_lon/2
new_grid_max_lon = 180-new_grid_delta_lon/2
```

Create the xgcm 'grid' object

[15]:
```
# Change time axis of the snapshot variables
ds.time_snp.attrs['c_grid_axis_shift'] = 0.5
```

[16]:
```
grid = ecco.get_llc_grid(ds)
```

Number of seconds in each month

The xgcm `grid` object includes information on the time axis, such that we can use it to get Δt, which is the time span between the beginning and end of each month (in seconds).

[17]:
```
delta_t = grid.diff(ds.time_snp, 'T', boundary='fill', fill_value=np.nan)
```

[18]:
```
# convert to seconds
delta_t = delta_t.astype('f4') / 1e9
```

1.22.4 Evaluating the salt budget

We will evalute each term in the above salt budget

$$G^{Slt}_{\text{total}} = G^{Slt}_{\text{advection}} + G^{Slt}_{\text{diffusion}} + G^{Slt}_{\text{forcing}}$$

The total tendency of salt (G^{Slt}_{total}) is the sum of the salt tendencies from advective convergence ($G^{Slt}_{\text{advection}}$), diffusive heat convergence ($G^{Slt}_{\text{diffusion}}$) and total forcing ($G^{Slt}_{\text{forcing}}$).

We present calculations sequentially for each term starting with G^{Slt}_{total} which will be derived by differencing instantaneous monthly snapshots of `SALT`. The terms on the right hand side of the heat budget are derived from monthly-averaged fields.

Total salt tendency

We calculate the monthly-averaged time tendency of `SALT` by differencing monthly `SALT` snapshots. Remember that we need to include a scale factor due to the nonlinear free surface formulation. Thus, we need to use snapshots of both `ETAN` and `SALT` to evaluate s^*S.

[19]:
```
# Calculate the s*S term
sSALT = ds.SALT_snp*(1+ds.ETAN_snp/ecco_grid.Depth)
```

[20]:
```
# Total tendency (psu/s)
G_total_Slt = grid.diff(sSALT, 'T', boundary='fill', fill_value=0.0)/delta_t
```

The nice thing is that now the time values of G^{Slt}_{total} (`G_total_Slt`) line up with the time values of the time-mean fields (middle of the month)

Advective salt convergence

Horizontal advective salt convergence

The relevant fields from the diagnostic output here are - ADVx_SLT: U Component Advective Flux of Salinity (psu m^3/s) - ADVy_SLT: V Component Advective Flux of Salinity (psu m^3/s)

The xgcm grid object is then used to take the convergence of the horizontal heat advection.

[21]:
```
ADVxy_diff = grid.diff_2d_vector({'X' : ds.ADVx_SLT, 'Y' : ds.ADVy_SLT}, boundary = 'fill
↪')

# Convergence of horizontal advection (psu m^3/s)
adv_hConvS = (-(ADVxy_diff['X'] + ADVxy_diff['Y']))
```

Vertical advective salt convergence

The relevant field from the diagnostic output is - ADVr_SLT: Vertical Advective Flux of Salinity (psu m^3/s)

[22]:
```
# Load monthly averages of vertical advective flux
ADVr_SLT = ds.ADVr_SLT.transpose('time','tile','k_l','j','i')
```

> **Note**: For ADVr_SLT, DFrE_SLT and DFrI_SLT, we need to make sure that sequence of dimensions are consistent. When loading the fields use .transpose('time','tile','k_l','j','i'). Otherwise, the divergences will be not correct (at least for tile = 12).

[23]:
```
# Convergence of vertical advection (psu/s)
adv_vConvS = grid.diff(ADVr_SLT, 'Z', boundary='fill')
```

> **Note**: In case of the volume budget (and salinity conservation), the surface forcing (oceFWflx) is already included at the top level (k_l = 0) in WVELMASS. Thus, to keep the surface forcing term explicitly represented, one needs to zero out the values of WVELMASS at the surface so as to avoid double counting (see ECCO_v4_Volume_budget_closure.ipynb). The salt budget only balances when the sea surface forcing is not added to the vertical salt flux (at the air-sea interface).

Total advective salt convergence

We can get the total convergence by simply adding the horizontal and vertical component.

[26]:
```
# Sum horizontal and vertical convergences and divide by volume (psu/s)
G_advection_Slt = (adv_hConvS + adv_vConvS)/vol
```

Diffusive salt convergence

Horizontal diffusive salt convergence

The relevant fields from the diagnostic output here are - DFxE_SLT: U Component Diffusive Flux of Salinity (psu m^3/s) - DFyE_SLT: V Component Diffusive Flux of Salinity (psu m^3/s)

As with advective fluxes, we use the xgcm grid object to calculate the convergence of horizontal salt diffusion.

[27]:
```
DFxyE_diff = grid.diff_2d_vector({'X' : ds.DFxE_SLT, 'Y' : ds.DFyE_SLT}, boundary = 'fill
↪')

# Convergence of horizontal diffusion (psu m^3/s)
dif_hConvS = (-(DFxyE_diff['X'] + DFxyE_diff['Y']))
```

Vertical diffusive salt convergence

The relevant fields from the diagnostic output are - DFrE_SLT: Vertical Diffusive Flux of Salinity (Explicit part) (psu m^3/s) - DFrI_SLT: Vertical Diffusive Flux of Salinity (Implicit part) (psu m^3/s) > **Note**: Vertical diffusion has both an explicit (DFrE_SLT) and an implicit (DFrI_SLT) part.

[28]:
```
# Load monthly averages of vertical diffusive fluxes
DFrE_SLT = ds.DFrE_SLT.transpose('time','tile','k_l','j','i')
DFrI_SLT = ds.DFrI_SLT.transpose('time','tile','k_l','j','i')

# Convergence of vertical diffusion (psu m^3/s)
dif_vConvS = grid.diff(DFrE_SLT, 'Z', boundary='fill') + grid.diff(DFrI_SLT, 'Z',␣
↪boundary='fill')
```

Total diffusive salt convergence

[29]:
```
# Sum horizontal and vertical convergences and divide by volume (psu/s)
G_diffusion_Slt = (dif_hConvS + dif_vConvS)/vol
```

Salt forcing

There are two relevant model diagnostics: - SFLUX: total salt flux (match salt-content variations) (g/m^2/s) - oceSPtnd: salt tendency due to salt plume flux (g/m^2/s)

The local forcing term reflects surface salt exchanges. There are two relevant model diagnostics here, namely the total salt exchange at the surface (SFLUX), which is nonzero only when sea ice melts or freezes, and the salt plume tendency (oceSPtnd), which vertically redistributes surface salt input by sea ice formation. We will merge SFLUX and oceSPtnd into a single data array (forcS) and convert it to units of psu per second.

[30]:
```
# Load SFLUX and add vertical coordinate
SFLUX = ds.SFLUX.assign_coords(k=0).expand_dims('k').transpose('time','tile','k','j','i')

# Calcualte forcing term by adding SFLUX and oceSPtnd (g/m^2/s)
forcS = xr.concat([SFLUX+ds.oceSPtnd,ds.oceSPtnd[:,:,1:]], dim='k')
```

SFLUX and oceSPtnd is given in g/m^2/s. Dividing by density and corresponding vertical length scale (drF) results in g/kg/s, which is the same as psu/s.

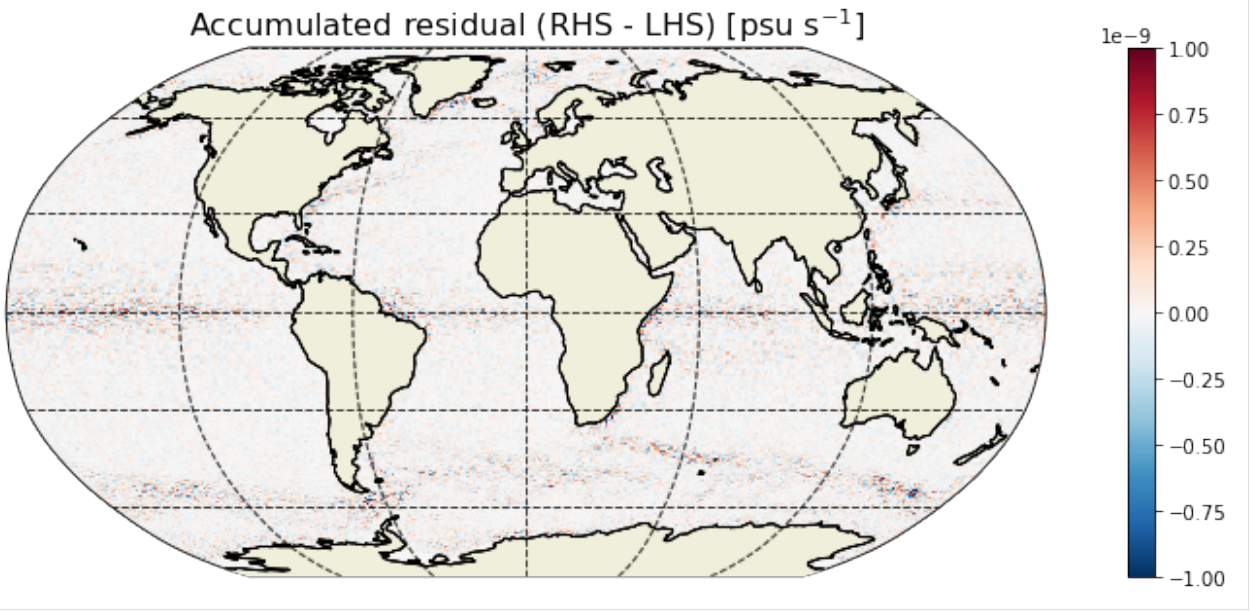
[31]:
```
# Forcing (psu/s)
G_forcing_Slt = forcS/rhoconst/(ecco_grid.hFacC*ecco_grid.drF)
```

Salt budget: Map of residual

[40]:
```
# Total convergence (psu/s)
ConvSlt = G_advection_Slt + G_diffusion_Slt

# Sum of terms in RHS of equation (psu/s)
rhs = ConvSlt + G_forcing_Slt
```

[41]:
```
# Accumulated residual
resSlt = (rhs-G_total_Slt).sum(dim='k').sum(dim='time').compute()
```

[42]:
```
plt.figure(figsize=(15,5))
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, resSlt,
                              cmin=-1e-9, cmax=1e-9, show_colorbar=True, cmap='RdBu_r',
↪dx=0.2, dy=0.2)
plt.title(r'Accumulated residual (RHS - LHS) [psu s$^{-1}$]', fontsize=16)
plt.show()
```



The above map confirms that the residual (summed over depth and time) is essentially zero everywhere, and the ECCOv4 salt budget can be closed to machine precision.

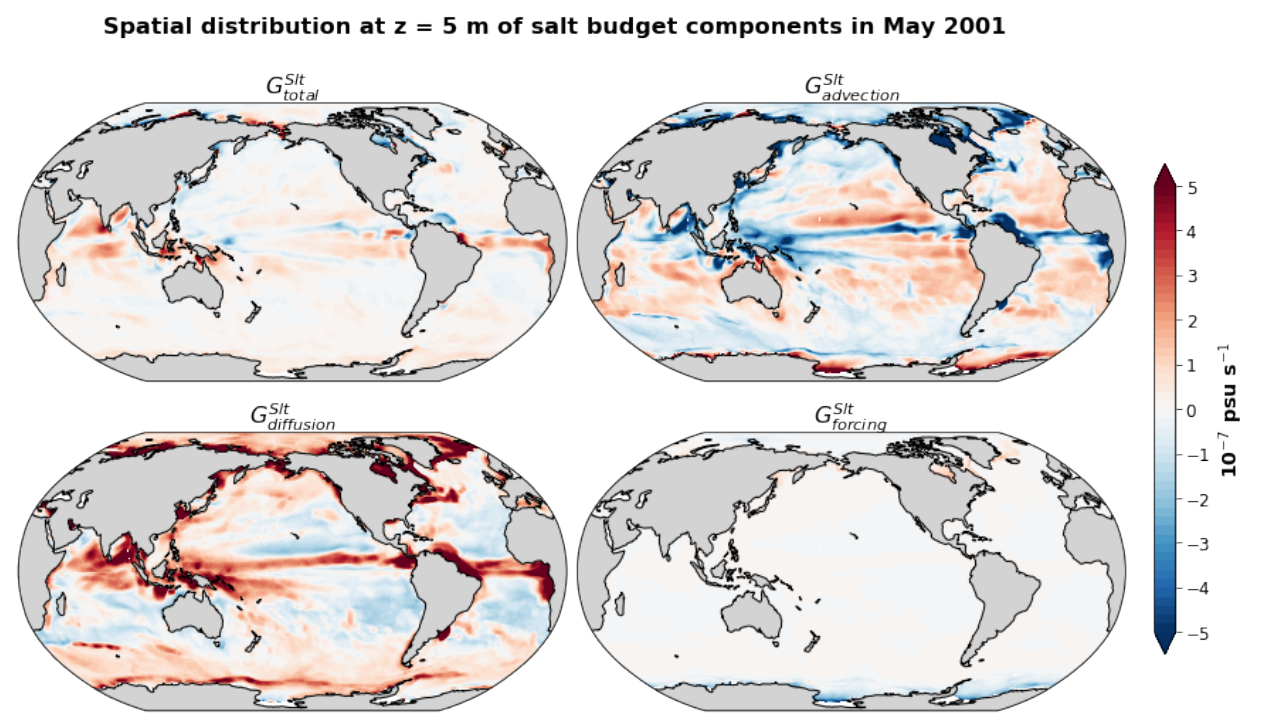
Salt budget: Spatial distributions

[43]:
```
# In order to plot the budget terms in one figure, let's add them in a list
var = [G_total_Slt,G_advection_Slt,G_diffusion_Slt,G_forcing_Slt]
varstrngs = [r'$G^{Slt}_{total}$',r'$G^{Slt}_{advection}$',r'$G^{Slt}_{diffusion}$',r'$G^
↪{Slt}_{forcing}$']
```

[44]:
```
# Set an index for the time (t) and depth (k) axis
t, k = 100, 0
```

Example maps at a particular time and depth level

[48]:
```
axes_class = (GeoAxes,dict(map_projection=cartopy.crs.Robinson(central_longitude=-159)))

fig = plt.figure(figsize=(14,8))
fig.suptitle('Spatial distribution at z = %i m of salt budget components in '\
             %np.round(-ecco_grid.Z[k].values)+str(ds.time[t].dt.strftime("%b %Y").
↪values),
             fontsize=16, fontweight='bold')
axgr = AxesGrid(fig, 111, axes_class=axes_class, nrows_ncols=(2, 2), axes_pad=(0.1 ,0.5),
                share_all=True, label_mode='')

for i, ax in enumerate(axgr):

    new_grid_lon, new_grid_lat, field_nearest_1deg =\
                    ecco.resample_to_latlon(ecco_grid.XC, ecco_grid.YC,
                                            var[i].isel(time=t,k=k),
                                            new_grid_min_lat, new_grid_max_lat, new_grid_
↪delta_lat,
                                            new_grid_min_lon, new_grid_max_lon, new_grid_
↪delta_lon,
                                            fill_value = np.NaN, mapping_method =
↪'nearest_neighbor',
                                            radius_of_influence = 120000)

    ax.coastlines(linewidth=1.0)
    ax.add_feature(cartopy.feature.LAND,color='lightgrey')
    ax.set_title(varstrngs[i],fontsize=16)
    p = ax.contourf(new_grid_lon, new_grid_lat, field_nearest_1deg*1e7,␣
↪transform=cartopy.crs.PlateCarree(),
                    vmin=-5, vmax=5, cmap='RdBu_r', levels=np.linspace(-5, 5, 51),␣
↪extend='both')

cax = fig.add_axes([0.92, 0.2, 0.015, 0.6])
cb = fig.colorbar(p, cax=cax, orientation='vertical',ticks=np.linspace(-5, 5, 11))
cb.ax.tick_params(labelsize=12)
cb.set_label(r'10$^{-7}$ psu s$^{-1}$', fontsize=14, fontweight='bold')

plt.show()
```



Time-mean distribution

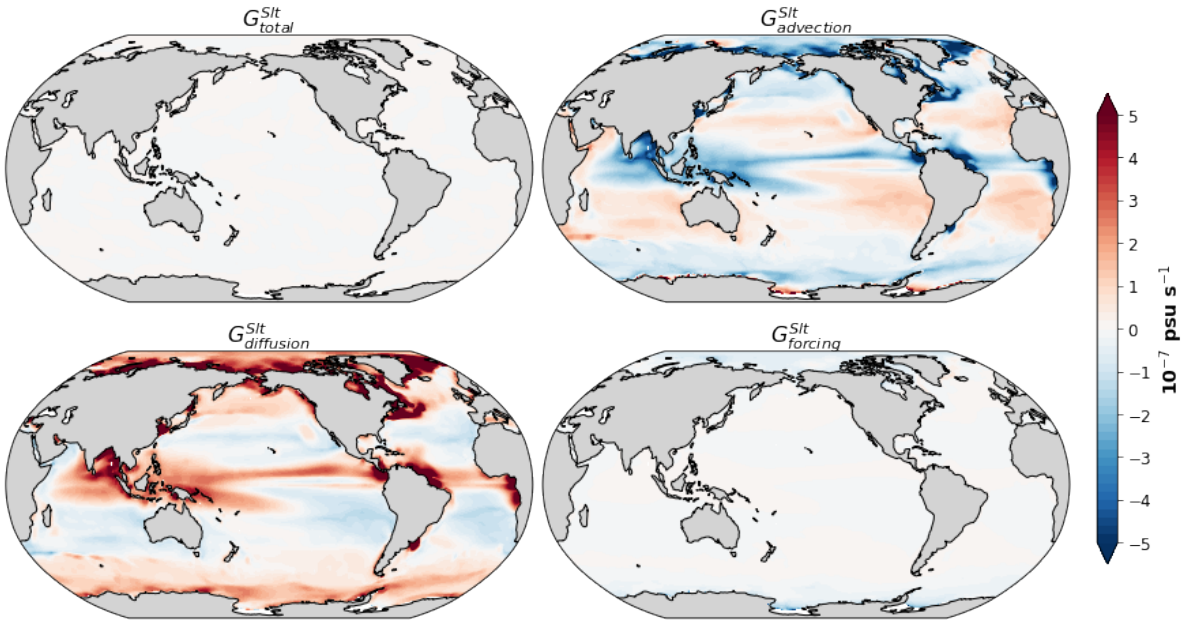
[49]:
```
axes_class = (GeoAxes,dict(map_projection=cartopy.crs.Robinson(central_longitude=-159)))

fig = plt.figure(figsize=(14,8))
fig.suptitle('Spatial distribution at z = %i m of salt budget components averaged over␣
↪period %i-%i,'\
             %(np.round(-ecco_grid.Z[k].values),year_start,year_end),
             fontsize=16, fontweight='bold')
axgr = AxesGrid(fig, 111, axes_class=axes_class, nrows_ncols=(2, 2), axes_pad=(0.1 ,0.5),
                share_all=True, label_mode='')

for i, ax in enumerate(axgr):

    new_grid_lon, new_grid_lat, field_nearest_1deg =\
                    ecco.resample_to_latlon(ecco_grid.XC, ecco_grid.YC,
                                            var[i].mean('time').isel(k=k),
                                            new_grid_min_lat, new_grid_max_lat, new_grid_
↪delta_lat,
                                            new_grid_min_lon, new_grid_max_lon, new_grid_
↪delta_lon,
                                            fill_value = np.NaN, mapping_method =
↪'nearest_neighbor',
                                            radius_of_influence = 120000)

    ax.coastlines(linewidth=1.0)
    ax.add_feature(cartopy.feature.LAND,color='lightgrey')
```

(continues on next page)

(continued from previous page)

```
    ax.set_title(varstrngs[i],fontsize=16)
    p = ax.contourf(new_grid_lon, new_grid_lat, field_nearest_1deg*1e7,␣
↪transform=cartopy.crs.PlateCarree(),
                    vmin=-5, vmax=5, cmap='RdBu_r', levels=np.linspace(-5, 5, 51),␣
↪extend='both')

cax = fig.add_axes([0.92, 0.2, 0.015, 0.6])
cb = fig.colorbar(p, cax=cax, orientation='vertical',ticks=np.linspace(-5, 5, 11))
cb.ax.tick_params(labelsize=12)
cb.set_label(r'10$^{-7}$ psu s$^{-1}$', fontsize=14, fontweight='bold')
plt.show()
```

Spatial distribution at z = 5 m of salt budget components averaged over period 1993-2015,



From the maps above, we can see that the balance in the salt budget is mostly between the advective and diffusive convergence, and the forcing term is only relevant close to the sea ice edge.

Salt budget closure through time

Global average budget closure

[50]:
```
# Take volume-weighted mean of these terms
tmp_a1 = (G_total_Slt*vol).sum(dim=('k','i','j','tile'))/vol.sum()
tmp_a2 = (rhs*vol).sum(dim=('k','i','j','tile'))/vol.sum()
tmp_b = (G_advection_Slt*vol).sum(dim=('k','i','j','tile'))/vol.sum()
tmp_c = (G_diffusion_Slt*vol).sum(dim=('k','i','j','tile'))/vol.sum()
tmp_d = (G_forcing_Slt*vol).sum(dim=('k','i','j','tile'))/vol.sum()
```

[51]:
```
fig, axs = plt.subplots(2, 2, figsize=(14,8))
```

(continues on next page)

(continued from previous page)

```
plt.sca(axs[0,0])
tmp_a1.plot(color='k',lw=2)
tmp_a2.plot(color='grey')
axs[0,0].set_title(r'a. $G^{Slt}_{total}$ (black) / RHS (grey) [psu s$^{-1}$]',␣
↪fontsize=12)
plt.grid()

plt.sca(axs[0,1])
tmp_b.plot(color='r')
axs[0,1].set_title(r'b. $G^{Slt}_{advection}$ [psu s$^{-1}$]', fontsize=12)
plt.grid()

plt.sca(axs[1,0])
tmp_c.plot(color='orange')
axs[1,0].set_title(r'c. $G^{Slt}_{diffusion}$ [psu s$^{-1}$]', fontsize=12)
plt.grid()

plt.sca(axs[1,1])
tmp_d.plot(color='b')
axs[1,1].set_title(r'd. $G^{Slt}_{forcing}$ [psu s$^{-1}$]', fontsize=12)
plt.grid()
plt.subplots_adjust(hspace = .5, wspace=.2)
plt.suptitle('Global Salt Budget', fontsize=16)
plt.show()
```

The globally-averaged salt budget is driven by the forcing term, which mostly represents the input/output of salt from

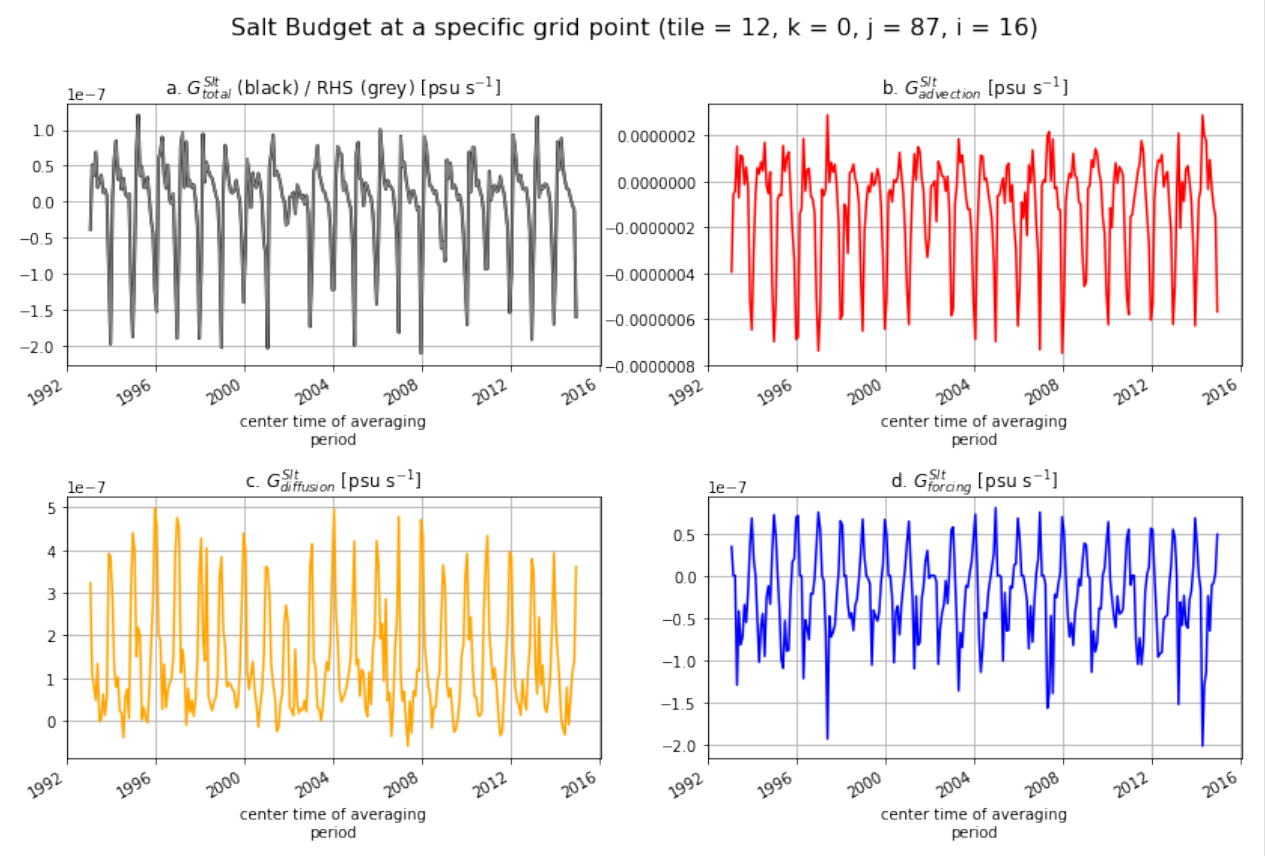
sea ice melting/freezing.

Local salt budget closure

[52]:
```
# Pick any set of indices (tile, k, j, i) corresponding to an ocean grid point
t,k,j,i = (12,0,87,16)
print(t,k,j,i)
```

```
12 0 87 16
```

[53]:
```
fig, axs = plt.subplots(2, 2, figsize=(14,8))

plt.sca(axs[0,0])
G_total_Slt.isel(tile=t,k=k,j=j,i=i).plot(color='k',lw=2)
rhs.isel(tile=t,k=k,j=j,i=i).plot(color='grey')
axs[0,0].set_title(r'a. $G^{Slt}_{total}$ (black) / RHS (grey) [psu s$^{-1}$]',␣
↪fontsize=12)
plt.grid()

plt.sca(axs[0,1])
G_advection_Slt.isel(tile=t,k=k,j=j,i=i).plot(color='r')
axs[0,1].set_title(r'b. $G^{Slt}_{advection}$ [psu s$^{-1}$]', fontsize=12)
plt.grid()

plt.sca(axs[1,0])
G_diffusion_Slt.isel(tile=t,k=k,j=j,i=i).plot(color='orange')
axs[1,0].set_title(r'c. $G^{Slt}_{diffusion}$ [psu s$^{-1}$]', fontsize=12)
plt.grid()

plt.sca(axs[1,1])
G_forcing_Slt.isel(tile=t,k=k,j=j,i=i).plot(color='b')
axs[1,1].set_title(r'd. $G^{Slt}_{forcing}$ [psu s$^{-1}$]', fontsize=12)
plt.grid()
plt.subplots_adjust(hspace = .5, wspace=.2)
plt.suptitle('Salt Budget at a specific grid point (tile = %i, k = %i, j = %i, i = %i)'
↪%(t,k,j,i), fontsize=16)

plt.show()
```



The balance looks very different for the local salt budget of a specific grid point. We see much greater magnitudes, mostly in the advective and diffusive part. The forcing component is an order of magnitude smaller than $G^{Slt}_{advection}$ and $G^{Slt}_{diffusion}$ and only relevant when sea ice is melting/freezing.

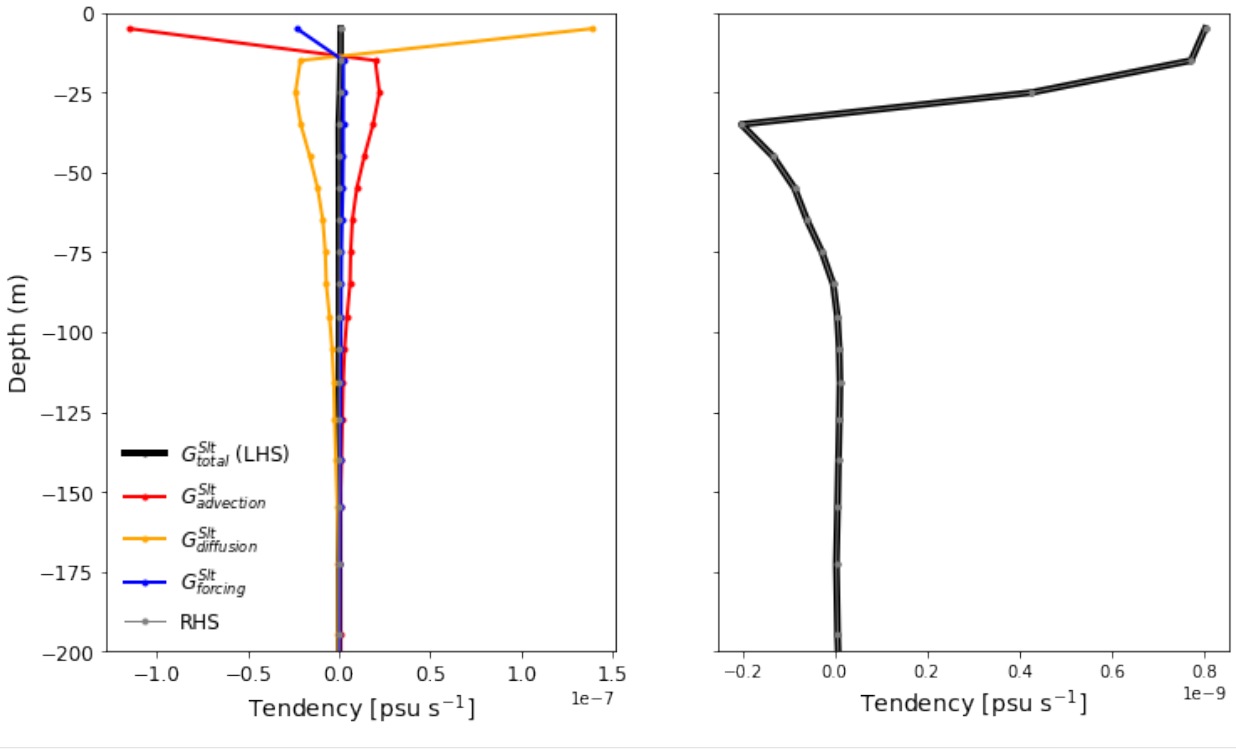
Vertical profiles of the salt budget terms

[45]:
```
fig = plt.subplots(1, 2, sharey=True, figsize=(12,7))

plt.subplot(1, 2, 1)
plt.plot(G_total_Slt.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z,
         lw=4, color='black', marker='.', label=r'$G^{Slt}_{total}$ (LHS)')

plt.plot(G_advection_Slt.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z,
         lw=2, color='red', marker='.', label=r'$G^{Slt}_{advection}$')

plt.plot(G_diffusion_Slt.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z,
         lw=2, color='orange', marker='.', label=r'$G^{Slt}_{diffusion}$')

plt.plot(G_forcing_Slt.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z,
         lw=2, color='blue', marker='.', label=r'$G^{Slt}_{forcing}$')
plt.plot(rhs.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z, lw=1, color='grey', marker=
↪'.', label='RHS')
plt.xlabel(r'Tendency [psu s$^{-1}$]', fontsize=14)
plt.ylim([-200,0])
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Depth (m)', fontsize=14)
plt.gca().tick_params(axis='both', which='major', labelsize=12)
plt.legend(loc='lower left', frameon=False, fontsize=12)

plt.subplot(1, 2, 2)
plt.plot(G_total_Slt.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z,
         lw=4, color='black', marker='.', label=r'$G^{Slt}_{total}$ (LHS)')
plt.plot(rhs.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z, lw=1, color='grey', marker=
↪'.', label='RHS')
plt.setp(plt.gca(), 'yticklabels',[])
plt.xlabel(r'Tendency [psu s$^{-1}$]', fontsize=14)
plt.ylim([-200,0])
plt.show()
```



The above examples illustrate that we can close the salt budget globally/spatially averaged, locally (for each grid point) at a specific time or averaged over time.

Given the nonlinear free surface condition, budgets for salt content (an extensive quantity) are not the same as budgets for salinity (an intensive quantity). The surface freshwater exchanges do not enter into the salt budget, since such fluxes do not affect the overall salt content, but rather make it more or less concentrated. However, a budget for salinity can be derived based on the conservation equations for salt and volume, and estimated using diagnostic model output. Such details are given in the section below.

1.22.5 Evaluating the salinity budget

In this section, we demonstrate how to estimate the salinity budget using output from the ECCOv4 solution. Each term in the following salinity budget equation will be evaluated.

$$G^{Sln}_{\text{total}} = G^{Sln}_{\text{advection}} + G^{Sln}_{\text{diffusion}} + G^{Sln}_{\text{forcing}}$$

Scale factor

Closing the salinity budget requires accurate estimates of volume changes for each grid cell. Thus, we need to explicitly calculate the scale factor (s^*) to be used in our calculations below.

$$s^* = 1 + \frac{\eta}{H}$$

This requires following model output: - `Depth`: Ocean depth, H (m) - `ETAN`: Surface Height Anomaly, η (m)

[52]:
```
# Scale factor
rstarfac = ((ecco_grid.Depth + ecco_monthly_mean.ETAN)/ecco_grid.Depth)
```

Total salinity tendency

We calculate the monthly-averaged time tendency of salinity by differencing monthly `SALT` snapshots. This operation includes dividing by the number of seconds between each snapshot.

$$G^{Sln}_{\text{total}} = \frac{\Delta S}{\Delta t}$$

[57]:
```
# Total tendency (psu/s)
G_total_Sln = grid.diff(ds.SALT_snp, 'T', boundary='fill', fill_value=0.0)/delta_t
```

Advective salinity convergence

Based on the derivation in the Introduction section, the salinity budget requires terms from both the volume and salt budgets. For the advective convergence of salinity, we first need to derive the convergence of volume.

Horizontal convergence

Relevant model output: - `UVELMASS`: U Mass-Weighted Component of Velocity (m/s) - `VVELMASS`: V Mass-Weighted Component of Velocity (m/s)

[58]:
```
# Horizontal volume transports (m^3/s)
u_transport = ds.UVELMASS * ecco_grid.dyG * ecco_grid.drF
v_transport = ds.VVELMASS * ecco_grid.dxG * ecco_grid.drF

uv_diff = grid.diff_2d_vector({'X' : u_transport, 'Y' : v_transport}, boundary = 'fill')

# Convergence of the horizontal flow (m^3/s)
hConvV = -(uv_diff['X'] + uv_diff['Y'])
```

Advective convergence of salinity has two parts: the advective salt flux (adv_hConvS), and the tendency due to volume convergence (hConvV).

[63]:
```
# Horizontal convergence of salinity (m^3/s)
adv_hConvSln = (-ds.SALT*hConvV + adv_hConvS)/rstarfac
```

Vertical convergence

Relevant model output: - WVELMASS: Vertical Mass-Weighted Component of Velocity (m/s) > Note: WVELMASS[k=0] == -oceFWflx/rho0. If we don't zero out the top cell, we end up double counting the surface flux.

[61]:
```
# Vertical volume transport (m^3/s)
w_transport = ds.WVELMASS.where(ds.k_l>0).fillna(0.) * ecco_grid.rA
```

[62]:
```
# Convergence of the vertical flow (m^3/s)
vConvV = grid.diff(w_transport, 'Z', boundary='fill')
```

Again, to get the vertical convergence of salinity we need both the vertical salt flux (adv_vConvS) and convergece of vertical flow (vConvV).

[64]:
```
# Vertical convergence of salinity (psu m^3/s)
adv_vConvSln = (-ds.SALT*vConvV + adv_vConvS)/rstarfac
```

Total advective salinity convergence

[65]:
```
# Total convergence of advective salinity flux (psu/s)
G_advection_Sln = (adv_hConvSln + adv_vConvSln)/vol
```

Diffusive salinity convergence

The diffusive flux of salinity is pretty much the same as for salt. The only step is dividing the convergence of salt diffusion by the scale factor.

[66]:
```
# Horizontal convergence
dif_hConvSln = dif_hConvS/rstarfac

# Vertical convergence
dif_vConvSln = dif_vConvS/rstarfac

# Sum horizontal and vertical convergences and divide by volume (psu/s)
G_diffusion_Sln = (dif_hConvSln + dif_vConvSln)/vol
```

Salinity forcing

The forcing term is comprised of both salt flux (`forcS`) and volume (i.e., surface freshwater) fluxes (`forcV`). We now require monthly mean salinity `SALT` to convert `forcV` to appropriate units.

Volume forcing

- `oceFWflx`: net surface Fresh-Water flux into the ocean (kg/m^2/s)

[67]:
```
# Load monthly averaged freshwater flux and add vertical coordinate
oceFWflx = ds.oceFWflx.assign_coords(k=0).expand_dims('k')

# Sea surface forcing on volume (1/s)
forcV = xr.concat([(oceFWflx/rhoconst)/(ecco_grid.hFacC*ecco_grid.drF),
                   xr.zeros_like(((oceFWflx[0]/rhoconst)/\
                                  (ecco_grid.hFacC*ecco_grid.drF)).transpose('time','tile
↪','k','j','i'))[:,:,1:]],
                  dim='k').transpose('time','tile','k','j','i')
```

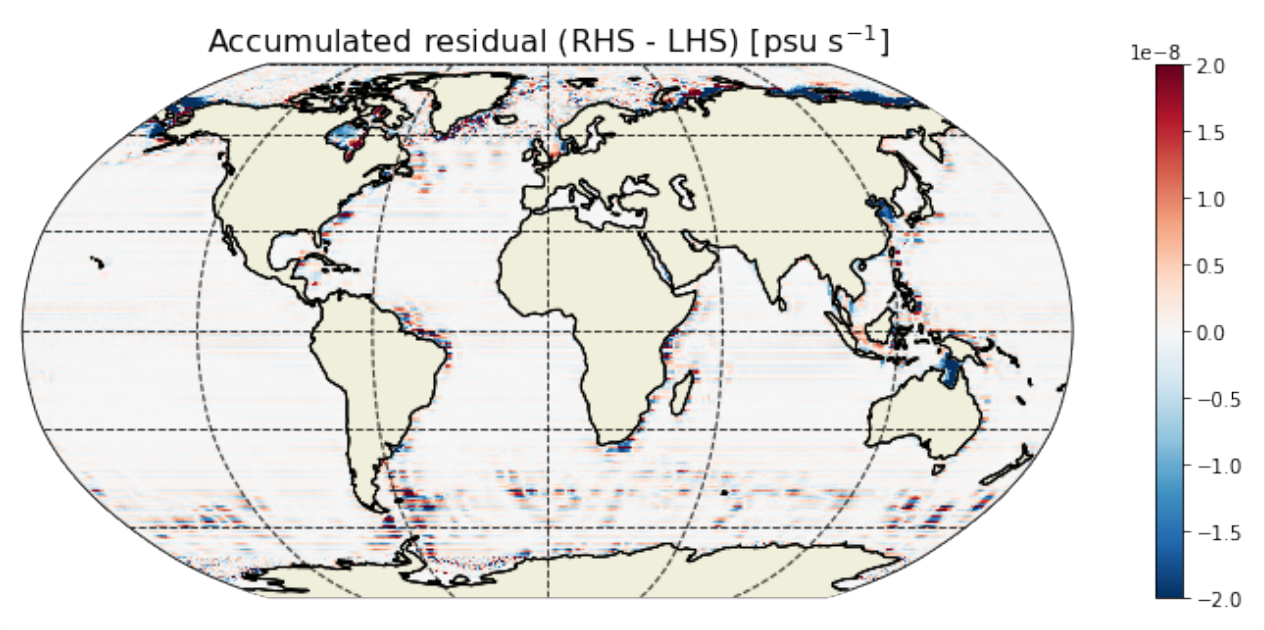
[68]:
```
# Sea surface forcing for salinity (psu/s)
G_forcing_Sln = (-ds.SALT*forcV + G_forcing_Slt)/rstarfac
```

Salinity budget: Map of residual

[69]:
```
# Total convergence (psu/s)
ConvSln = G_advection_Sln + G_diffusion_Sln

# Sum of terms in RHS of equation (psu/s)
rhs_Sln = ConvSln + G_forcing_Sln
```

[70]:
```
# Accumulated residual
resSln = (rhs_Sln-G_total_Sln).sum(dim='k').sum(dim='time').compute()
```

[78]:
```
plt.figure(figsize=(15,5))
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, resSln,
                              cmin=-2e-8, cmax=2e-8, show_colorbar=True, cmap='RdBu_r',
↪dx=0.2, dy=0.2)
plt.title(r'Accumulated residual (RHS - LHS) [psu s$^{-1}$]', fontsize=16)
plt.show()
```



The residual in the salinity budget are more extensive compared to the salt budget. Here errors occur that are mostly found in the continental shelves and high latitudes. However, given that the above map shows the accumulated residual, the errors are very small compared to the salinity tendencies' overall range of values.

Salinity budget: Spatial distributions

[89]:
```
# In order to plot the budget terms in one figure, let's add them in a list
var = [G_total_Sln,G_advection_Sln,G_diffusion_Sln,G_forcing_Sln]
varstrngs = [r'$G^{Sln}_{total}$',r'$G^{Sln}_{advection}$',r'$G^{Sln}_{diffusion}$',r'$G^
↪{Sln}_{forcing}$']
```

[90]:
```
# Set an index for the time (t) and depth (k) axis
t, k = 100, 0
```

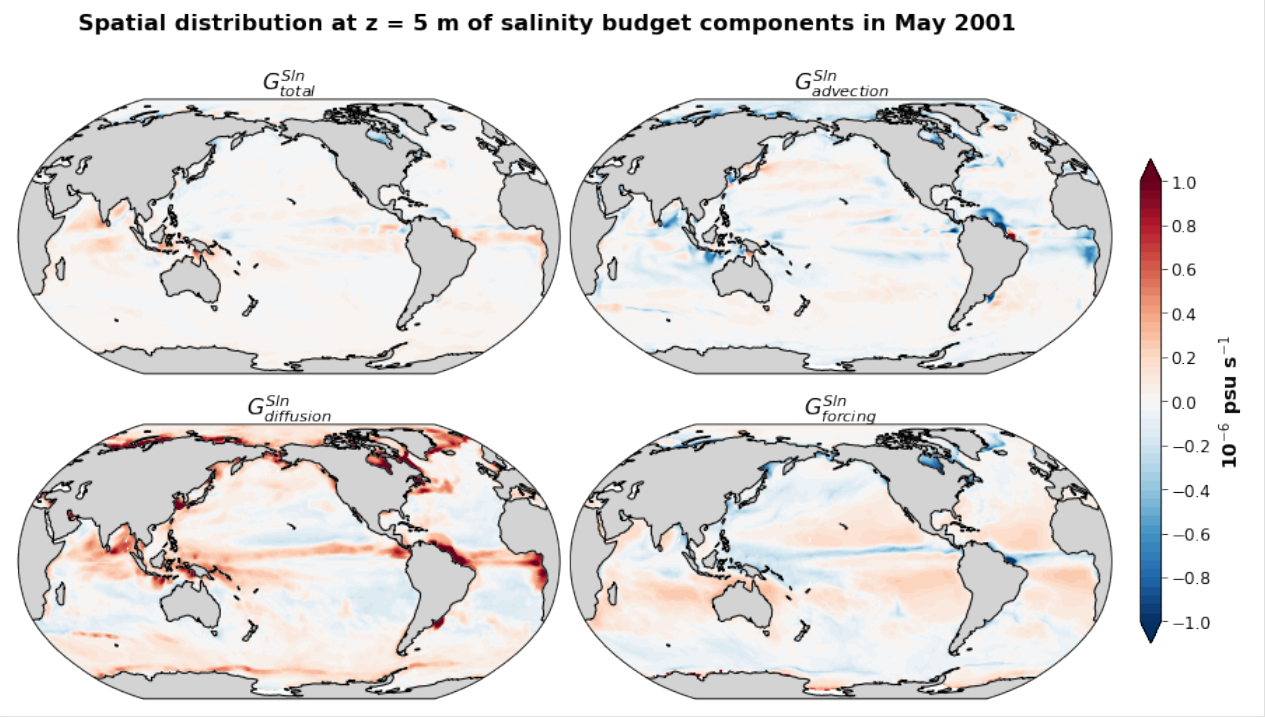
Example maps at a particular time and depth level

[93]:
```
axes_class = (GeoAxes,dict(map_projection=cartopy.crs.Robinson(central_longitude=-159)))

fig = plt.figure(figsize=(14,8))
fig.suptitle('Spatial distribution at z = %i m of salinity budget components in '\
             %np.round(-ecco_grid.Z[k].values)+str(ds.time[t].dt.strftime("%b %Y").
↪values),
             fontsize=16, fontweight='bold')
axgr = AxesGrid(fig, 111, axes_class=axes_class, nrows_ncols=(2, 2), axes_pad=(0.1 ,0.5),
                share_all=True, label_mode='')

for i, ax in enumerate(axgr):

    new_grid_lon, new_grid_lat, field_nearest_1deg =\
```

(continues on next page)

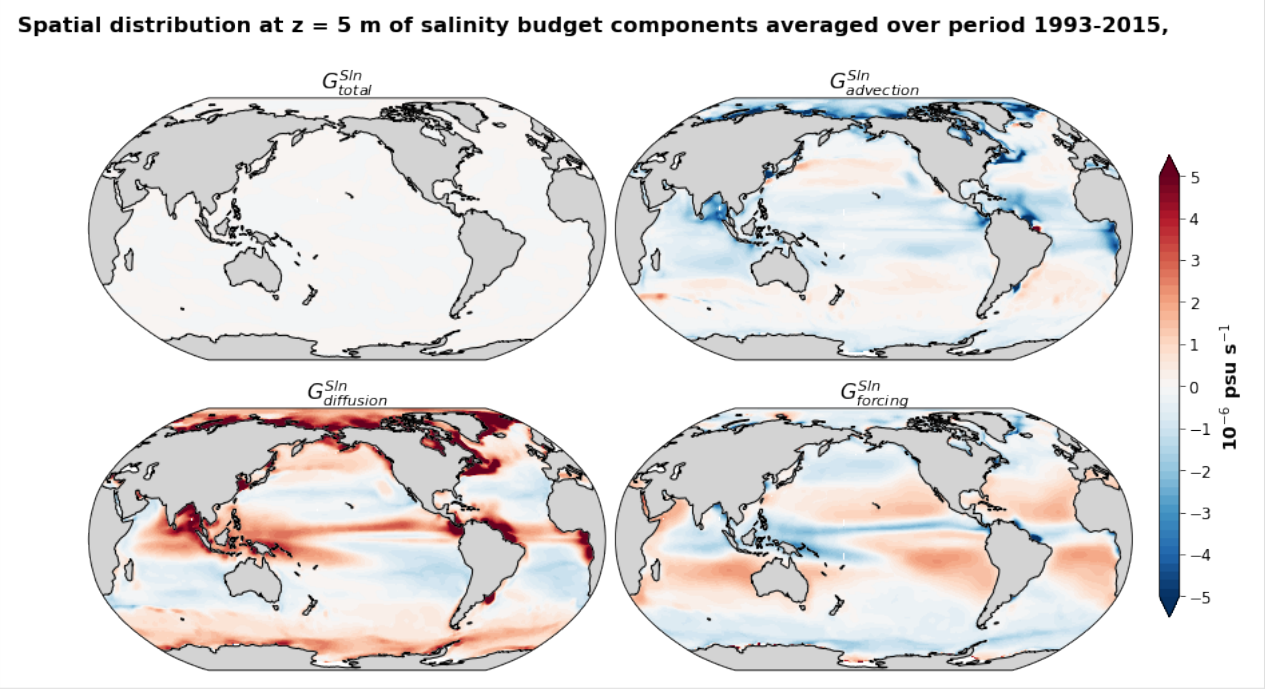
(continued from previous page)

```
                    ecco.resample_to_latlon(ecco_grid.XC, ecco_grid.YC,
                                            var[i].isel(time=t,k=k),
                                            new_grid_min_lat, new_grid_max_lat, new_grid_
↪delta_lat,
                                            new_grid_min_lon, new_grid_max_lon, new_grid_
↪delta_lon,
                                            fill_value = np.NaN, mapping_method =
↪'nearest_neighbor',
                                            radius_of_influence = 120000)

    ax.coastlines(linewidth=1.0,zorder=2)
    ax.add_feature(cartopy.feature.LAND,color='lightgrey',zorder=1)
    ax.set_title(varstrngs[i],fontsize=16)
    p = ax.contourf(new_grid_lon, new_grid_lat, field_nearest_1deg*1e6,␣
↪transform=cartopy.crs.PlateCarree(),
                    vmin=-1, vmax=1, cmap='RdBu_r', levels=np.linspace(-1, 1, 51),␣
↪extend='both',zorder=0)

cax = fig.add_axes([0.92, 0.2, 0.015, 0.6])
cb = fig.colorbar(p, cax=cax, orientation='vertical',ticks=np.linspace(-1, 1, 11))
cb.ax.tick_params(labelsize=12)
cb.set_label(r'10$^{-6}$ psu s$^{-1}$', fontsize=14, fontweight='bold')

plt.show()
```



Time-mean distribution

[99]:
```
axes_class = (GeoAxes,dict(map_projection=cartopy.crs.Robinson(central_longitude=-159)))

fig = plt.figure(figsize=(14,8))
fig.suptitle('Spatial distribution at z = %i m of salinity budget components averaged␣
↪over period %i-%i,'\
             %(np.round(-ecco_grid.Z[k].values),year_start,year_end),
             fontsize=16, fontweight='bold')
axgr = AxesGrid(fig, 111, axes_class=axes_class, nrows_ncols=(2, 2), axes_pad=(0.1 ,0.5),
                share_all=True, label_mode='')

for i, ax in enumerate(axgr):

    new_grid_lon, new_grid_lat, field_nearest_1deg =\
                    ecco.resample_to_latlon(ecco_grid.XC, ecco_grid.YC,
                                            var[i].mean('time').isel(k=k),
                                            new_grid_min_lat, new_grid_max_lat, new_grid_
↪delta_lat,
                                            new_grid_min_lon, new_grid_max_lon, new_grid_
↪delta_lon,
                                            fill_value = np.NaN, mapping_method =
↪'nearest_neighbor',
                                            radius_of_influence = 120000)

    ax.coastlines(linewidth=1.0,zorder=2)
    ax.add_feature(cartopy.feature.LAND,color='lightgrey',zorder=1)
    ax.set_title(varstrngs[i],fontsize=16)
    p = ax.contourf(new_grid_lon, new_grid_lat, field_nearest_1deg*1e7,␣
↪transform=cartopy.crs.PlateCarree(),
                    vmin=-5, vmax=5, cmap='RdBu_r', levels=np.linspace(-5, 5, 51),␣
↪extend='both',zorder=0)

cax = fig.add_axes([0.92, 0.2, 0.015, 0.6])
cb = fig.colorbar(p, cax=cax, orientation='vertical',ticks=np.linspace(-5, 5, 11))
cb.ax.tick_params(labelsize=12)
cb.set_label(r'10$^{-6}$ psu s$^{-1}$', fontsize=14, fontweight='bold')
plt.show()
```



Unlike with the salt budget, we now see a clear spatial pattern in the forcing term, which resembles surface freshwater flux.

Salinity budget closure through time

This section illustrates that we can close the salinity budget globally and locally (i.e., at any given grid point).

Global average budget closure

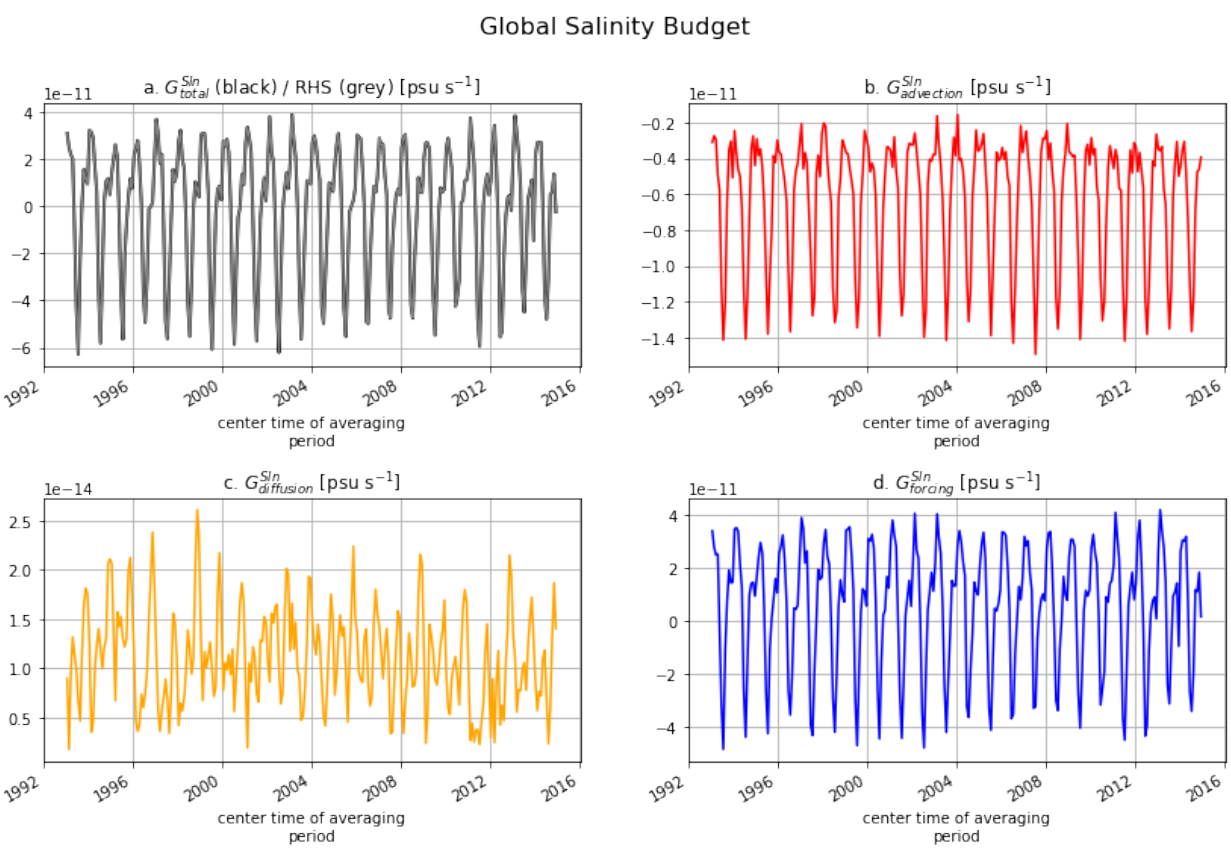
[94]:
```
# Take volume-weighted mean of these terms
tmp_a1 = (G_total_Sln*vol).sum(dim=('k','i','j','tile'))/vol.sum()
tmp_a2 = (rhs_Sln*vol).sum(dim=('k','i','j','tile'))/vol.sum()
tmp_b = (G_advection_Sln*vol).sum(dim=('k','i','j','tile'))/vol.sum()
tmp_c = (G_diffusion_Sln*vol).sum(dim=('k','i','j','tile'))/vol.sum()
tmp_d = (G_forcing_Sln*vol).sum(dim=('k','i','j','tile'))/vol.sum()
```

[95]:
```
fig, axs = plt.subplots(2, 2, figsize=(14,8))

plt.sca(axs[0,0])
tmp_a1.plot(color='k',lw=2)
tmp_a2.plot(color='grey')
axs[0,0].set_title(r'a. $G^{Sln}_{total}$ (black) / RHS (grey) [psu s$^{-1}$]',␣
↪fontsize=12)
plt.grid()

plt.sca(axs[0,1])
tmp_b.plot(color='r')
axs[0,1].set_title(r'b. $G^{Sln}_{advection}$ [psu s$^{-1}$]', fontsize=12)
```

(continues on next page)

(continued from previous page)

```
plt.grid()

plt.sca(axs[1,0])
tmp_c.plot(color='orange')
axs[1,0].set_title(r'c. $G^{Sln}_{diffusion}$ [psu s$^{-1}$]', fontsize=12)
plt.grid()

plt.sca(axs[1,1])
tmp_d.plot(color='b')
axs[1,1].set_title(r'd. $G^{Sln}_{forcing}$ [psu s$^{-1}$]', fontsize=12)
plt.grid()
plt.subplots_adjust(hspace = .5, wspace=.2)
plt.suptitle('Global Salinity Budget', fontsize=16)
plt.show()
```

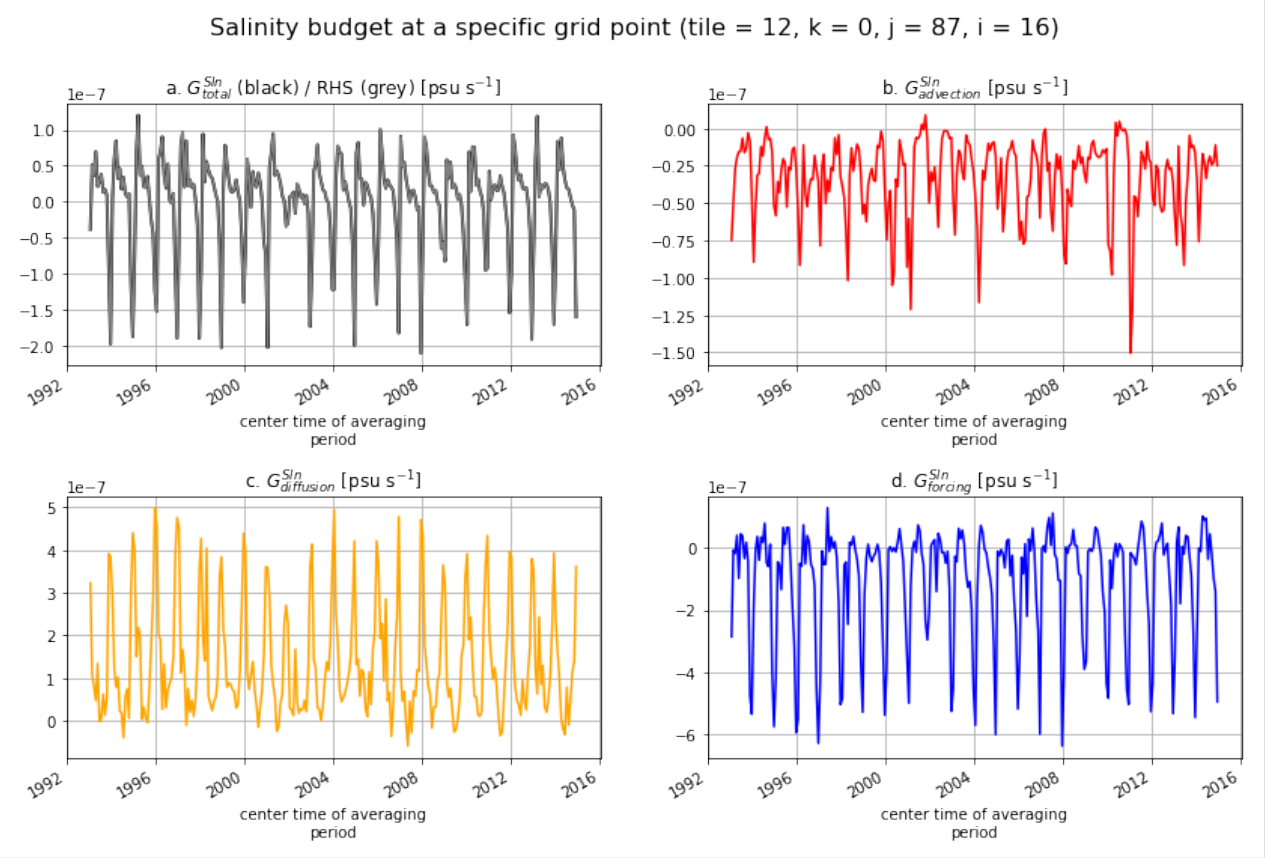


Local salt budget closure

[96]:
```
# Pick any set of indices (tile, k, j, i) corresponding to an ocean grid point
t,k,j,i = (12,0,87,16)
print(t,k,j,i)
```

```
12 0 87 16
```

[97]:
```
fig, axs = plt.subplots(2, 2, figsize=(14,8))

plt.sca(axs[0,0])
G_total_Sln.isel(tile=t,k=k,j=j,i=i).plot(color='k',lw=2)
rhs_Sln.isel(tile=t,k=k,j=j,i=i).plot(color='grey')
axs[0,0].set_title(r'a. $G^{Sln}_{total}$ (black) / RHS (grey) [psu s$^{-1}$]',␣
↪fontsize=12)
plt.grid()

plt.sca(axs[0,1])
G_advection_Sln.isel(tile=t,k=k,j=j,i=i).plot(color='r')
axs[0,1].set_title(r'b. $G^{Sln}_{advection}$ [psu s$^{-1}$]', fontsize=12)
plt.grid()

plt.sca(axs[1,0])
G_diffusion_Sln.isel(tile=t,k=k,j=j,i=i).plot(color='orange')
axs[1,0].set_title(r'c. $G^{Sln}_{diffusion}$ [psu s$^{-1}$]', fontsize=12)
plt.grid()

plt.sca(axs[1,1])
G_forcing_Sln.isel(tile=t,k=k,j=j,i=i).plot(color='b')
axs[1,1].set_title(r'd. $G^{Sln}_{forcing}$ [psu s$^{-1}$]', fontsize=12)
plt.grid()
plt.subplots_adjust(hspace = .5, wspace=.2)
plt.suptitle('Salinity budget at a specific grid point (tile = %i, k = %i, j = %i, i =
↪%i)'%(t,k,j,i), fontsize=16)

plt.show()
```



Vertical profiles of the salinity budget terms

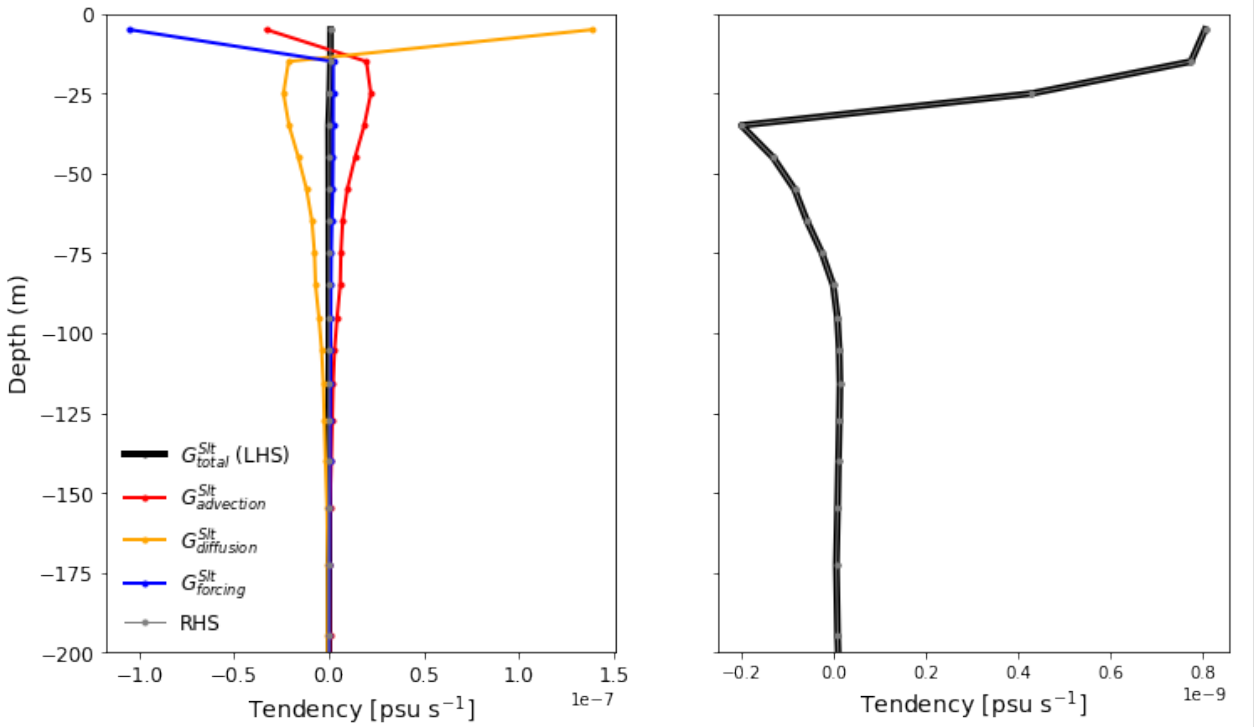
This section illustrates the balance in the salinity budget along the depth axis.

[98]:
```
fig = plt.subplots(1, 2, sharey=True, figsize=(12,7))

plt.subplot(1, 2, 1)
plt.plot(G_total_Sln.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z,
         lw=4, color='black', marker='.', label=r'$G^{Slt}_{total}$ (LHS)')

plt.plot(G_advection_Sln.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z,
         lw=2, color='red', marker='.', label=r'$G^{Slt}_{advection}$')

plt.plot(G_diffusion_Sln.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z,
         lw=2, color='orange', marker='.', label=r'$G^{Slt}_{diffusion}$')

plt.plot(G_forcing_Sln.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z,
         lw=2, color='blue', marker='.', label=r'$G^{Slt}_{forcing}$')
plt.plot(rhs_Sln.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z, lw=1, color='grey',␣
↪marker='.', label='RHS')
plt.xlabel(r'Tendency [psu s$^{-1}$]', fontsize=14)
plt.ylim([-200,0])
plt.ylabel('Depth (m)', fontsize=14)
plt.gca().tick_params(axis='both', which='major', labelsize=12)
```

(continues on next page)

(continued from previous page)

```
plt.legend(loc='lower left', frameon=False, fontsize=12)

plt.subplot(1, 2, 2)
plt.plot(G_total_Sln.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z,
         lw=4, color='black', marker='.', label=r'$G^{Slt}_{total}$ (LHS)')
plt.plot(rhs_Sln.isel(tile=t,j=j,i=i).mean('time'), ecco_grid.Z, lw=1, color='grey',␣
↪marker='.', label='RHS')
plt.setp(plt.gca(), 'yticklabels',[])
plt.xlabel(r'Tendency [psu s$^{-1}$]', fontsize=14)
plt.ylim([-200,0])
plt.show()
```



1.22.6 Evaluating the freshwater budget - *Work in progress*

As with the salt and a salinity budget we will evaluate each term in the freshwater budget.

$$G^{fw}_{\text{total}} = G^{fw}_{\text{advection}} + G^{fw}_{\text{diffusion}} + G^{fw}_{\text{forcing}}$$

Suggestion by Matt Mazloff:

The reference is just a multiplying factor

$$f = \frac{S_{ref} - S}{S_{ref}}$$

So the budget is

$$\frac{\partial f}{\partial t} = -\frac{1}{S_{ref}}\frac{\partial S}{\partial t} = -\frac{1}{S_{ref}}\left(G^{Sln}_{\text{advection}} + G^{Sln}_{\text{diffusion}} + G^{Sln}_{\text{forcing}}\right)$$

And the user can specify the S_{ref} later or easily convert. I would just use 35 for the tutorial. And you use the exact salinity budget terms. The diffusive flux of freshwater is minus the diffusive flux of salt.

$$G^{Fw}_{\text{total}} = -\frac{G^{Sln}_{\text{advection}}}{S_{ref}} - \frac{G^{Sln}_{\text{diffusion}}}{S_{ref}} - \frac{G^{Sln}_{\text{forcing}}}{S_{ref}}$$

[46]:
```
# Reference salinity
Sref = 35.0
```

Total freshwater tendency

[47]:
```
f = (Sref - ecco_monthly_snaps.SALT)/Sref

G_total_Fw_month = f.isel(time=range(1,num_months)).values - f.isel(time=range(0,num_
↪months-1)).values
```

[48]:
```
# Convert numpy array to an xarray DataArray with matching dimensions as the monthly␣
↪mean fields
G_total_Fw_month = xr.DataArray(G_total_Fw_month, coords={'time': ecco_monthly_mean.time.
↪values,
                                                          'tile': ecco_monthly_mean.tile.
↪values,
                                                          'k': ecco_monthly_mean.k.
↪values,
                                                          'j': ecco_monthly_mean.j.
↪values,
                                                          'i': ecco_monthly_mean.i.
↪values},
                                dims=['time','tile','k','j','i'])
```

[49]:
```
# Freshwater tendency (m^3/s)
G_total_Fw = G_total_Fw_month*vol / secs_per_month
```

Advective flux of freshwater

Advective fluxes of freshwater are calculated offline using salinity and velocity fields:

$$\mathcal{F}_{\mathbf{adv}} = \iint_A \mathbf{u}_{res} \cdot \left(\frac{S_{ref} - S}{S_{ref}} \right) dA \tag{1.20}$$

GM Bolus Velocity

[50]:
```
UVELSTAR = grid.diff(ecco_monthly_mean.GM_PsiX, 'Z', boundary='fill')/ecco_grid.drF
VVELSTAR = grid.diff(ecco_monthly_mean.GM_PsiY, 'Z', boundary='fill')/ecco_grid.drF
```

[51]:
```
GM_PsiXY_diff = grid.diff_2d_vector({'X' : ecco_monthly_mean.GM_PsiX*ecco_grid.dyG,
                                     'Y' : ecco_monthly_mean.GM_PsiY*ecco_grid.dxG},␣
↪boundary = 'fill')
WVELSTAR = (GM_PsiXY_diff['X'] + GM_PsiXY_diff['Y'])/ecco_grid.rA
```

Calculate advective freshwater flux

[52]:
```
SALT_at_u = grid.interp(ecco_monthly_mean.SALT, 'X', boundary='extend')
SALT_at_v = grid.interp(ecco_monthly_mean.SALT, 'Y', boundary='extend')
SALT_at_w = grid.interp(ecco_monthly_mean.SALT, 'Z', boundary='extend')
```

[53]:
```
# Freshwater advective (Eulerian+Bolus) fluxes (m^3/s)
ADVx_FW = (ecco_monthly_mean.UVELMASS+UVELSTAR)*ecco_grid.dyG*ecco_grid.drF*(Sref-SALT_
↪at_u)/Sref
ADVy_FW = (ecco_monthly_mean.VVELMASS+VVELSTAR)*ecco_grid.dxG*ecco_grid.drF*(Sref-SALT_
↪at_v)/Sref
ADVr_FW = WVELMASS*ecco_grid.rA*(Sref-SALT_at_w)/Sref
```

Salinity advective (Eulerian+Bolus) fluxes (psu m$^3/s$)$ADVx_FW = (ecco_monthly_mean.UVELMASS + UVELSTAR) * ecco_grid.dyG * ecco_grid.drF * SALT_at_u ADVy_FW = (ecco_monthly_mean.VVELMASS + VVELSTAR) * ecco_grid.dxG * ecco_grid.drF * SALT_at_v ADVr_FW = WVELMASS * ecco_grid.rA * SALT_at_w$

[54]:
```
ADVxy_diff = grid.diff_2d_vector({'X' : ADVx_FW, 'Y' : ADVy_FW}, boundary = 'fill')

# Convergence of horizontal advection (m^3/s)
adv_hConvFw = (-(ADVxy_diff['X'] + ADVxy_diff['Y']))
```

[55]:
```
# Convergence of vertical advection (m^3/s)
adv_vConvFw = grid.diff(ADVr_FW, 'Z', boundary='fill')
```

[56]:
```
# Sum horizontal and vertical convergences (m^3/s)
G_advection_Fw = adv_hConvFw + adv_vConvFw
```

Freshwater forcing

[57]:
```
# Freshwater forcing (m^3/s)
forcFw = ecco_monthly_mean.oceFWflx/rhoconst*ecco_grid.rA

# Expand to fully 3d (using G_advection_Fw as template)
G_forcing_Fw = xr.concat([forcFw.reset_coords(drop=True).assign_coords(k=0).expand_dims(
↪'k')\
                        .transpose('time','tile','k','j','i'),
                           xr.zeros_like(G_advection_Fw.transpose('time','tile','k','j','i
↪'))[:,:,1:]],
                          dim='k').transpose('time','tile','k','j','i').where(ecco_grid.
↪hFacC==1)
```

Diffusive freshwater flux

[58]:
```
# Convergence of freshwater diffusion (m^3/s)
G_diffusion_Fw = G_total_Fw - G_forcing_Fw - G_advection_Fw
```

1.22.7 Save budget terms

Now that we have all the terms evaluated, let's save them to a dataset. Here are two examples:

Add all variables to a new dataset

[59]:
```
#varnames = ['G_total_Slt','G_advection_Slt','G_diffusion_Slt','G_forcing_Slt',
#            'G_total_Sln','G_advection_Sln','G_diffusion_Sln','G_forcing_Sln',
#            'G_total_Fw', 'G_advection_Fw', 'G_diffusion_Fw', 'G_forcing_Fw']
varnames = ['G_total_Sln','G_advection_Sln','G_diffusion_Sln','G_forcing_Sln']
ds = xr.Dataset(data_vars={})
for varname in varnames:
    ds[varname] = globals()[varname].chunk(chunks={'time':1,'tile':13,'k':50,'j':90,'i':
↪90})
```

[60]:
```
ds.time.encoding = {}
ds = ds.reset_coords(drop=True)
```

Save to zarr dataset

[61]:
```
from dask.diagnostics import ProgressBar
```

[62]:
```
ds
```

[62]:
```
<xarray.Dataset>
Dimensions:          (i: 90, j: 90, k: 50, tile: 13, time: 264)
Coordinates:
  * time             (time) datetime64[ns] 1993-01-16T12:00:00 ... 2014-12-16...
```

(continues on next page)

(continued from previous page)

```
  * tile             (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
  * k                (k) int32 0 1 2 3 4 5 6 7 8 ... 41 42 43 44 45 46 47 48 49
  * j                (j) int32 0 1 2 3 4 5 6 7 8 ... 81 82 83 84 85 86 87 88 89
  * i                (i) int32 0 1 2 3 4 5 6 7 8 ... 81 82 83 84 85 86 87 88 89
Data variables:
    G_total_Sln      (time, tile, k, j, i) float64 dask.array<chunksize=(1, 13, 50, 90,␣
↪90), meta=np.ndarray>
    G_advection_Sln  (time, tile, k, j, i) float32 dask.array<chunksize=(1, 13, 50, 90,␣
↪90), meta=np.ndarray>
    G_diffusion_Sln  (time, tile, k, j, i) float32 dask.array<chunksize=(1, 13, 50, 90,␣
↪90), meta=np.ndarray>
    G_forcing_Sln    (time, tile, k, j, i) float32 dask.array<chunksize=(1, 13, 50, 90,␣
↪90), meta=np.ndarray>
```

[63]:
```
with ProgressBar():
    ds.to_zarr(base_dir + '/eccov4r3_budg_Slt_Sln_Fw')
```

```
[########################################] | 100% Completed | 12min 42.1s
```

1.22.8 Load budget variables from file

After having saved the budget terms to file, let's restart the kernel and load only the relevant data and Python modules.

[1]:
```
# Suppress warning messages for a cleaner presentation
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import xarray as xr
import ecco_v4_py as ecco

import matplotlib.pyplot as plt
%matplotlib inline

base_dir = '/mnt/efs/data/ECCOv4-release'
ECCO_dir = base_dir + '/Release3_alt'
grid_dir= ECCO_dir + '/nctiles_grid/'
ecco_grid = ecco.load_ecco_grid_nc(grid_dir, 'ECCOv4r3_grid.nc')
grid = ecco.get_llc_grid(ecco_grid)
# Volume (m^3)
vol = (ecco_grid.rA*ecco_grid.drF*ecco_grid.hFacC).transpose('tile','k','j','i')

data_dir= ECCO_dir + '/nctiles_monthly'
year_start = 1993
year_end = 2015
ecco_monthly_mean = ecco.recursive_load_ecco_var_from_years_nc(data_dir, \
                        vars_to_load=['UVELMASS', 'VVELMASS', 'WVELMASS', 'GM_PsiX', 'GM_
↪PsiY', 'SALT', 'oceFWflx'],
                                                               years_to_load=range(year_
↪start, year_end))
```

```
loading files of  GM_PsiX
loading files of  GM_PsiY
loading files of  SALT
loading files of  UVELMASS
loading files of  VVELMASS
loading files of  WVELMASS
loading files of  oceFWflx
```

[2]:
```
# Load terms from zarr dataset
G_total_Sln = xr.open_zarr(base_dir + '/eccov4r3_budg_Slt_Sln_Fw').G_total_Sln
G_advection_Sln = xr.open_zarr(base_dir + '/eccov4r3_budg_Slt_Sln_Fw').G_advection_Sln
G_diffusion_Sln = xr.open_zarr(base_dir + '/eccov4r3_budg_Slt_Sln_Fw').G_diffusion_Sln
G_forcing_Sln = xr.open_zarr(base_dir + '/eccov4r3_budg_Slt_Sln_Fw').G_forcing_Sln
```

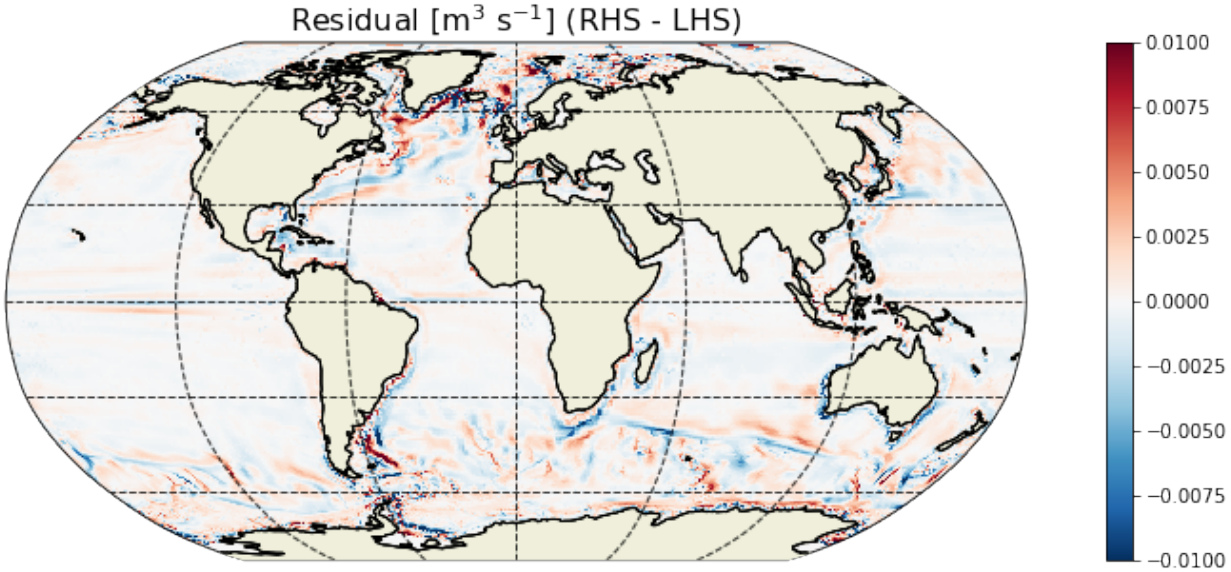
[4]:
```
UVELSTAR = grid.diff(ecco_monthly_mean.GM_PsiX, 'Z', boundary='fill')/ecco_grid.drF
VVELSTAR = grid.diff(ecco_monthly_mean.GM_PsiY, 'Z', boundary='fill')/ecco_grid.drF

GM_PsiXY_diff = grid.diff_2d_vector({'X' : ecco_monthly_mean.GM_PsiX*ecco_grid.dyG,
                                     'Y' : ecco_monthly_mean.GM_PsiY*ecco_grid.dxG},␣
↪boundary = 'fill')
WVELSTAR = (GM_PsiXY_diff['X'] + GM_PsiXY_diff['Y'])/ecco_grid.rA

SALT_at_u = grid.interp(ecco_monthly_mean.SALT, 'X', boundary='extend')
SALT_at_v = grid.interp(ecco_monthly_mean.SALT, 'Y', boundary='extend')
SALT_at_w = grid.interp(ecco_monthly_mean.SALT, 'Z', boundary='extend')

# Remove oceFWflx from WVELMASS
WVELMASS = ecco_monthly_mean.WVELMASS.transpose('time','tile','k_l','j','i')
oceFWflx = ecco_monthly_mean.oceFWflx.assign_coords(k_l=0).expand_dims('k_l').transpose(
↪'time','tile','k_l','j','i')

# Seawater density (kg/m^3)
rhoconst = 1029

oceFWflx = (oceFWflx/rhoconst)
WVELMASS = xr.concat([WVELMASS.sel(k_l=0) + oceFWflx, WVELMASS[:,:,1:]],
                     dim='k_l').transpose('time','tile','k_l','j','i')

# Salinity advective (Eulerian+Bolus) fluxes (psu m^3/s)
ADVx_SLT = (ecco_monthly_mean.UVELMASS+UVELSTAR)*ecco_grid.dyG*ecco_grid.drF*SALT_at_u
#ADVx_SLT = ecco_monthly_mean.UVELMASS*ecco_grid.dyG*ecco_grid.drF*SALT_at_u

ADVy_SLT = (ecco_monthly_mean.VVELMASS+VVELSTAR)*ecco_grid.dxG*ecco_grid.drF*SALT_at_v
#ADVy_SLT = ecco_monthly_mean.VVELMASS*ecco_grid.dxG*ecco_grid.drF*SALT_at_v

ADVr_SLT = (WVELMASS+WVELSTAR)*ecco_grid.rA*SALT_at_w
#ADVr_SLT = WVELMASS*ecco_grid.rA*SALT_at_w

ADVxy_diff = grid.diff_2d_vector({'X' : ADVx_SLT, 'Y' : ADVy_SLT}, boundary = 'fill')
adv_hConvS = (-(ADVxy_diff['X'] + ADVxy_diff['Y']))
adv_vConvS = grid.diff(ADVr_SLT, 'Z', boundary='fill')
```

(continues on next page)

(continued from previous page)

```
G_advection = (adv_hConvS + adv_vConvS)/vol
```

[98]:



1.22.9 Comparison between LHS and RHS of the budget equation

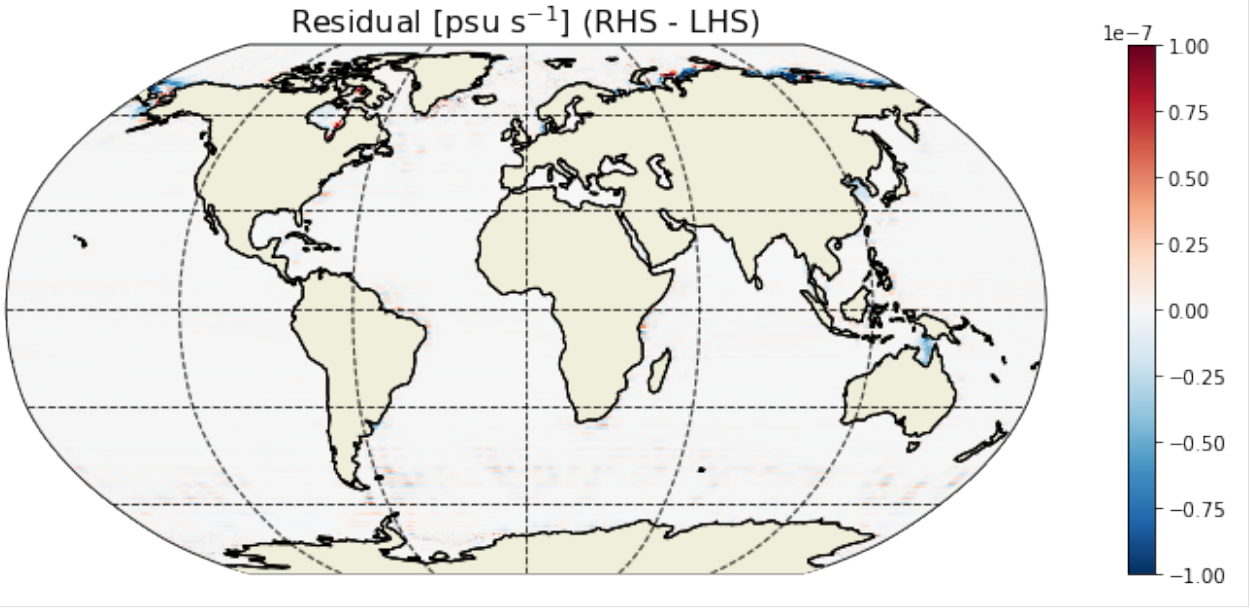
[5]:
```
# Total convergence
ConvSln = G_advection_Sln + G_diffusion_Sln
ConvSln2 = G_advection     + G_diffusion_Sln
```

[6]:
```
# Sum of terms in RHS of equation
rhsSln = ConvSln + G_forcing_Sln
rhsSln2 = ConvSln2 + G_forcing_Sln
```

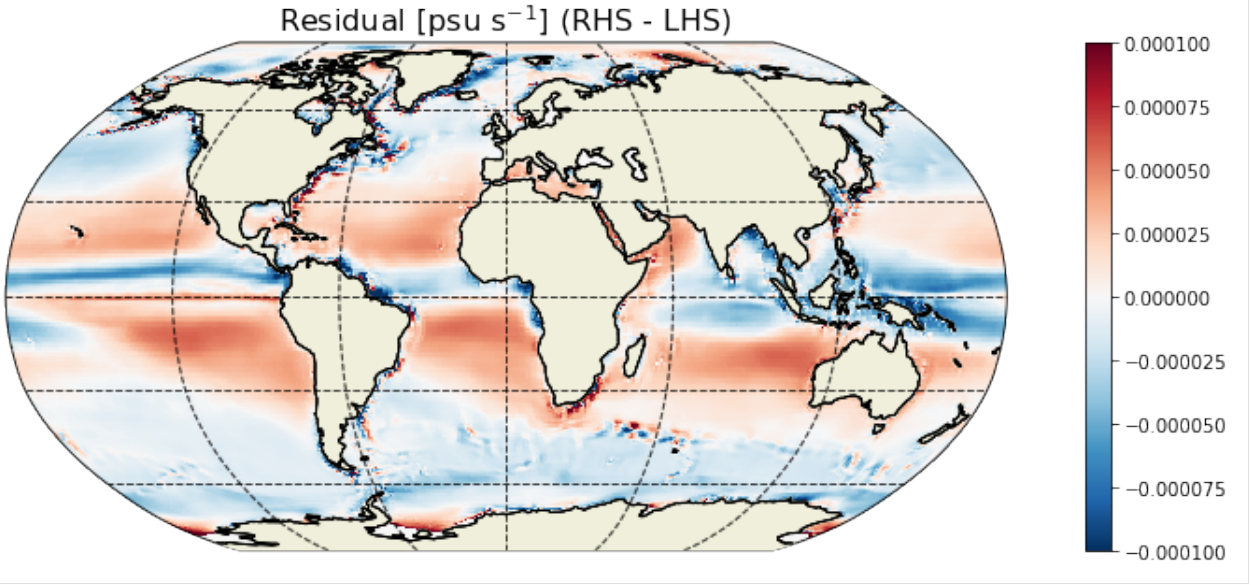
Map of residuals

[7]:
```
resSln =  (G_advection_Sln + G_diffusion_Sln + G_forcing_Sln - G_total_Sln).sum(dim='k').
↪sum(dim='time').compute()
resSln2 = (rhsSln2 - G_total_Sln).sum(dim='k').sum(dim='time').compute()
```

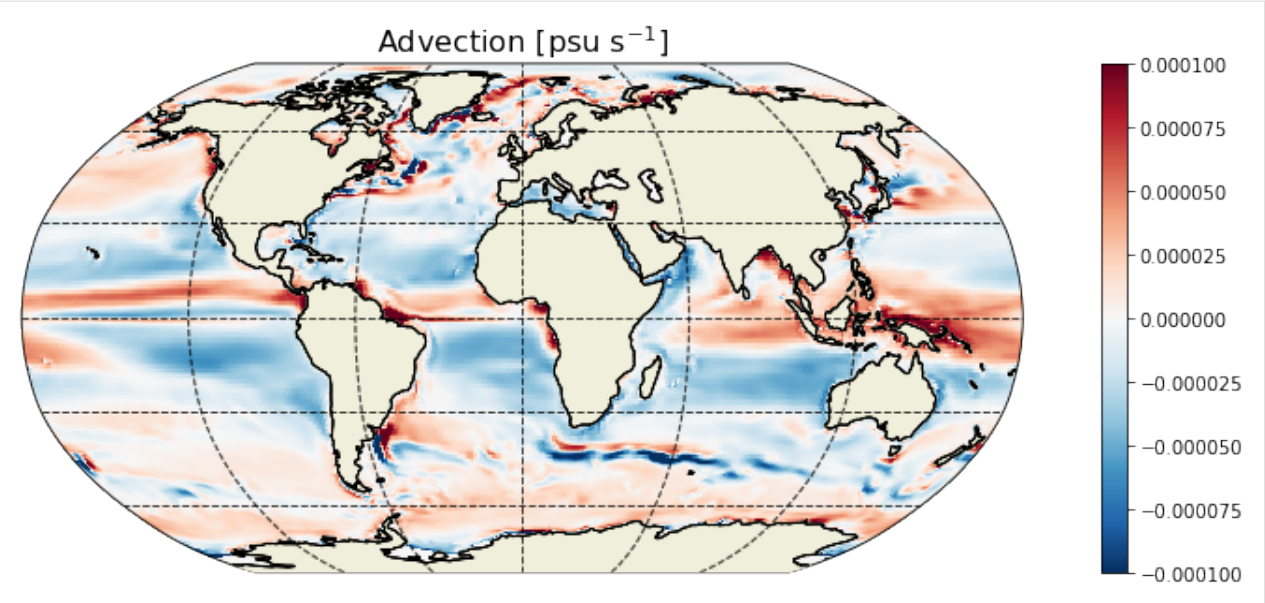
[8]:
```
plt.figure(figsize=(15,5))
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, resSln,
                              cmin=-1e-7, cmax=1e-7, show_colorbar=True, cmap='RdBu_r',
↪dx=0.2, dy=0.2)
plt.title(r'Residual [psu s$^{-1}$] (RHS - LHS)', fontsize=16)
plt.show()
```



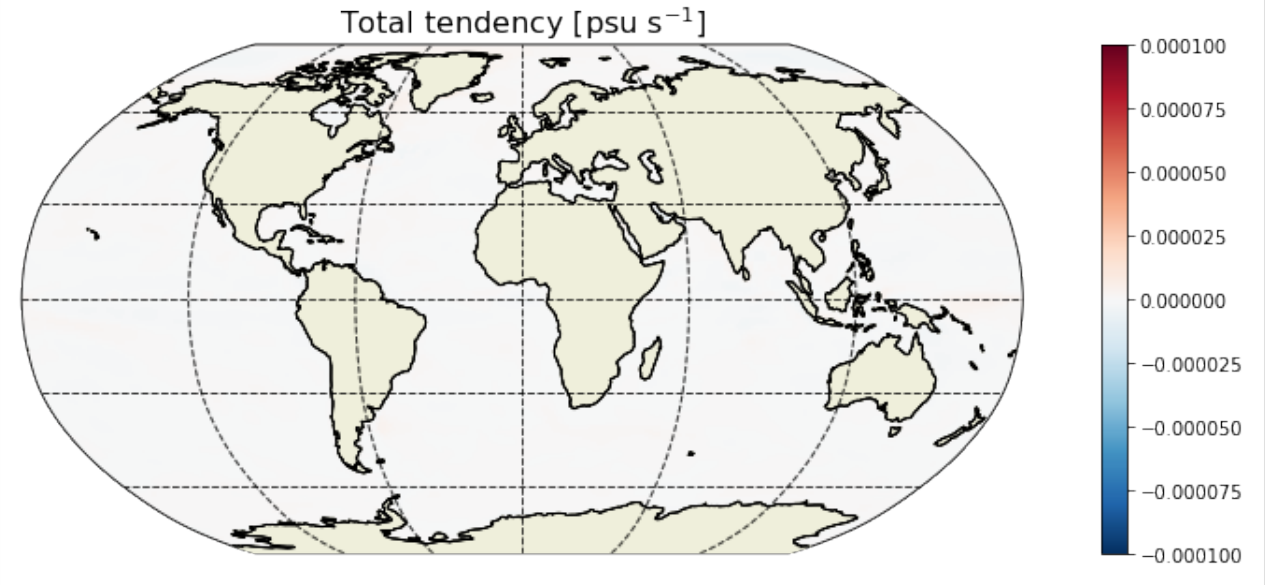
[20]:
```
plt.figure(figsize=(15,5))
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, resSln2,
                              cmin=-1e-4, cmax=1e-4, show_colorbar=True, cmap='RdBu_r',
↪dx=0.2, dy=0.2)
plt.title(r'Residual [psu s$^{-1}$] (RHS - LHS)', fontsize=16)
plt.show()
```



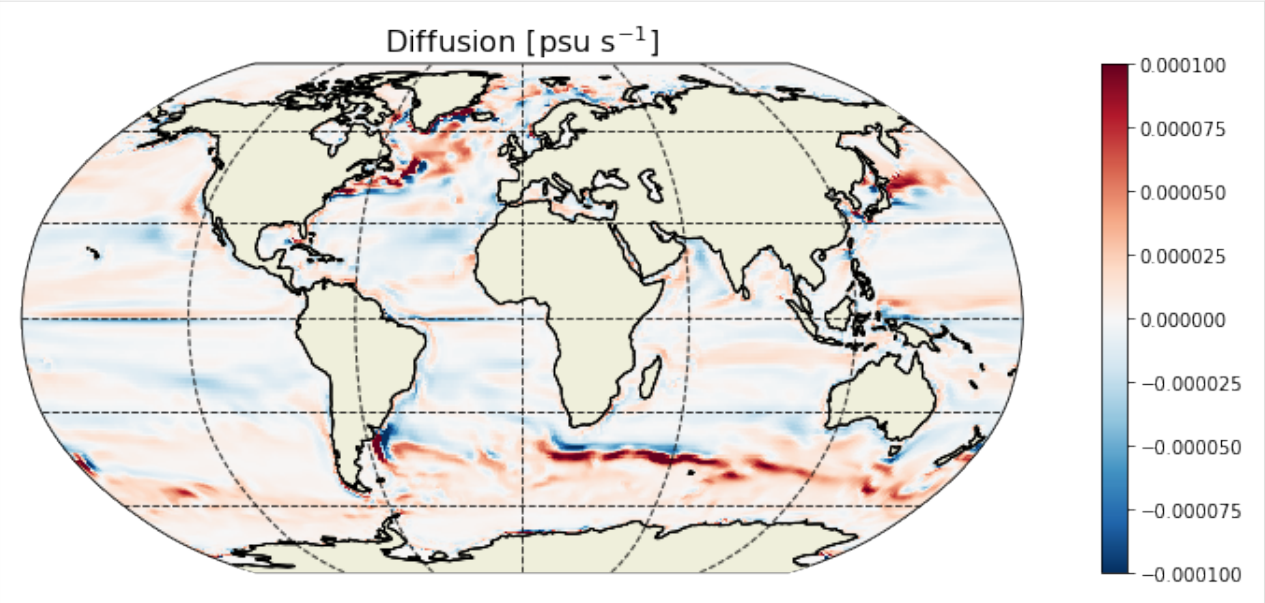
[21]:
```
plt.figure(figsize=(15,5))
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, G_advection,\
                              cmin=-1e-4, cmax=1e-4, show_colorbar=True, cmap='RdBu_r',
↪dx=0.2, dy=0.2)
plt.title(r'Advection [psu s$^{-1}$]', fontsize=16)
plt.show()
```



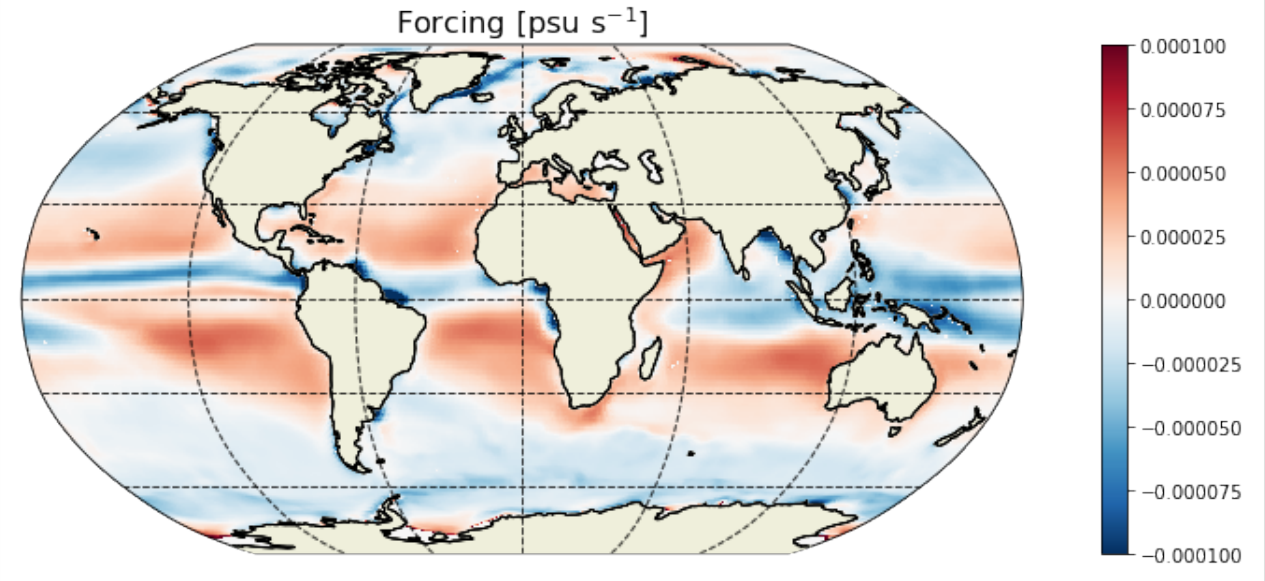
[22]:
```
plt.figure(figsize=(15,5))
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, G_total,
                              cmin=-1e-4, cmax=1e-4, show_colorbar=True, cmap='RdBu_r',
↪dx=0.2, dy=0.2)
plt.title(r'Total tendency [psu s$^{-1}$]', fontsize=16)
plt.show()
```



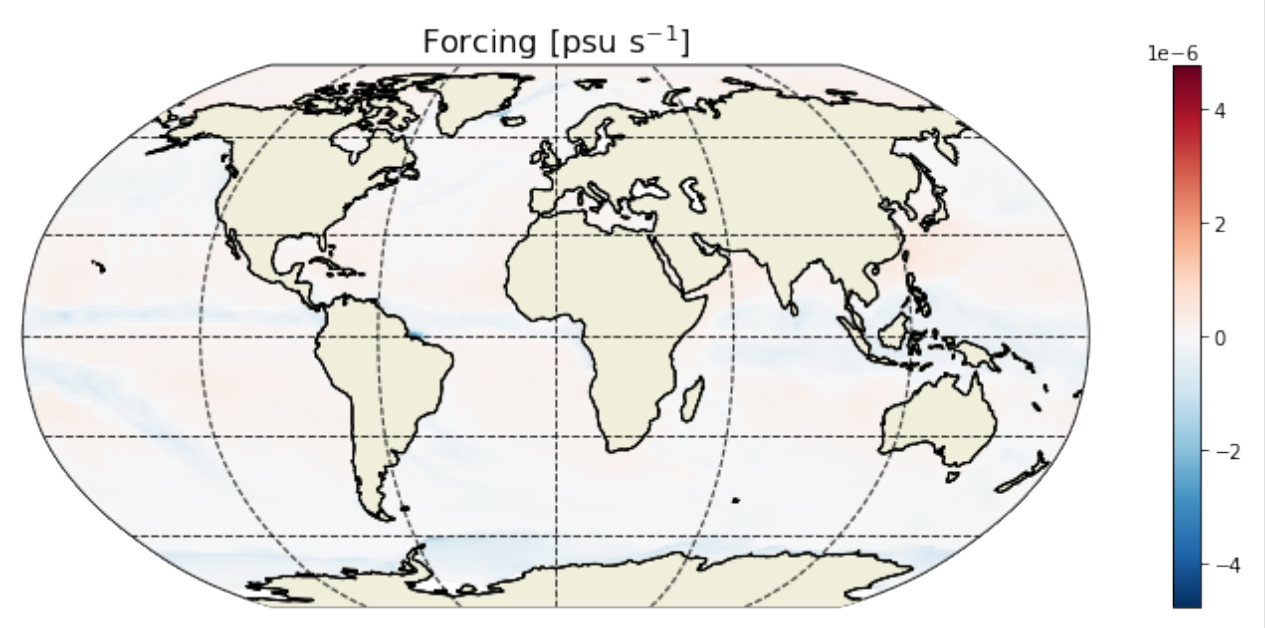
[23]:
```
plt.figure(figsize=(15,5))
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, G_diffusion,
                              cmin=-1e-4, cmax=1e-4,show_colorbar=True, cmap='RdBu_r',
↪dx=0.2, dy=0.2)
plt.title(r'Diffusion [psu s$^{-1}$]', fontsize=16)
plt.show()
```



[24]:
```
plt.figure(figsize=(15,5))
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, G_forcing,
                              cmin=-1e-4, cmax=1e-4,show_colorbar=True, cmap='RdBu_r',
↪dx=0.2, dy=0.2)
plt.title(r'Forcing [psu s$^{-1}$]', fontsize=16)
plt.show()
```



[88]:
```
plt.figure(figsize=(15,5))
ecco.plot_proj_to_latlon_grid(ecco_grid.XC, ecco_grid.YC, G_forcing_Sln[-1,:,0],
                              show_colorbar=True, cmap='RdBu_r',dx=0.2, dy=0.2)
plt.title(r'Forcing [psu s$^{-1}$]', fontsize=16)
plt.show()
```



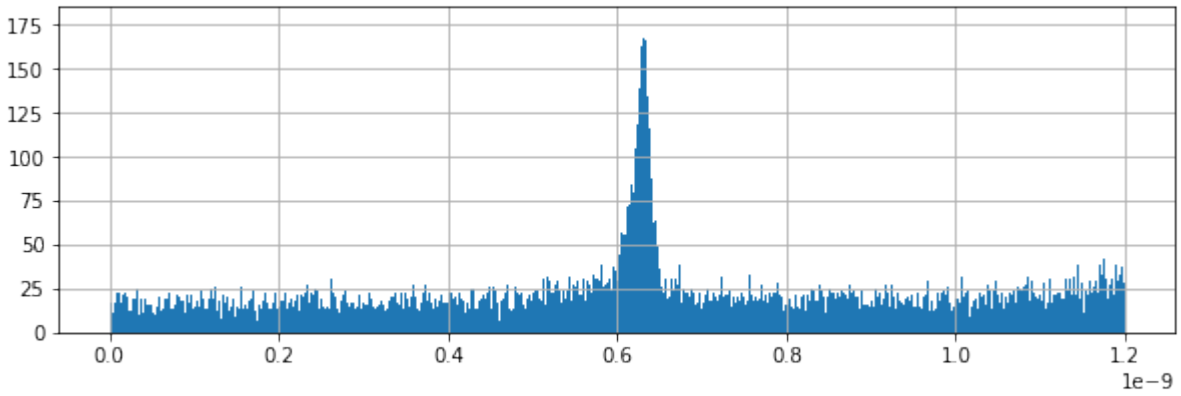
[87]:

[87]:
```
<xarray.DataArray 'G_forcing_Sln' (tile: 13, j: 90, i: 90)>
dask.array<getitem, shape=(13, 90, 90), dtype=float32, chunksize=(13, 90, 90),␣
↪chunktype=numpy.ndarray>
Coordinates:
  * i        (i) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
  * j        (j) int32 0 1 2 3 4 5 6 7 8 9 10 ... 80 81 82 83 84 85 86 87 88 89
    k        int32 0
  * tile     (tile) int32 0 1 2 3 4 5 6 7 8 9 10 11 12
    time     datetime64[ns] 2014-12-16T12:00:00
```

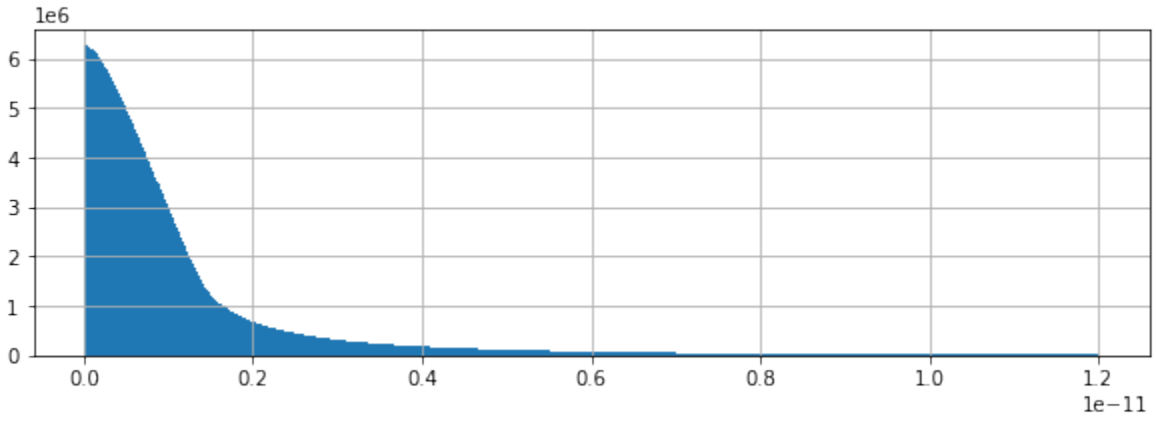
[35]:
```
tmp = np.abs(G_advection_Sln + G_diffusion_Sln + G_forcing_Sln - G_total_Sln).values.
↪ravel()
```

[52]:
```
tmp = res_offline.values.ravel()
```

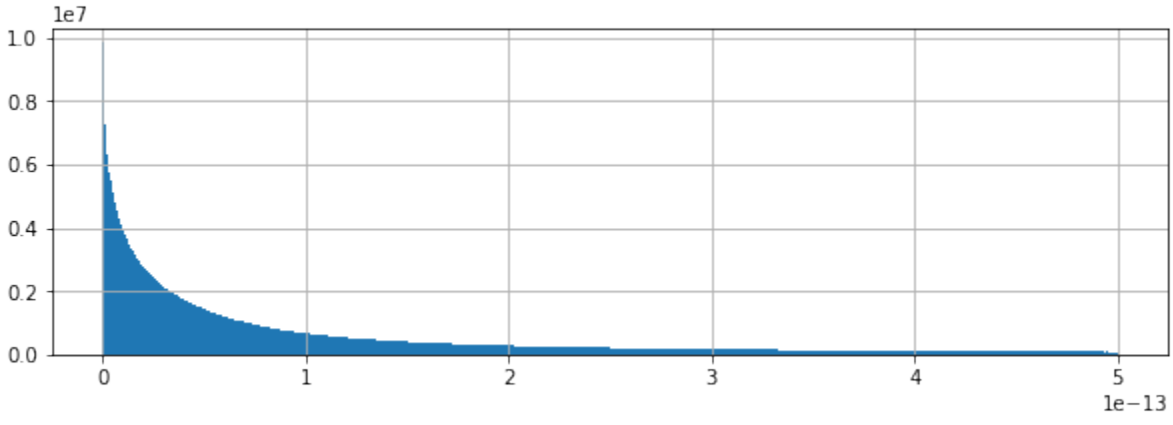
[53]:
```
plt.figure(figsize=(10,3));

plt.hist(tmp[np.nonzero(tmp > 0)],np.linspace(0, 1.2e-9,1000));
plt.grid()
```



[36]:
```
plt.figure(figsize=(10,3));

plt.hist(tmp[np.nonzero(tmp > 0)],np.linspace(0, 1.2e-11,1000));
plt.grid()
```



[12]:



Histogram of residuals

We can look at the distribution of residuals to get a little more confidence.

[25]:
```
from xhistogram.xarray import histogram
```

[48]:
```
res_closed = np.abs(G_advection_Sln + G_diffusion_Sln + G_forcing_Sln - G_total_Sln)
res_closed.name = 'Residual_closed'
```

[50]:
```
res_offline = np.abs(G_advection + G_diffusion_Sln + G_forcing_Sln - G_total_Sln)
res_offline.name = 'Residual_offline'
```

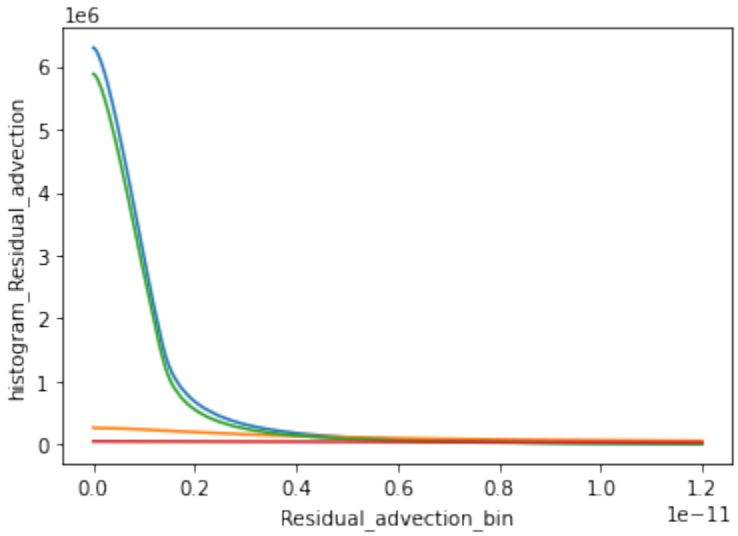
[71]:
```
res_adv = np.abs(G_diffusion_Sln + G_forcing_Sln - G_total_Sln)
res_adv.name = 'Residual_advection'
```

[72]:
```
res_dif = np.abs(G_advection_Sln  + G_forcing_Sln - G_total_Sln)
res_dif.name = 'Residual_diffusion'
```

[73]:
```
res_frc = np.abs(G_advection_Sln + G_diffusion_Sln - G_total_Sln)
res_frc.name = 'Residual_forcing'
```

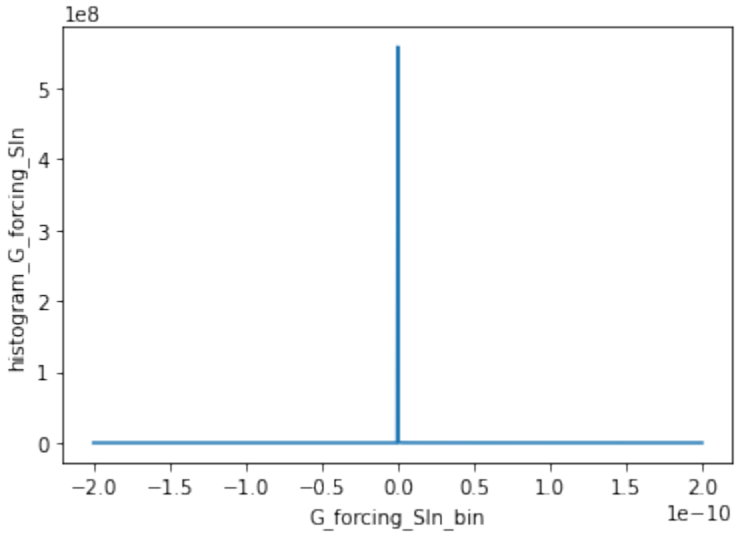
[74]:
```
histogram(res_closed, bins = [np.linspace(0, 1.2e-11,1000)]).plot()
histogram(res_dif, bins = [np.linspace(0, 1.2e-11,1000)]).plot()
histogram(res_frc, bins = [np.linspace(0, 1.2e-11,1000)]).plot()
histogram(res_adv, bins = [np.linspace(0, 1.2e-11,1000)]).plot()
```

[74]:
```
[<matplotlib.lines.Line2D at 0x7f502dc650f0>]
```



[85]:
```
histogram(G_forcing_Sln, bins = [np.linspace(-2e-10, 2e-10,1000)]).plot()
```

[85]:
```
[<matplotlib.lines.Line2D at 0x7f502e71fd30>]
```

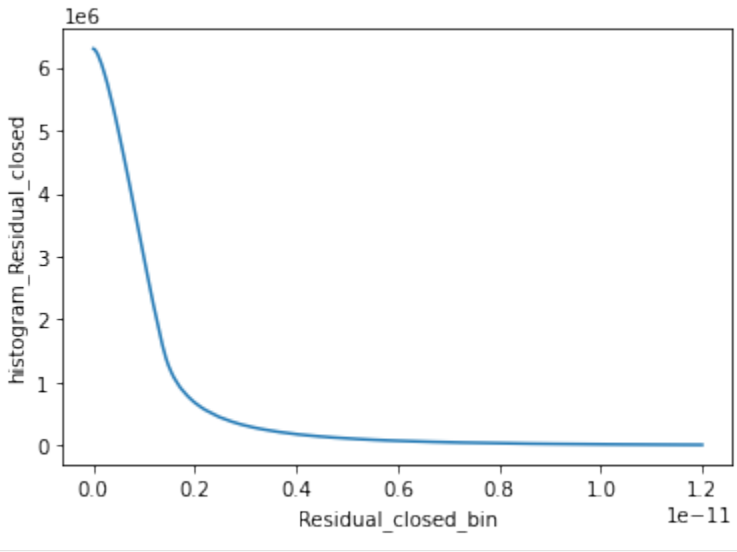


[78]:
```
G_forcing_Sln.max().values
```

[78]:
```
array(3.7624181e-06, dtype=float32)
```

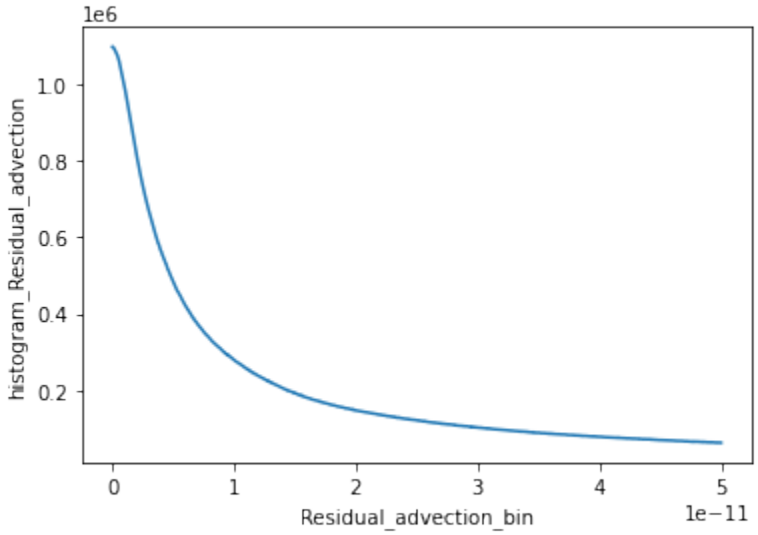
[49]:
```

```

[49]:
```
[<matplotlib.lines.Line2D at 0x7f507b70ca20>]
```



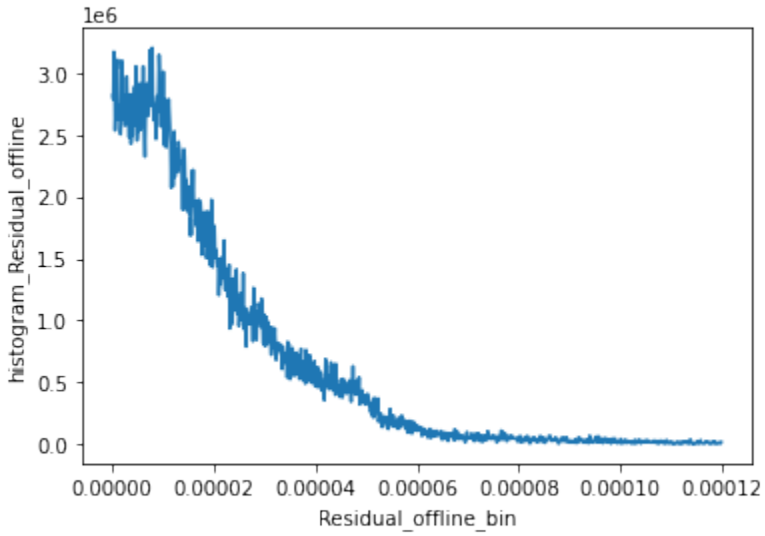
[67]:
```
histogram(res_adv, bins = [np.linspace(0, 5e-11,1000)]).plot()
```

[67]:
```
[<matplotlib.lines.Line2D at 0x7f50e440d978>]
```



[59]:
```
histogram(res_offline, bins = [np.linspace(0, 1.2e-4,1000)]).plot()
```

[59]:
```
[<matplotlib.lines.Line2D at 0x7f507ba18d68>]
```



[ ]:

1.23 Vector calculus in ECCO: The Transport, divergence, vorticity and the Barotropic Vorticity Budget

1.23.1 Using the xgcm and xmitgcm tools

This example calculation uses the xmitgcm and xgcm tool buildt by Rayan Abernathey. We borrow heavily from his documentation, adapting it to the ECCO data: https://xgcm.readthedocs.io/en/latest/example_mitgcm.html. Thanks for the great work Ryan!

1.23.2 This tutorial: Transport, divergence, vorticity, and finally the batotropic *vorticity budget*

This example ses the binary DMS data. We assume you have some background using python. We start from global calculations of *transport*, *divergence*, *vorticity*, and finally the *batotropic vorticity*. We then move on to calculateing vorticity budgets (using the batrotopic vorticity budget). We use this example as it is conceptually accessible, but technically a bit tricky because angles are involved. This means that it is extra important to have a good handle on the grid and where everything is in relation to everything else.

1.23.3 Context for the barotropic vorticity budget

For context, the barotropic vorticity is arrived at taking the momentum equations of the ocean as a thin fluid on a rotating sphere:

$$\partial_t \mathbf{u} + f\mathbf{k} \times \mathbf{u} = -\frac{1}{\rho_0}\nabla p + \frac{1}{\rho_0}\partial_z \tau + \mathbf{a} + \mathbf{b}, \partial_z p = -g\rho, \nabla \cdot \mathbf{v} = 0. \tag{1.21}$$

The pressure, acceleration due to gravity, density and vertical shear stress are denoted p, g, ρ and τ respectively, with ρ_0 the reference density; the three dimensional velocity field $\mathbf{v} = (u, v, w) = (\mathbf{u}, w)$; the unit vector is denoted $\mathbf{k}$; planetary vorticity as a function of latitude ϕ in $f\mathbf{k} = (0, 0, 2\Omega sin\phi)$; the viscous forcing by vertical shear is denoted $\partial_z \tau$; the non-linear terms are $\mathbf{a}$ and the horizontal viscous forcing $\mathbf{b}$. Assuming steady state, the vertical integral from $z = \eta(x, y, t)$ to $z = H(x, y)$ gives

:nbsphinx-math:`begin{equation}
0 = nabla cdot (fmathbf{U})+dfrac{1}{rho_0} nabla p_b times nabla H+dfrac{1}{rho_0}nablatimes mathbf{tau} + nabla times mathrm{textbf{A}}+ nabla times mathrm{textbf{B}},

% label{BV_eq} end{equation}`

the bottom pressure is denoted p_b, $\mathbf{A} = \int_H^\eta \mathbf{a} \mathrm{d}z$ and $\mathbf{B} = \int_H^\eta \mathbf{b} \mathrm{d}z$. The LHS of the equation above is the planetary vorticity advective term, while the RHS of the equation above is the bottom pressure torque, the wind and bottom stress curl, the non-linear torque and the viscous torque, respectively.

For more detail, see Sonnewald et al. 2018: https://github.com/maikejulie/inPrep/blob/master/classificationManuscript.pdf

We start by loading some of the modules we will need. This is a basic step ensuring the tools python will need are available.

[1]:
```
import numpy as np
import xarray
import dask.array as daskarray
from matplotlib import pyplot as plt
%matplotlib inline
from matplotlib.colors import SymLogNorm
from xmitgcm import open_mdsdataset
import xgcm
from mpl_toolkits.basemap import Basemap, cm, shiftgrid
import xarray as xr
```

1.23.4 Using the MDS ECCOv4r2 data

For this work, I modified the "data.diagnostics" file a little, because I wanted some specific outputs. Using xmitgcm, we can read the binary files that my ECCO run produces.

The matadata needs to be available, and for an example download the data here: `bash curl -L -J -O https://ndownloader.figshare.com/files/6494721 tar -xvzf global_oce_llc90.tar.gz`

Loading the binary MDS files, the metadate and creating a "grid" object

For loading the data, we use the comand "open_mdsdataset". The directory given is where the data is, and we have control over how many timesteps or "iters" we extract (here "all"), the geometry (here the LatLongCap "llc") and many other things. The documentation is great!

[3]:
```
#Wondering how the "open_mdsdataset" works? Try looking at the documentation by running␣
↪the command below!
open_mdsdataset?
```

[14]:
```
ds_llc = open_mdsdataset('global_oce_llc90/', iters=8, geometry='llc')
```

We now have an object "ds_llc", where our variables are housed, along with metadata. For more information of the metadata, particularly what points they reference, I made a little table:

Notice that we have the metadata on both tracer and velocity points. This can be in important distinction!

Table modified from http://wwwcvs.mitgcm.org/viewvc/MITgcm/MITgcm_contrib/gael/matlab_class/gcmfaces.pdf?view=co

We can have a look at the content of the object by calling it. To get at the variables, we go "ds_llc.variable". The dimenstions have the extra "face" attribute, which are the different tiles in the ECCO LLC grid.

[11]:
```
#We can have a look at the contents of a dataset object by calling the name.
ds_llc
```

[11]:
```
<xarray.Dataset>
Dimensions:   (face: 13, i: 90, i_g: 90, j: 90, j_g: 90, k: 50, k_l: 50, k_p1: 51, k_u:␣
↪50, time: 240)
Coordinates:
  * k_p1      (k_p1) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...
  * j_g       (j_g) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...
  * i_g       (i_g) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...
  * k         (k) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ...
  * j         (j) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ...
  * k_u       (k_u) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...
  * i         (i) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ...
  * k_l       (k_l) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...
  * face      (face) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
    iter      (time) int64 dask.array<shape=(240,), chunksize=(1,)>
  * time      (time) int64 732 1428 2172 2892 3636 4356 5100 5844 6564 7308 ...
    XC        (face, j, i) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    YC        (face, j, i) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    XG        (face, j_g, i_g) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    YG        (face, j_g, i_g) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    Zl        (k_l) >f4 dask.array<shape=(50,), chunksize=(50,)>
    Zu        (k_u) >f4 dask.array<shape=(50,), chunksize=(50,)>
```

(continues on next page)

(continued from previous page)

```
    Z          (k) >f4 dask.array<shape=(50,), chunksize=(50,)>
    Zp1        (k_p1) >f4 dask.array<shape=(51,), chunksize=(51,)>
    dxC        (face, j, i_g) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    rAs        (face, j_g, i) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    rAw        (face, j, i_g) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    Depth      (face, j, i) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    rA         (face, j, i) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    dxG        (face, j_g, i) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    dyG        (face, j, i_g) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    rAz        (face, j_g, i_g) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    dyC        (face, j_g, i) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    PHrefC     (k) >f4 dask.array<shape=(50,), chunksize=(50,)>
    drC        (k_p1) >f4 dask.array<shape=(51,), chunksize=(51,)>
    PHrefF     (k_p1) >f4 dask.array<shape=(51,), chunksize=(51,)>
    drF        (k) >f4 dask.array<shape=(50,), chunksize=(50,)>
    hFacS      (k, face, j_g, i) >f4 dask.array<shape=(50, 13, 90, 90), chunksize=(1, 1,␣
↪90, 90)>
    hFacC      (k, face, j, i) >f4 dask.array<shape=(50, 13, 90, 90), chunksize=(1, 1, 90,
↪ 90)>
    hFacW      (k, face, j, i_g) >f4 dask.array<shape=(50, 13, 90, 90), chunksize=(1, 1,␣
↪90, 90)>
Data variables:
    PHI_SURF   (time, face, j, i) float32 dask.array<shape=(240, 13, 90, 90),␣
↪chunksize=(1, 1, 90, 90)>
    ETAN       (time, face, j, i) float32 dask.array<shape=(240, 13, 90, 90),␣
↪chunksize=(1, 1, 90, 90)>
    Vm_Advec   (time, k, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Vm_Cori    (time, k, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    AB_gV      (time, k, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Vm_dPHdy   (time, k, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Vm_Ext     (time, k, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    TOTVTEND   (time, k, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    VISrI_Vm   (time, k_l, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    VBotDrag   (time, k, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Vm_Diss    (time, k, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    AB_gU      (time, k, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    VISrI_Um   (time, k_l, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Um_dPHdx   (time, k, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Um_Ext     (time, k, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
```

(continues on next page)

(continued from previous page)

```
    Um_Diss   (time, k, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    TOTUTEND  (time, k, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    UBotDrag  (time, k, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Um_Cori   (time, k, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Um_Advec  (time, k, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
```

1.23.5 Transport and divergence calculations

We start with some transport calculations. This is something where we need to know the cell dimentions etc. so the meta data comes in really handy.

The object we created loading the data "ds_llc" is where our data is stored. We access the fields with the dot syntax (U field is ds_llc.U), and can perform opperations associated with this field using the same syntax (mean across the variable 'time' is ds_llc.U.mean('time')). Calculating transport following:

[15]:
```
u_transport_llc = (ds_llc.U.mean('time')) * ds_llc.dyG * ds_llc.hFacW * ds_llc.drF
v_transport_llc = (ds_llc.V.mean('time')) * ds_llc.dxG * ds_llc.hFacS * ds_llc.drF
```

To plot the data, we employ a slight hack, where we simply twist the faces until the match up. This involves ignoring the Arctic for now (sorry!), and means that the overlap region between the faces is currently not treated very elegantly. This should be fixed in the future, and please refer back to this tutorial/subscribe to our email list to know when we update this!

[20]:
```
u_transport_llc_collated,lat_collated,lon_collated=getField(u_transport_llc[0])
v_transport_llc_collated,lat_collated,lon_collated=getField(v_transport_llc[0])
```

[21]:
```
makeFig(u_transport_llc_collated/1e6, -1, 1, 0.1, "Surface u transport (Sv)", plt.cm.
↪coolwarm, 'surf_u_transp.png')
```

```
/home/maike/anaconda2/lib/python2.7/site-packages/mpl_toolkits/basemap/__init__.py:3608:␣
↪MatplotlibDeprecationWarning: The ishold function was deprecated in version 2.0.
  b = ax.ishold()
/home/maike/anaconda2/lib/python2.7/site-packages/matplotlib/contour.py:967: UserWarning:
↪ The following kwargs were not used by contour: 'shading'
  s)
/home/maike/anaconda2/lib/python2.7/site-packages/mpl_toolkits/basemap/__init__.py:3677:␣
↪MatplotlibDeprecationWarning: axes.hold is deprecated.
    See the API Changes document (http://matplotlib.org/api/api_changes.html)
    for more details.
  ax.hold(b)
/home/maike/anaconda2/lib/python2.7/site-packages/mpl_toolkits/basemap/__init__.py:1623:␣
↪MatplotlibDeprecationWarning: The get_axis_bgcolor function was deprecated in version␣
↪2.0. Use get_facecolor instead.
  fill_color = ax.get_axis_bgcolor()
/home/maike/anaconda2/lib/python2.7/site-packages/mpl_toolkits/basemap/__init__.py:1767:␣
↪MatplotlibDeprecationWarning: The get_axis_bgcolor function was deprecated in version␣
```

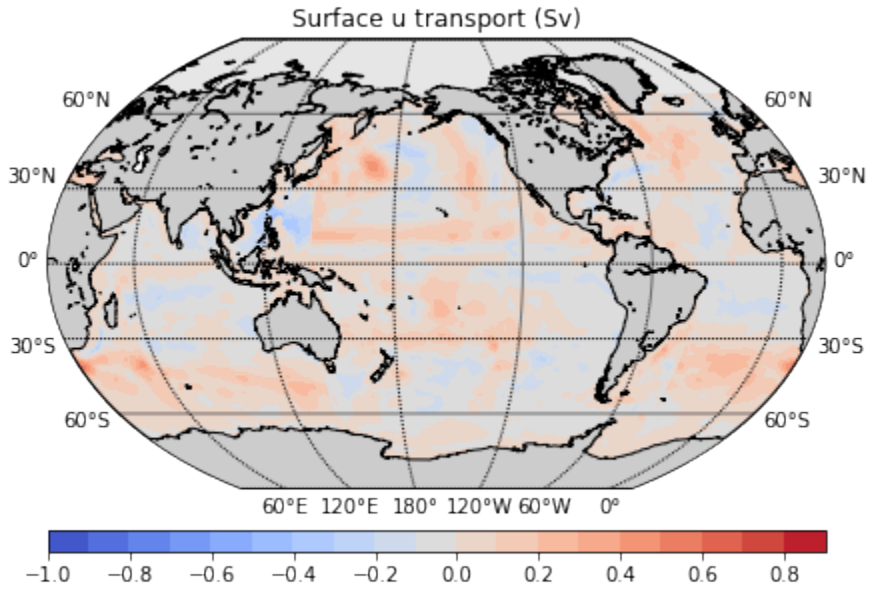
(continues on next page)

(continued from previous page)

```
↪2.0. Use get_facecolor instead.
  axisbgc = ax.get_axis_bgcolor()
```

```
WARNING: x coordinate not montonically increasing - contour plot
may not be what you expect.  If it looks odd, your can either
adjust the map projection region to be consistent with your data, or
(if your data is on a global lat/lon grid) use the shiftgrid
function to adjust the data to be consistent with the map projection
region (see examples/contour_demo.py).
```

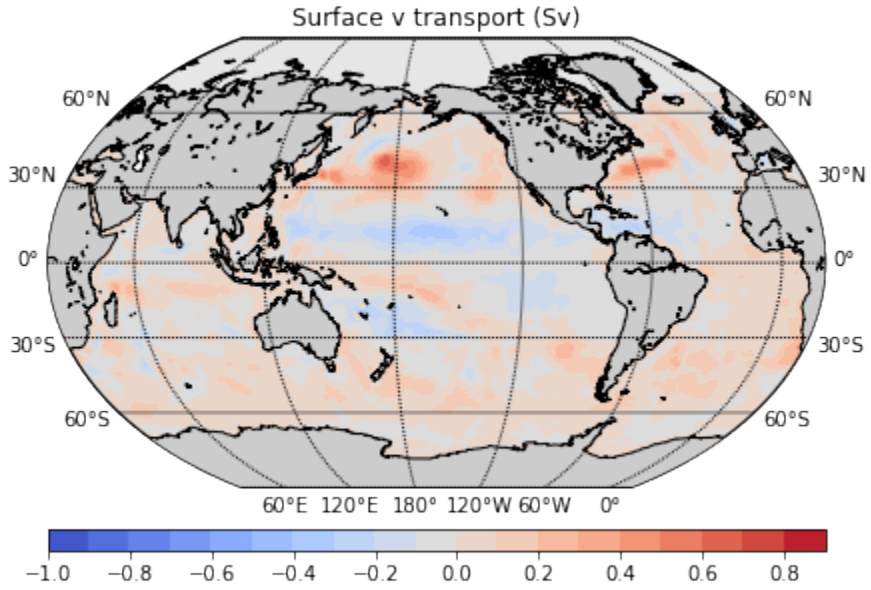
```
<matplotlib.figure.Figure at 0x7f1b68d9bc10>
```



[22]:
```
makeFig(v_transport_llc_collated/1e6, -1, 1, 0.1, "Surface v transport (Sv)", plt.cm.
↪coolwarm, 'surf_v_transp.png')
```

```
WARNING: x coordinate not montonically increasing - contour plot
may not be what you expect.  If it looks odd, your can either
adjust the map projection region to be consistent with your data, or
(if your data is on a global lat/lon grid) use the shiftgrid
function to adjust the data to be consistent with the map projection
region (see examples/contour_demo.py).
```

```
<matplotlib.figure.Figure at 0x7f1d6e762e10>
```



For the divergence, we do something similar.

(from the docs xmitgcm website)

ux+vy+wz=0.

In discrete form, using MITgcm notation, the equation becomes

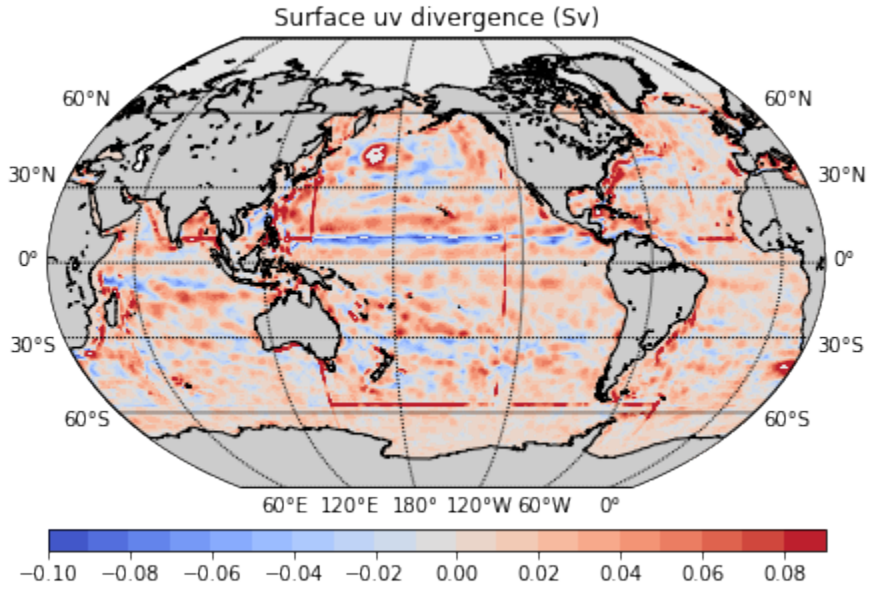
iygrfhwu+jxgrfhsv+kAcw=kAc(EP)r=0

[23]:
```
div_uv_llc = (grid_llc.diff(u_transport_llc, 'X') + grid_llc.diff(v_transport_llc, 'Y'))
```

[24]:
```
div_uv_llc_collated,lat_collated,lon_collated=getField(div_uv_llc[0])
makeFig(div_uv_llc_collated/1e6, -1e-1, 1e-1, 0.1e-1, "Surface uv divergence (Sv)", plt.
↪cm.coolwarm, 'surf_uv_divergence.png')
```

```
WARNING: x coordinate not montonically increasing - contour plot
may not be what you expect.  If it looks odd, your can either
adjust the map projection region to be consistent with your data, or
(if your data is on a global lat/lon grid) use the shiftgrid
function to adjust the data to be consistent with the map projection
region (see examples/contour_demo.py).
```

```
<matplotlib.figure.Figure at 0x7f1aff177d50>
```



1.23.6 Calculating vorticity

We start (again borrowing from the xmitgcm docs) with the vertical component:

=uy+vx.

On the c-grid, a finite-volume representation is given by =(jxcu+iycv)/A .

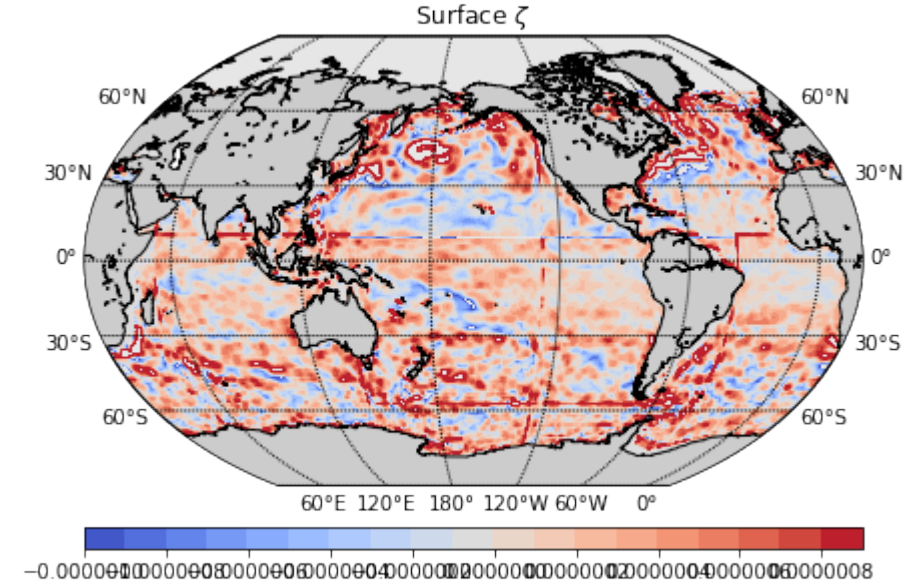
[25]:
```
zeta = (-grid_llc.diff(ds_llc.U.mean('time') * ds_llc.dxC, 'Y') + grid_llc.diff(ds_llc.V.
↪mean('time') * ds_llc.dyC, 'X'))/ds_llc.rAz
```

[26]:
```
zeta_collated,lat_collated,lon_collated=getField(zeta[0])
makeFig(zeta_collated, -1e-6, 1e-6, 0.1e-6, "Surface $\zeta$", plt.cm.coolwarm, 'surf_
↪zeta.png')
```

```
/home/maike/anaconda2/lib/python2.7/site-packages/dask/local.py:271: RuntimeWarning:␣
↪divide by zero encountered in divide
  return func(*args2)
/home/maike/anaconda2/lib/python2.7/site-packages/dask/local.py:271: RuntimeWarning:␣
↪invalid value encountered in divide
  return func(*args2)
```

```
WARNING: x coordinate not montonically increasing - contour plot
may not be what you expect.  If it looks odd, your can either
adjust the map projection region to be consistent with your data, or
(if your data is on a global lat/lon grid) use the shiftgrid
function to adjust the data to be consistent with the map projection
region (see examples/contour_demo.py).
```

```
<matplotlib.figure.Figure at 0x7f1aebcad690>
```



Barotropic vorticity

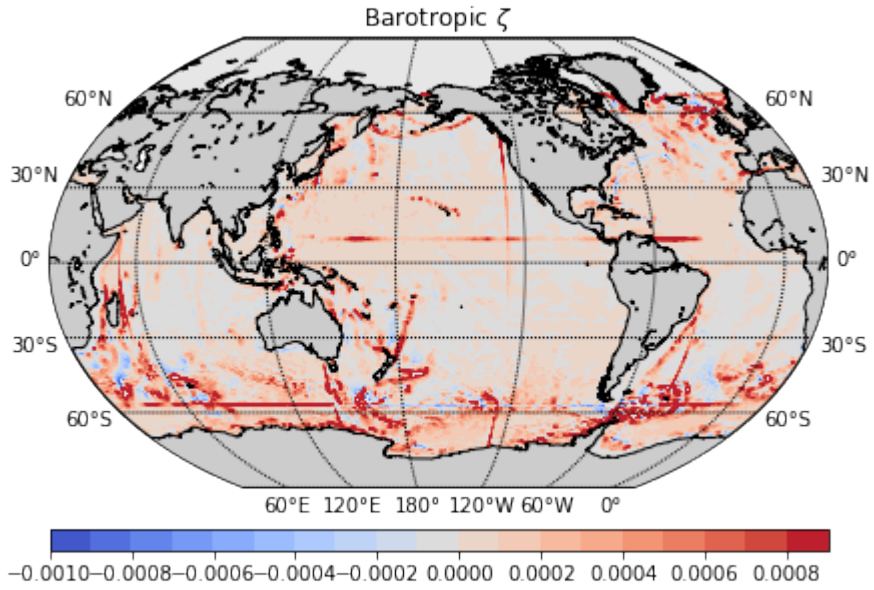
The depth-integrated (barotropic) version of this quanitiy is interesting:

[27]:
```
zeta_bt = (zeta * ds_llc.drF).sum(dim='k')
```

[28]:
```
zeta_bt_collated,lat_collated,lon_collated=getField(zeta_bt)
makeFig(zeta_bt_collated, -1e-3, 1e-3, 0.1e-3, "Barotropic $\zeta$", plt.cm.coolwarm,
↪'bt_zeta.png')
```

```
WARNING: x coordinate not montonically increasing - contour plot
may not be what you expect.  If it looks odd, your can either
adjust the map projection region to be consistent with your data, or
(if your data is on a global lat/lon grid) use the shiftgrid
function to adjust the data to be consistent with the map projection
region (see examples/contour_demo.py).
```

```
<matplotlib.figure.Figure at 0x7f1aebe2be90>
```



1.23.7 The Barotropic vorticity budget

Loading the binary MDS files from my own run

For the budget, I modified the "data.diagnostics" file from an ECCOv4r2 run a little, because I wanted some specific outputs.

For loading the data, we use the comand "open_mdsdataset" again. The directory given is where the data is, and we have control over how many timesteps (here "all"), the geometry (here the LatLongCap "llc") and many other things. Check out the documentation using "open_mdsdataset?", it's great!

[5]:
```
ds_llc = open_mdsdataset('global_oce_llc90/ecco4r2', iters='all', geometry='llc')
```

```
/home/maike/anaconda2/lib/python2.7/site-packages/xmitgcm/utils.py:314: UserWarning: Not␣
↪sure what to do with rlev = L
  warnings.warn("Not sure what to do with rlev = " + rlev)
/home/maike/anaconda2/lib/python2.7/site-packages/xmitgcm/mds_store.py:220:␣
↪FutureWarning: iteration over an xarray.Dataset will change in xarray v0.11 to only␣
↪include data variables, not coordinates. Iterate over the Dataset.variables property␣
↪instead to preserve existing behavior in a forwards compatible manner.
  for vname in ds:
```

[30]:
```
ds_llc
```

[30]:
```
<xarray.Dataset>
Dimensions:   (face: 13, i: 90, i_g: 90, j: 90, j_g: 90, k: 50, k_l: 50, k_p1: 51, k_u:␣
↪50, time: 240)
Coordinates:
  * k_p1      (k_p1) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...
  * j_g       (j_g) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...
  * i_g       (i_g) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...
  * k         (k) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ...
```

(continues on next page)

(continued from previous page)

```
  * j          (j) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ...
  * k_u        (k_u) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...
  * i          (i) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ...
  * k_l        (k_l) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ...
  * face       (face) int64 0 1 2 3 4 5 6 7 8 9 10 11 12
    iter       (time) int64 dask.array<shape=(240,), chunksize=(1,)>
  * time       (time) int64 732 1428 2172 2892 3636 4356 5100 5844 6564 7308 ...
    XC         (face, j, i) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    YC         (face, j, i) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    XG         (face, j_g, i_g) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    YG         (face, j_g, i_g) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    Zl         (k_l) >f4 dask.array<shape=(50,), chunksize=(50,)>
    Zu         (k_u) >f4 dask.array<shape=(50,), chunksize=(50,)>
    Z          (k) >f4 dask.array<shape=(50,), chunksize=(50,)>
    Zp1        (k_p1) >f4 dask.array<shape=(51,), chunksize=(51,)>
    dxC        (face, j, i_g) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    rAs        (face, j_g, i) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    rAw        (face, j, i_g) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    Depth      (face, j, i) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    rA         (face, j, i) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    dxG        (face, j_g, i) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    dyG        (face, j, i_g) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    rAz        (face, j_g, i_g) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    dyC        (face, j_g, i) >f4 dask.array<shape=(13, 90, 90), chunksize=(1, 90, 90)>
    PHrefC     (k) >f4 dask.array<shape=(50,), chunksize=(50,)>
    drC        (k_p1) >f4 dask.array<shape=(51,), chunksize=(51,)>
    PHrefF     (k_p1) >f4 dask.array<shape=(51,), chunksize=(51,)>
    drF        (k) >f4 dask.array<shape=(50,), chunksize=(50,)>
    hFacS      (k, face, j_g, i) >f4 dask.array<shape=(50, 13, 90, 90), chunksize=(1, 1,␣
↪90, 90)>
    hFacC      (k, face, j, i) >f4 dask.array<shape=(50, 13, 90, 90), chunksize=(1, 1, 90,
↪ 90)>
    hFacW      (k, face, j, i_g) >f4 dask.array<shape=(50, 13, 90, 90), chunksize=(1, 1,␣
↪90, 90)>
Data variables:
    PHI_SURF   (time, face, j, i) float32 dask.array<shape=(240, 13, 90, 90),␣
↪chunksize=(1, 1, 90, 90)>
    ETAN       (time, face, j, i) float32 dask.array<shape=(240, 13, 90, 90),␣
↪chunksize=(1, 1, 90, 90)>
    Vm_Advec   (time, k, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Vm_Cori    (time, k, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    AB_gV      (time, k, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Vm_dPHdy   (time, k, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Vm_Ext     (time, k, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    TOTVTEND   (time, k, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    VISrI_Vm   (time, k_l, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
```

(continues on next page)

(continued from previous page)

```
↪chunksize=(1, 1, 1, 90, 90)>
    VBotDrag  (time, k, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Vm_Diss   (time, k, face, j_g, i) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    AB_gU     (time, k, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    VISrI_Um  (time, k_l, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Um_dPHdx  (time, k, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Um_Ext    (time, k, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Um_Diss   (time, k, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    TOTUTEND  (time, k, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    UBotDrag  (time, k, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Um_Cori   (time, k, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
    Um_Advec  (time, k, face, j, i_g) float32 dask.array<shape=(240, 50, 13, 90, 90),␣
↪chunksize=(1, 1, 1, 90, 90)>
```

Now we need to tell xgcm what our axes are like. We create a "grid" object with the dataset as an argument. We have periodic axes, so we create a grid object:

[6]:
```
xgcm.Grid?
```

[6]:
```
grid_llc = xgcm.Grid(ds_llc, periodic=['X', 'Y'])
grid_llc
```

[6]:
```
<xgcm.Grid>
Y Axis (periodic):
  * center   j (90) --> left
  * left     j_g (90) --> center
X Axis (periodic):
  * center   i (90) --> left
  * left     i_g (90) --> center
Z Axis (not periodic):
  * center   k (50) --> left
  * left     k_l (50) --> center
  * outer    k_p1 (51) --> center
  * right    k_u (50) --> center
T Axis (not periodic):
  * center   time (240)
```

We have to do some opperations repeatedly, accomodating the format of the ECCO output, so let's put them is a seperate function:

[7]:
```
def makeVectorECCO_global(vector):
    """Function taking a vector in depth (k) and repeating it to make a gloabl field (k,␣
↪faces, j, i)."""
```

(continues on next page)

(continued from previous page)

```
    return np.rollaxis(np.tile(vector,[13,90,90,1]).transpose(),3,1)

def repeatFieldInTime(field, timesteps, depthToo):
    """Function repeating a field (k, faces, j, i) in time, returning a field with
    dimentions (time, k, face, j, i) if 'depthToo'==1, and otherwise a field with
    dimentions (time, face, j, i)"""
    field_repeated = np.repeat(np.reshape(np.array(field), [1,13,90,90]),240,axis=0)
    if depthToo==1:
        field_repeated = np.repeat(np.reshape(np.array(field), [1,1,13,90,90]),240,
↪axis=0)
        field_repeated = np.repeat(np.reshape(field_repeated, [240,1,13,90,90]),50,
↪axis=1)
    return field_repeated
```

To compute the appropriate Vertical Viscous Flux of U momentum (Implicit part), we vertically integrate. This involves extrapolating the initially 'time, face, j, i' field initially, and using the "diff" opperator along the vertical axis. Using the hFacS and hFacW we take the partial cells into account.

[16]:
```
timesteps = 240
latViscV = ds_llc.VISrI_Vm/repeatFieldInTime(ds_llc.rAs, timesteps, 1)
gridSpacingV = ds_llc.hFacS*makeVectorECCO_global(ds_llc.drF)
diffZlatViscV = grid_llc.diff(latViscV, 'Z', boundary='fill')
latViscV = diffZlatViscV/gridSpacingV
```

[17]:
```
latViscU = ds_llc.VISrI_Um/repeatFieldInTime(ds_llc.rAw, timesteps, 1)
gridSpacingU = ds_llc.hFacW*makeVectorECCO_global(ds_llc.drF)
diffZlatViscV = grid_llc.diff(latViscU, 'Z', boundary='fill')
latViscU = diffZlatViscV/gridSpacingU
```

To have the right pressure field, we need to combine the contribution of the hydrostatic pressure gradient (ϕ_{hyd}) and the surface component (ϕ_{surf}). The Hydrostatic part is readily available, but we need manipulate the η field slightly:

[10]:
```
ETAN=(1/9.81)*ds_llc.PHI_SURF
dETANdx = grid_llc.diff(ETAN, 'X') / repeatFieldInTime(ds_llc.dxC,240,0)
depth_x=np.nansum(ds_llc.hFacW*makeVectorECCO_global(ds_llc.drF), axis=0) #Den e bra
Um_dETANdx=dETANdx*depth_x

dETANdy = grid_llc.diff(ETAN, 'Y') / repeatFieldInTime(ds_llc.dyC,240,0)
depth_y=np.nansum(ds_llc.hFacS*makeVectorECCO_global(ds_llc.drF), axis=0) #Den e bra
Vm_dETANdy=dETANdy*depth_y
```

Barotropic vorticity

This is what I'm personally quite excited by! It's one of those fundamental quantities that are a bit hard to warp your head around, but making them easier to calculate will make it more accessible.

We start (again borrowing from the xmitgcm docs) with the vertical component of the three-dimensional vorticity:

=uy+vx.

On the c-grid, a finite-volume representation is given by =(jxcu+iycv)/A .

We now apply this to the components of the depth integrated momentum equation equation, where we take the 20 year average.

[18]:
```
Um_Ext_int = depthIntVelocity_i(ds_llc.Um_Ext)
Vm_Ext_int = depthIntVelocity_j(ds_llc.Vm_Ext)

UBotDrag_int = depthIntVelocity_i(ds_llc.UBotDrag)
VBotDrag_int = depthIntVelocity_j(ds_llc.VBotDrag)

Um_Cori_int = depthIntVelocity_i(ds_llc.Um_Cori)
Vm_Cori_int = depthIntVelocity_j(ds_llc.Vm_Cori)

Um_dPHdx_int = depthIntVelocity_i(ds_llc.Um_dPHdx)
Vm_dPHdy_int = depthIntVelocity_j(ds_llc.Vm_dPHdy)

Um_Diss_int = depthIntVelocity_i(ds_llc.Um_Diss)
Vm_Diss_int = depthIntVelocity_j(ds_llc.Vm_Diss)

latViscU_int = depthIntVelocity_i(latViscU)
latViscV_int = depthIntVelocity_j(latViscV)

Um_Advec_int = depthIntVelocity_i(ds_llc.Um_Advec)
Vm_Advec_int = depthIntVelocity_j(ds_llc.Vm_Advec)

AB_gU_int = depthIntVelocity_i(ds_llc.AB_gU)
AB_gV_int = depthIntVelocity_j(ds_llc.AB_gV)

TOTUTEND_int = depthIntVelocity_i(ds_llc.TOTUTEND)
TOTVTEND_int = depthIntVelocity_j(ds_llc.TOTVTEND)
```

The surface and bottom stress terms are added together to make up $\nabla \times \tau$

[20]:
```
zeta_wind_int = (-grid_llc.diff(Um_Ext_int.mean(dim='time') * ds_llc.dxC, 'Y') +
                 grid_llc.diff(Vm_Ext_int.mean(dim='time') * ds_llc.dyC, 'X'))/ds_llc.rAz
zeta_bottomDrag_int = (-grid_llc.diff(UBotDrag_int.mean(dim='time') * ds_llc.dxC, 'Y') +
                       grid_llc.diff(VBotDrag_int.mean(dim='time') * ds_llc.dyC, 'X'))/
↪ds_llc.rAz
zeta_tau_bt = zeta_wind_int+zeta_bottomDrag_int
```

The planetary advection term $\nabla \cdot (f\mathbf{U})$

[21]:
```
zeta_cori_bt = (-grid_llc.diff(Um_Cori_int.mean(dim='time') * ds_llc.dxC, 'Y') +
                grid_llc.diff(Vm_Cori_int.mean(dim='time') * ds_llc.dyC, 'X'))/ds_llc.rAz
```

The bottom pressure torque $\nabla p_b \times \nabla H$ as the sum of contributions from ϕ_{hyd} and ϕ_{surf}

[22]:
```
zeta_phiHyd_int = (-grid_llc.diff((Um_dPHdx_int.mean(dim='time')) * ds_llc.dxC, 'Y') +
                   grid_llc.diff((Vm_dPHdy_int.mean(dim='time')) * ds_llc.dyC, 'X'))/ds_
↪llc.rAz
zeta_phiSurf_int = (-grid_llc.diff((Um_dETANdx.mean(dim='time')) * ds_llc.dxC, 'Y') +
                    grid_llc.diff((Vm_dETANdy.mean(dim='time')) * ds_llc.dyC, 'X'))/ds_
↪llc.rAz
zeta_bpt_bt = zeta_phiHyd_int+zeta_phiSurf_int
```

The non-linear terms $\nabla \times \mathbf{A}$

[23]:
```
zeta_Diss_int = (-grid_llc.diff(Um_Diss_int.mean(dim='time') * ds_llc.dxC, 'Y') +
                 grid_llc.diff(Vm_Diss_int.mean(dim='time') * ds_llc.dyC, 'X'))/ds_llc.
↪rAz
zeta_ViscLat_int = (-grid_llc.diff(latViscU_int.mean(dim='time') * ds_llc.dxC, 'Y') +
                    grid_llc.diff(latViscV_int.mean(dim='time') * ds_llc.dyC, 'X'))/ds_
↪llc.rAz
zeta_A_bt = (zeta_Diss_int-zeta_bottomDrag_int)+zeta_ViscLat_int
```

and finally the viscous terms $\nabla \times \mathbf{B}$

[33]:
```
zeta_Adv_int = (-grid_llc.diff(Um_Advec_int.mean(dim='time') * np.array(ds_llc.dxC), 'Y
↪') +
                grid_llc.diff(Vm_Advec_int.mean(dim='time') * np.array(ds_llc.dyC), 'X
↪'))/ds_llc.rAz
zeta_AB_int = (-grid_llc.diff(AB_gU_int.mean(dim='time') * ds_llc.dxC, 'Y') +
               grid_llc.diff(AB_gV_int.mean(dim='time') * ds_llc.dyC, 'X'))/ds_llc.rAz
zeta_B_bt = (zeta_Adv_int-zeta_cori_bt)+zeta_AB_int
```

[32]:
```
zeta_Adv_int.shape, Vm_Advec_int.shape, ds_llc.dxC.shape
```

[32]:
```
((13, 90, 90, 90, 90), (240, 13, 90, 90), (13, 90, 90))
```

[14]:
```
```

We now stitch together the faces for plotting. This an ugly step that wll become depreciated once the exchange routines have been finished; something we are working on that should happen soon!

[26]:
```
zeta_A_bt_collated,lat_collated,lon_collated=getField(zeta_A_bt)
```

```
/home/maike/anaconda2/lib/python2.7/site-packages/dask/local.py:271: RuntimeWarning:␣
↪divide by zero encountered in divide
  return func(*args2)
/home/maike/anaconda2/lib/python2.7/site-packages/dask/local.py:271: RuntimeWarning:␣
↪invalid value encountered in divide
  return func(*args2)
/home/maike/anaconda2/lib/python2.7/site-packages/dask/local.py:271: RuntimeWarning:␣
↪invalid value encountered in subtract
  return func(*args2)
/home/maike/anaconda2/lib/python2.7/site-packages/dask/local.py:271: RuntimeWarning:␣
↪invalid value encountered in add
  return func(*args2)
```

[39]:
```
zeta_B_bt_collated,lat_collated,lon_collated=getField(zeta_B_bt)
```

[40]:
```
zeta_cori_bt_collated,lat_collated,lon_collated=getField(zeta_cori_bt)
```

[41]:
```
zeta_pbt_bt_collated,lat_collated,lon_collated=getField(zeta_bpt_bt)
```

```
/home/maike/anaconda2/lib/python2.7/site-packages/dask/local.py:271: RuntimeWarning:␣
↪invalid value encountered in multiply
  return func(*args2)
/home/maike/anaconda2/lib/python2.7/site-packages/dask/array/numpy_compat.py:46:␣
↪RuntimeWarning: invalid value encountered in divide
  x = np.divide(x1, x2, out)
```

[42]:
```
zeta_tau_bt_collated,lat_collated,lon_collated=getField(zeta_tau_bt)
```

[43]:
```
%store zeta_A_bt_collated
%store zeta_B_bt_collated
%store zeta_cori_bt_collated
%store zeta_tau_bt_collated
%store zeta_bpt_bt_collated
#%store depthAvTorque_av_collated
```

```
Stored 'zeta_A_bt_collated' (ndarray)
Stored 'zeta_B_bt_collated' (ndarray)
Stored 'zeta_cori_bt_collated' (ndarray)
Stored 'zeta_tau_bt_collated' (ndarray)
```

```
UsageError: Unknown variable 'zeta_bpt_bt_collated'
```

[44]:
```
makeFig(zeta_cori_bt_collated, -1e-9, 1e-9, 0.1e-9, "Curl of the coriolis term", plt.cm.
↪coolwarm, 'zeta_cori_bt.png')
```

```
/home/maike/anaconda2/lib/python2.7/site-packages/mpl_toolkits/basemap/__init__.py:3608:␣
↪MatplotlibDeprecationWarning: The ishold function was deprecated in version 2.0.
  b = ax.ishold()
```

```
WARNING: x coordinate not montonically increasing - contour plot
may not be what you expect.  If it looks odd, your can either
adjust the map projection region to be consistent with your data, or
(if your data is on a global lat/lon grid) use the shiftgrid
function to adjust the data to be consistent with the map projection
region (see examples/contour_demo.py).
```

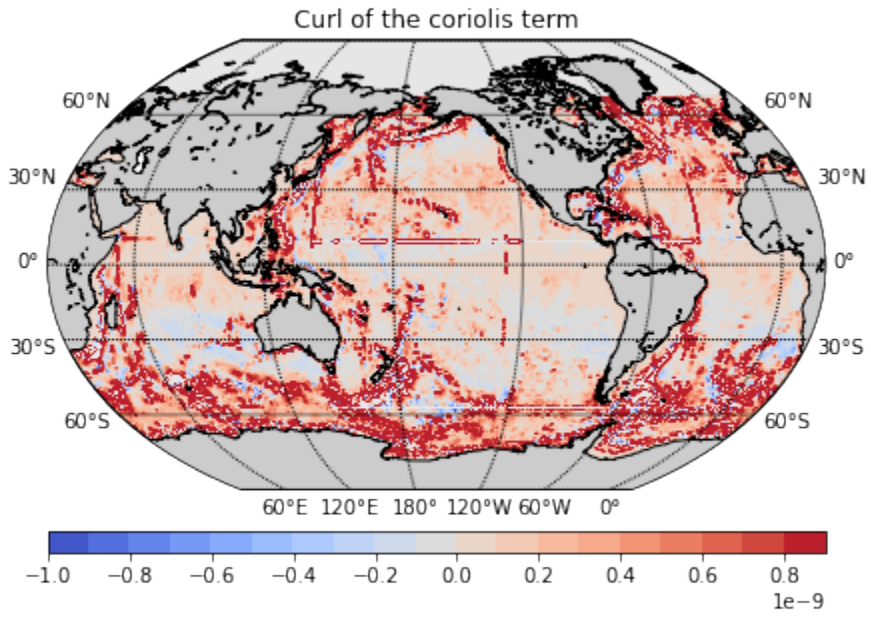
```
/home/maike/anaconda2/lib/python2.7/site-packages/matplotlib/contour.py:967: UserWarning:
↪ The following kwargs were not used by contour: 'shading'
  s)
/home/maike/anaconda2/lib/python2.7/site-packages/mpl_toolkits/basemap/__init__.py:3677:␣
↪MatplotlibDeprecationWarning: axes.hold is deprecated.
    See the API Changes document (http://matplotlib.org/api/api_changes.html)
    for more details.
  ax.hold(b)
/home/maike/anaconda2/lib/python2.7/site-packages/mpl_toolkits/basemap/__init__.py:1623:␣
↪MatplotlibDeprecationWarning: The get_axis_bgcolor function was deprecated in version␣
↪2.0. Use get_facecolor instead.
```

(continues on next page)

(continued from previous page)

```
  fill_color = ax.get_axis_bgcolor()
/home/maike/anaconda2/lib/python2.7/site-packages/mpl_toolkits/basemap/__init__.py:1767:␣
↪MatplotlibDeprecationWarning: The get_axis_bgcolor function was deprecated in version␣
↪2.0. Use get_facecolor instead.
  axisbgc = ax.get_axis_bgcolor()
```

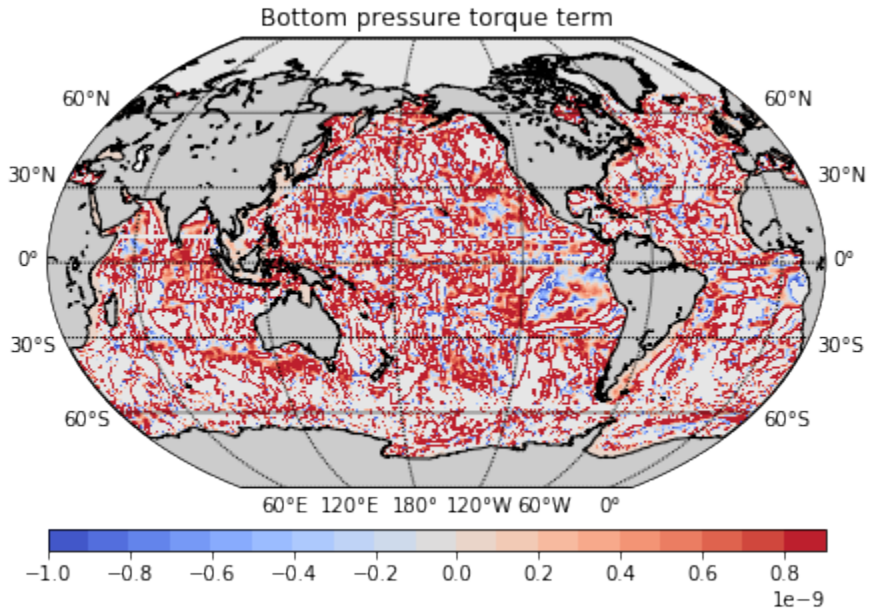
```
<matplotlib.figure.Figure at 0x7f9d44cadd50>
```



[45]:
```
makeFig(zeta_pbt_bt_collated, -1e-9, 1e-9, 0.1e-9, "Bottom pressure torque term", plt.cm.
↪coolwarm, 'zeta_bpt_bt.png')
```

```
WARNING: x coordinate not montonically increasing - contour plot
may not be what you expect.  If it looks odd, your can either
adjust the map projection region to be consistent with your data, or
(if your data is on a global lat/lon grid) use the shiftgrid
function to adjust the data to be consistent with the map projection
region (see examples/contour_demo.py).
```

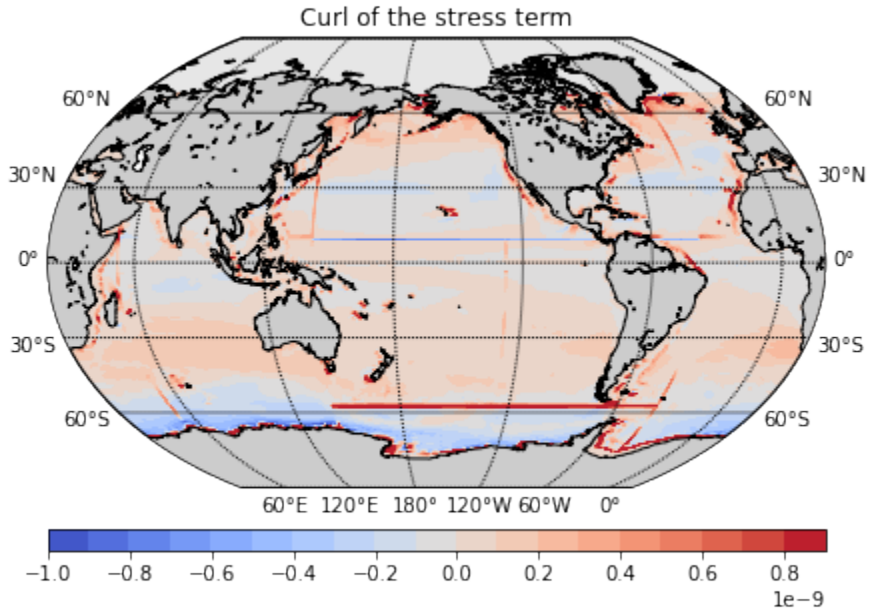
```
<matplotlib.figure.Figure at 0x7f9b94364650>
```



[46]:
```
makeFig(zeta_tau_bt_collated, -1e-9, 1e-9, 0.1e-9, "Curl of the stress term", plt.cm.
↪coolwarm, 'zeta_tau_bt.png')
```

```
WARNING: x coordinate not montonically increasing - contour plot
may not be what you expect.  If it looks odd, your can either
adjust the map projection region to be consistent with your data, or
(if your data is on a global lat/lon grid) use the shiftgrid
function to adjust the data to be consistent with the map projection
region (see examples/contour_demo.py).
```

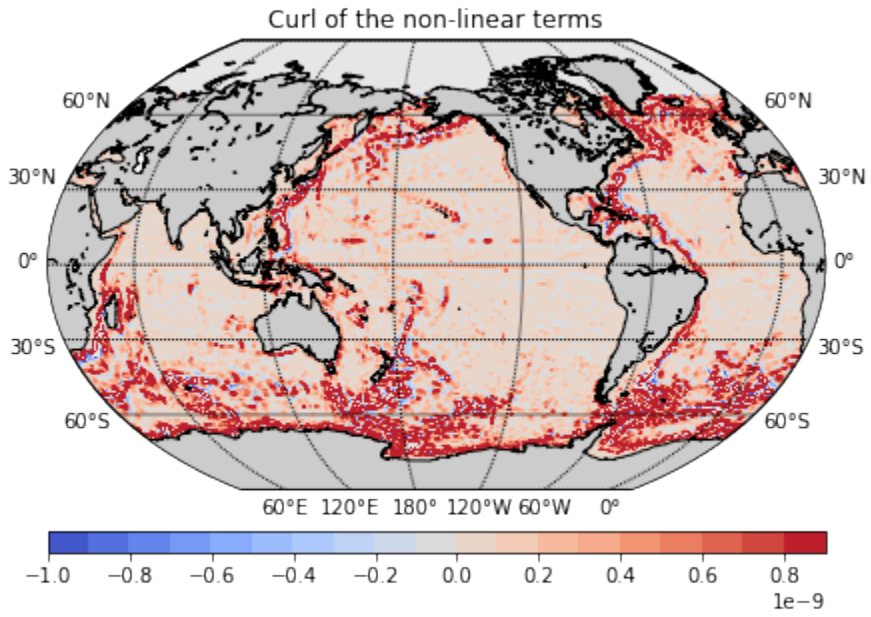
```
<matplotlib.figure.Figure at 0x7f9b99b42d50>
```



[47]:
```
makeFig(zeta_A_bt_collated, -1e-9, 1e-9, 0.1e-9, "Curl of the non-linear terms", plt.cm.
↪coolwarm, 'zeta_a_bt.png')
```

```
WARNING: x coordinate not montonically increasing - contour plot
may not be what you expect.  If it looks odd, your can either
adjust the map projection region to be consistent with your data, or
(if your data is on a global lat/lon grid) use the shiftgrid
function to adjust the data to be consistent with the map projection
region (see examples/contour_demo.py).
```

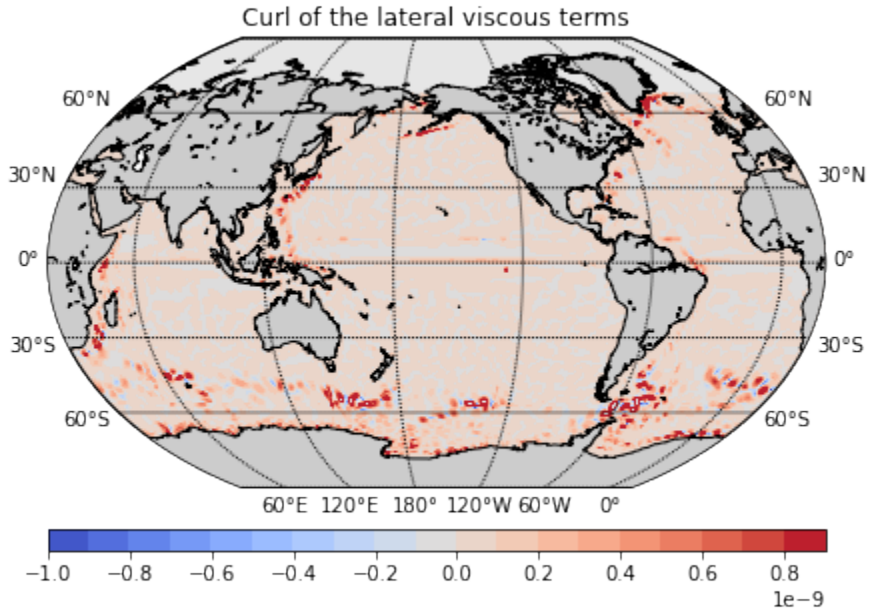
```
<matplotlib.figure.Figure at 0x7f9ba0bf9450>
```



[48]:
```
makeFig(zeta_B_bt_collated, -1e-9, 1e-9, 0.1e-9, "Curl of the lateral viscous terms",␣
↪plt.cm.coolwarm, 'zeta_b_bt.png')
```

```
WARNING: x coordinate not montonically increasing - contour plot
may not be what you expect.  If it looks odd, your can either
adjust the map projection region to be consistent with your data, or
(if your data is on a global lat/lon grid) use the shiftgrid
function to adjust the data to be consistent with the map projection
region (see examples/contour_demo.py).
```

```
<matplotlib.figure.Figure at 0x7f9b93f9f8d0>
```



[49]:
```
makeFig(zetaBT_collated, -1e-9, 1e-9, 0.1e-9, "Curl of the barotropic velocity field",␣
↪plt.cm.coolwarm, 'test_zeta_bt.png')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-49-872296fb8bb2> in <module>()
----> 1 makeFig(zetaBT_collated, -1e-9, 1e-9, 0.1e-9, "Curl of the barotropic velocity␣
↪field", plt.cm.coolwarm, 'test_zeta_bt.png')

NameError: name 'zetaBT_collated' is not defined
```

For my own purposes, I hacked together a globe for plotting, just rotating and flipping the "faces". In my workflow, these exsist as a little library I load. I added them here as an example of what-not-to-do. It's very tedious.

[11]:
```
def makeVectorECCO_global(vector):
    """Function taking a vector in depth (k) and repeating it to make a gloabl field (k,␣
↪faces, j, i)."""
    return np.rollaxis(np.tile(vector,[13,90,90,1]).transpose(),3,1)

def repeatFieldInTime(field, timesteps, depthToo):
    """Function repeating a field (k, faces, j, i) in time, returning a field with
    dimentions (time, k, face, j, i) if 'depthToo'==1, and otherwise a field with
    dimentions (time, face, j, i)"""
    field_repeated = np.repeat(np.reshape(np.array(field), [1,13,90,90]),240,axis=0)
    if depthToo==1:
        field_repeated = np.repeat(np.reshape(np.array(field), [1,1,13,90,90]),240,
↪axis=0)
        field_repeated = np.repeat(np.reshape(field_repeated, [240,1,13,90,90]),50,
↪axis=1)
    return field_repeated

def depthIntVelocity_i(field):
```

(continues on next page)

(continued from previous page)

```
    """Time mean and depth integrated version of the given field on tracer point."""
    return (field * np.array(ds_llc.drF * ds_llc.hFacW)).sum(dim='k')


def depthIntVelocity_j(field):
    """Time mean and depth integrated version of the given field on tracer point."""
    return (field * np.array(ds_llc.drF * ds_llc.hFacS)).sum(dim='k')
```

[2]:
```
def getField(field):
    """Rotating the different faces so they fit together. Takes a field on the
    ECCOv4r2 llc grid with dimensions (face, i, j), and returns a field with (90*4,90*3).
    Unfortunately we ingore the Arctic..."""
    field_collated=np.zeros([90*4,90*3])
    lon_collated=np.zeros([90*4,90*3])
    lat_collated=np.zeros([90*4,90*3])
    x, y, nr = 3, 2, 4
    field_collated, lat_collated, lon_collated = addToMatrix(field_collated, lon_
↪collated, lat_collated, x, y, rot90Case(getFace(nr, field)))
    x, y, nr = 3, 1, 5
    field_collated, lat_collated, lon_collated = addToMatrix(field_collated, lon_
↪collated, lat_collated, x, y, rot90Case(getFace(nr, field)))
    x, y, nr = 3, 0, 6
    field_collated, lat_collated, lon_collated = addToMatrix(field_collated, lon_
↪collated, lat_collated, x, y, rot90Case(getFace(nr, field)))
    x, y, nr = 0, 2, 1
    field_collated, lat_collated, lon_collated = addToMatrix(field_collated, lon_
↪collated, lat_collated, x, y, rot90Case(getFace(nr, field)))
    x, y, nr = 0, 1, 2
    field_collated, lat_collated, lon_collated = addToMatrix(field_collated, lon_
↪collated, lat_collated, x, y, rot90Case(getFace(nr, field)))
    x, y, nr = 0, 0, 3
    field_collated, lat_collated, lon_collated = addToMatrix(field_collated, lon_
↪collated, lat_collated, x, y, rot90Case(getFace(nr, field)))
    x, y, nr = 1, 0, 11
    field_collated, lat_collated, lon_collated = addToMatrix(field_collated, lon_
↪collated, lat_collated, x, y, fliplrCase(getFace(nr, field)))
    x, y, nr = 1, 1, 12
    field_collated, lat_collated, lon_collated = addToMatrix(field_collated, lon_
↪collated, lat_collated, x, y, fliplrCase(getFace(nr, field)))
    x, y, nr = 1, 2, 13
    field_collated, lat_collated, lon_collated = addToMatrix(field_collated, lon_
↪collated, lat_collated, x, y, fliplrCase(getFace(nr, field)))
    x, y, nr = 2, 0, 8
    field_collated, lat_collated, lon_collated = addToMatrix(field_collated, lon_
↪collated, lat_collated, x, y, fliplrCase(getFace(nr, field)))
    x, y, nr = 2, 1, 9
    field_collated, lat_collated, lon_collated = addToMatrix(field_collated, lon_
↪collated, lat_collated, x, y, fliplrCase(getFace(nr, field)))
    x, y, nr = 2, 2, 10
    field_collated, lat_collated, lon_collated = addToMatrix(field_collated, lon_
↪collated, lat_collated, x, y, fliplrCase(getFace(nr, field)))
    return field_collated, lat_collated, lon_collated
```

(continues on next page)

(continued from previous page)

[3]:
```
#We need to glue together the "faces" of ECCOv4
def getFace(faceNr, field):
    """Extracting the face to be treated, along with the matching lat and long."""
    f =xr.open_dataset('/home/maike/Documents/ECCO4_rel2/PHIBOT/PHIBOT.00{:02d}.nc'.
↪format(faceNr))
    fieldRet = field[faceNr-1,:,:]
    lat = f.lat
    lon = f.lon
    return fieldRet, lat, lon

def rot90Case((fieldRet, lat, lon)):
    """The case where a face needs to be rotated."""
    fieldRet=np.flipud(np.rot90(fieldRet,3))
    lon=np.flipud(np.rot90(lon,3))
    lat=np.flipud(np.rot90(lat,3))
    return fieldRet, lat, lon

def fliplrCase((fieldRet, lat, lon)):
    """The case where the face needs to be flipped left to right."""
    fieldRet=np.flipud(fieldRet)
    lon=np.flipud(lon)
    lat=np.flipud(lat)
    return fieldRet, lat, lon

def addToMatrix(field_collated, lon_collated, lat_collated, x, y, (fieldRet, lat, lon)):
    """Adding the field to the overall matrix."""
    field_collated[90*x:90*(x+1), 90*y:90*(y+1)]=fieldRet
    lon_collated[90*x:90*(x+1), 90*y:90*(y+1)]=lon
    lat_collated[90*x:90*(x+1), 90*y:90*(y+1)]=lat
    return field_collated, lat_collated, lon_collated
```

Similarly, I plot using the basemap package. It's worked for me. I rudely ignotre the Arctic though, because interpolation is hard. This is pretty hard-coded to my *personal* workflow, so really just intended for inspirations.

[4]:
```
def makeFig(data, cMin, cMax, cStep, title, ColMap, saveName):
    """
    Example of a plot using latitude and longitude. The 'data' is the field
    we want to plot, cMin, cMax and cStep are the min, max and step of the colurbar.
    Give the title of the plot as 'Plot title', the colour map needs to be specified
    next, and lastly the name to save the plot under e.g, 'title.png'.

    Example usage:
    makeFig(zeta_tau_bt_collated, -1e-9, 1e-9, 0.1e-9, "Curl of the stress term", plt.cm.
↪coolwarm, 'zeta_tau_bt.png')
    """
    plt.figure(figsize=(20,12))
    lons = lon_collated[:,1]
    lats = lat_collated[1,:]
    llons, llats = np.meshgrid(lons,lats)
```

(continues on next page)

(continued from previous page)

```
    fig = plt.figure()
    ax = fig.add_axes([0.05,0.05,0.9,0.9])
    m = Basemap(projection='kav7',lon_0=200,resolution='l')
    im1 = m.contourf(llons,llats,np.fliplr(np.rot90(data,3)),np.arange(cMin, cMax,␣
↪cStep),shading='flat',cmap=ColMap,latlon=True)
    m.drawmapboundary(fill_color='0.9')
    m.drawparallels(np.arange(-90.,99.,30.),labels=[1,1,0,1])
    m.drawmeridians(np.arange(-180.,180.,60.),labels=[1,1,0,1])
    m.drawcoastlines()
    m.fillcontinents()
    cb = m.colorbar(im1,"bottom", size="5%", pad="9%")
    ax.set_title(title)
    plt.savefig(saveName, format='png', dpi=500, bbox_inches='tight')
    plt.show()
```

[ ]:

[ ]:

1.24 ECCOv4 budgets

Evaluating budgets in the ECCOv4 model run using xgcm. Currently, calculations are done only for one face (here `face = 1`).

This notebook is based on the calculations and MATLAB code in evaluating_budgets_in_eccov4r3.pdf by Christopher G. Piecuch (ftp://ecco.jpl.nasa.gov/Version4/Release3/doc/evaluating_budgets_in_eccov4r3.pdf).

[1]:
```
import numpy as np
import xarray as xr
import pandas as pd

from xmitgcm import open_mdsdataset
import xgcm

import matplotlib.pyplot as plt
%matplotlib inline
```

1.24.1 Load datasets

Note: Model output is saved into files with 4 different prefixes depending on whether the variable is averaged (`ave`) or a snapshot (`snp`) and whether it is 2D or 3D. Reference date and time step are defined to get appropriate time points. For this example the ECCOv4r2 solution has been run only for the first year (1992).

[2]:
```
ds_ave = open_mdsdataset('/rigel/ocp/users/jt2796/ECCO_v4_r2/run_budg3d_1yr/',
                         delta_t=3600, ref_date='1991-12-15 12:0:0', geometry='llc',
                         prefix=['monave2d','monave3d'])
```

```
/rigel/home/jt2796/miniconda/envs/default/lib/python2.7/site-packages/xmitgcm-0.2.1-py2.
↪7.egg/xmitgcm/utils.py:314: UserWarning: Not sure what to do with rlev = L
  warnings.warn("Not sure what to do with rlev = " + rlev)
```

[3]:
```
ds_snp = open_mdsdataset('/rigel/ocp/users/jt2796/ECCO_v4_r2/run_budg3d_1yr/',
                         delta_t=3600, ref_date='1992-1-1 12:0:0', geometry='llc',
                         prefix=['monsnp2d','monsnp3d'])
```

Adjust time axis (optional)

Note: This code can be run to update the time axis to a certain common format. It is used here to be able to compare budget terms with other datasets (e.g., standard output from ECCOv4 netcdf files). Here, the time axis is defined for monthly averages to have time points always at the same day in the middle of the month (day=15).

[4]:
```
ds_ave['time'] = pd.date_range(start='1992-01-15', periods=2*12, freq='SMS')[::2]
```

Geothermal flux

[5]:
```
geoflx = np.fromfile('/rigel/ocp/users/jt2796/ECCO_v4_r2/geothermalFlux.bin', dtype=np.
↪float32)
```

Note: Geothermal flux dataset needs to be saved as an xarray data array with the same format as the model output. In order to reformat the loaded data array the byte-ordering needs to be changed.

[6]:
```
# Data and type endianness don't match. Change data to match dtype and reshape to 1d
geoflx = geoflx.byteswap().reshape([105300,1])
```

[7]:
```
# Reshape data for each face and save as xarray data array in LLC format
geoflx00 = xr.DataArray(geoflx[:8100,0].reshape([90,90]),coords=[np.arange(0,90,1),np.
↪arange(0,90,1)],
                        dims=['j','i'])
geoflx01 = xr.DataArray(geoflx[8100:16200,0].reshape([90,90]),coords=[np.arange(0,90,1),
↪np.arange(0,90,1)],
                        dims=['j','i'])
geoflx02 = xr.DataArray(geoflx[16200:24300,0].reshape([90,90]),coords=[np.arange(0,90,1),
↪np.arange(0,90,1)],
                        dims=['j','i'])
geoflx03 = xr.DataArray(geoflx[24300:32400,0].reshape([90,90]),coords=[np.arange(0,90,1),
↪np.arange(0,90,1)],
                        dims=['j','i'])
geoflx04 = xr.DataArray(geoflx[32400:40500,0].reshape([90,90]),coords=[np.arange(0,90,1),
↪np.arange(0,90,1)],
                        dims=['j','i'])
geoflx05 = xr.DataArray(geoflx[40500:48600,0].reshape([90,90]),coords=[np.arange(0,90,1),
↪np.arange(0,90,1)],
                        dims=['j','i'])
geoflx06 = xr.DataArray(geoflx[48600:56700,0].reshape([90,90]),coords=[np.arange(0,90,1),
↪np.arange(0,90,1)],
                        dims=['j','i'])

geoflx0709 = geoflx[56700:81000,0].reshape([90,270])
geoflx07 = xr.DataArray(geoflx0709[:,:90],coords=[np.arange(0,90,1),np.arange(0,90,1)],
↪dims=['j','i'])
geoflx08 = xr.DataArray(geoflx0709[:,90:180],coords=[np.arange(0,90,1),np.arange(0,90,
```

(continues on next page)

(continued from previous page)

```
↪1)],dims=['j','i'])
geoflx09 = xr.DataArray(geoflx0709[:,180:],coords=[np.arange(0,90,1),np.arange(0,90,1)],
↪dims=['j','i'])

geoflx1012 = geoflx[81000:,0].reshape([90,270])
geoflx10 = xr.DataArray(geoflx1012[:,:90],coords=[np.arange(0,90,1),np.arange(0,90,1)],
↪dims=['j','i'])
geoflx11 = xr.DataArray(geoflx1012[:,90:180],coords=[np.arange(0,90,1),np.arange(0,90,
↪1)],dims=['j','i'])
geoflx12 = xr.DataArray(geoflx1012[:,180:],coords=[np.arange(0,90,1),np.arange(0,90,1)],
↪dims=['j','i'])
```

[8]:
```
geoflx_llc = xr.concat([geoflx00,geoflx01,geoflx02,geoflx03,geoflx04,geoflx05,geoflx06,
                        geoflx07,geoflx08,geoflx09,geoflx10,geoflx11,geoflx12], 'face')
```

Note: Geothermal flux needs to be a three dimensional field since the sources are distributed along the ocean floor at various depths. This requires a three dimensional mask (see below).

1.24.2 Define terms

Before doing the budget calculations we need to define some terms that will be used in the budget calculations

Number of seconds between each snapshot

Note: There are no snapshots for the first and last time point. Thus, we are skipping budget calculations for January and December 1992.

[9]:
```
dt = ds_snp.time[1:].load()
# delta t in seconds. Note: devide by 10**9 to convert nanoseconds to seconds
dt.values = [float(t)/10**9 for t in np.diff(ds_snp.time)]

# time axis of dt should be the same as of the monthly averages
dt.time.values = ds_ave.time[1:-1].values
```

Relevant constants

[10]:
```
# Density kg/m^3
rhoconst = 1029

# Heat capacity (J/kg/K)
c_p = 3994

# Constants for surface heat penetration (from Table 2 of Paulson and Simpson, 1977)
R = 0.62
zeta1 = 0.6
zeta2 = 20.0
```

Ocean depth

[11]:
```
# Ocean depth (m)
Depth = ds_snp.sel(face=1).Depth.load()
```

Grid dimensions

Note: Only use one face for testing

[12]:
```
dxG = ds_ave.sel(face=1).dxG.load()
dyG = ds_ave.sel(face=1).dyG.load()
```

[13]:
```
rA = ds_ave.sel(face=1).rA.load()
drF = ds_ave.sel(face=1).drF.load()
hFacC = ds_ave.sel(face=1).hFacC.load()
```

[14]:
```
# Volume (m^3)
vol = (rA*drF*hFacC).transpose('k','j','i')
```

Land mask

[15]:
```
# Make copy of hFacC
mskC = hFacC.copy(deep=True).load()

# Change all fractions (ocean) to 1. land = 0
mskC.values[mskC.values>0] = 1
```

[16]:
```
# Make 2D land mask for surface (This is just for plotting/mapping purposes)
land_mask = mskC[0]
land_mask.values[land_mask.values==0] = np.nan
```

Creating the grid object

[17]:
```
grid = xgcm.Grid(ds_ave.sel(face=1), periodic=False)
```

1.24.3 Evaluating the volume budget

$$G^{\eta,tot} = G^{\eta,conv} + G^{\eta,forc}$$

$$\frac{1}{H}\frac{\partial \eta}{\partial t} = -\nabla_{z^*}(s^* \, \mathbf{v}) - \frac{\partial w}{\partial z^*} + s^* \, F$$

Total tendency

- ETAN: Surface Height Anomaly (m)

[18]:
```
# Load snapshots for surface height anomaly from dataset (here: only one face is used)
ETANsnp = ds_snp.sel(face=1).ETAN.load()
```

[19]:
```
# Total tendency (1/month)
tendV_perMonth = (xr.DataArray(50*[1],coords={'k': np.array(range(0,50))},dims=['k'])*\
                  (1/Depth)*(ETANsnp.shift(time=-1)-ETANsnp)).transpose('time','k','j','i
↪')[:-1]
```

```
/rigel/home/jt2796/miniconda/envs/default/lib/python2.7/site-packages/xarray/core/
↪variable.py:1165: RuntimeWarning: divide by zero encountered in divide
  else f(other_data, self_data))
/rigel/home/jt2796/miniconda/envs/default/lib/python2.7/site-packages/xarray/core/
↪variable.py:1164: RuntimeWarning: invalid value encountered in multiply
  if not reflexive
```

[20]:
```
# Make sure time axis is the same as for the monthly variables
tendV_perMonth.time.values = ds_ave.time[1:-1].values
```

[21]:
```
# Convert tendency from 1/month to 1/s
tendV_perSec = tendV_perMonth/dt
```

[22]:
```
# Predefine tendV array with correct dimensions
tendV = xr.DataArray(np.nan*np.zeros([np.shape(tendV_perSec)[0]+2,50,90,90]),
                     coords={'time': range(np.shape(tendV_perSec)[0]+2),'k': np.
↪array(range(0,50)),
                             'j': np.array(range(0,90)),'i': np.array(range(0,90))},
↪dims=['time','k','j','i'])

# Time
tendV.time.values = ds_ave.time.values

# Add coordinates
tendV['XC'] = ds_snp.XC.sel(face=1)
tendV['YC'] = ds_snp.YC.sel(face=1)
tendV['Z'] = ds_snp.Z
```

[23]:
```
# Total tendency (1/s)
tendV.values[1:-1] = tendV_perSec.values
```

Forcing

- oceFWflx: net surface Fresh-Water flux into the ocean (kg/m^2/s)

[24]:
```
# Load monthly averaged freshwater flux (here: only one face is used)
oceFWflx = ds_ave.sel(face=1).oceFWflx.load()
```

[25]:
```
# Sea surface forcing on volume (1/s)
forcV = ((oceFWflx/rhoconst)/(hFacC*drF)).transpose('time','k','j','i')

# Make sure forcing term is zero below the surface
forcV.values[:,1:] = 0*forcV.values[:,1:]
```

```
/rigel/home/jt2796/miniconda/envs/default/lib/python2.7/site-packages/xarray/core/
↪variable.py:1164: RuntimeWarning: divide by zero encountered in divide
  if not reflexive
/rigel/home/jt2796/miniconda/envs/default/lib/python2.7/site-packages/xarray/core/
↪variable.py:1164: RuntimeWarning: invalid value encountered in divide
  if not reflexive
/rigel/home/jt2796/miniconda/envs/default/lib/python2.7/site-packages/ipykernel_launcher.
↪py:5: RuntimeWarning: invalid value encountered in multiply
  """
```

Horizontal convergence

- UVELMASS: U Mass-Weighted Comp of Velocity (m/s)
- VVELMASS: V Mass-Weighted Comp of Velocity (m/s)

[ ]:
```
```

[26]:
```
# Load monthly averaged velocities (here: only one face is used)
UVELMASS = ds_ave.sel(face=1).UVELMASS.load()
VVELMASS = ds_ave.sel(face=1).VVELMASS.load()
```

Note: Volume transports are calculated the same way as in the xgcm example (http://xgcm.readthedocs.io/en/latest/example_mitgcm.html#Divergence-Calculation). The only difference here is the omission of `hFacW` and `hFacS`. Including `hFacW` and `hFacS` in the calculation of the transport terms introduces unrealistic artifacts in the horizontal convergence near the ocean floor, which in turn causes the volume budget to be not balanced in the deeper ocean layers.

[27]:
```
# Horizontal volume transports (m^3/s)
u_transport = UVELMASS * dyG * drF
v_transport = VVELMASS * dxG * drF
```

[28]:
```
# Convergence of the horizontal flow (1/s)
hConvV = -(grid.diff(u_transport, 'X', boundary='extend') + \
           grid.diff(v_transport, 'Y', boundary='extend'))/vol
```

Vertical convergence

- WVELMASS: Vertical Mass-Weighted Comp of Velocity (m/s)

[29]:
```
# Load monthly averaged vertical velocity (here: only one face is used)
WVELMASS = ds_ave.sel(face=1).WVELMASS.load()
```

[30]:
```
# Vertical volume transport (m^3/s)
w_transport = WVELMASS * rA
```

Note: Apparently, it is required to add the vertical volume flux at the air-sea interface (`oceFWflx`) to the surface layer to balance the budget.

[31]:
```
# Add the vertical volume flux at the air-sea interface
w_transport[:,0] = w_transport[:,0]+(forcV*vol)[:,0]
```

[32]:
```
# Convergence of the vertical flow (m^3/s)
vConvV = grid.diff(w_transport, 'Z', boundary='extend')
```

Note: Convergence in the deepest depth layer in `vConvV` needs to be replaced by minus the vertical volume flux. Otherwise, the volume budget in the deepest layer will be unbalanced. This is probably an issue with the given way `grid.diff()` calculates values at the edges.

[33]:
```
vConvV[:,-1,:,:] = -w_transport[:,-1,:,:]
```

[34]:
```
# Convergence of the vertical flow (1/s)
vConvV = vConvV/vol
```

Total convergence

[35]:
```
ConvV = hConvV+vConvV
```

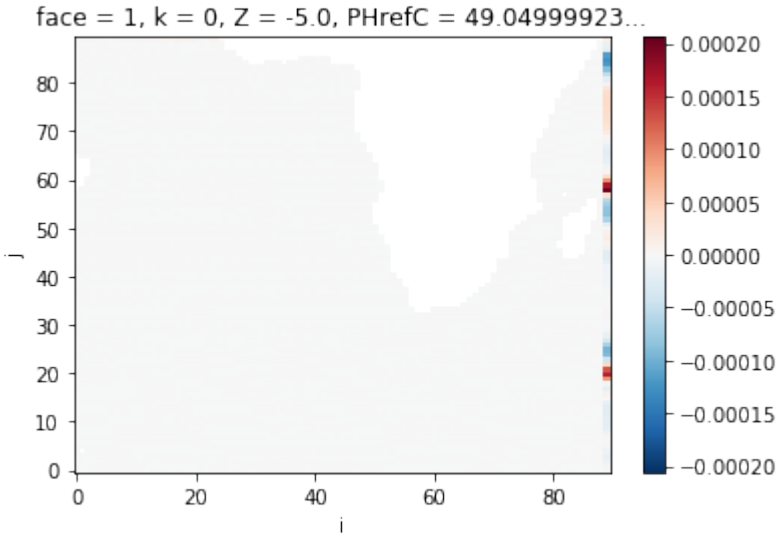
Total tendency

[36]:
```
totalV = ConvV + forcV
```

Map accumulated residual in volume budget

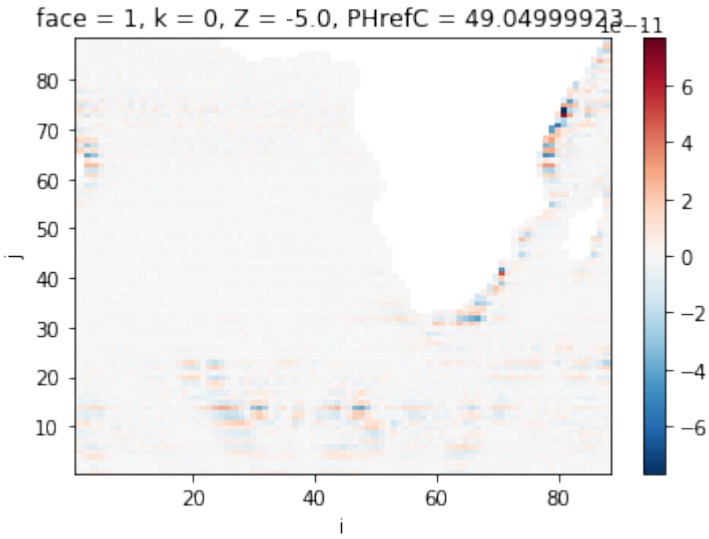
[37]:
```
((totalV-tendV).sum(dim='k').sum(dim='time')*land_mask).plot(cmap='RdBu_r')
```

[37]:
```
<matplotlib.collections.QuadMesh at 0x2aab17f0ca10>
```



[38]:
```
# Ignoring face boundaries
((totalV-tendV).sum(dim='k').sum(dim='time')*land_mask)[1:89,1:89].plot(cmap='RdBu_r')
```

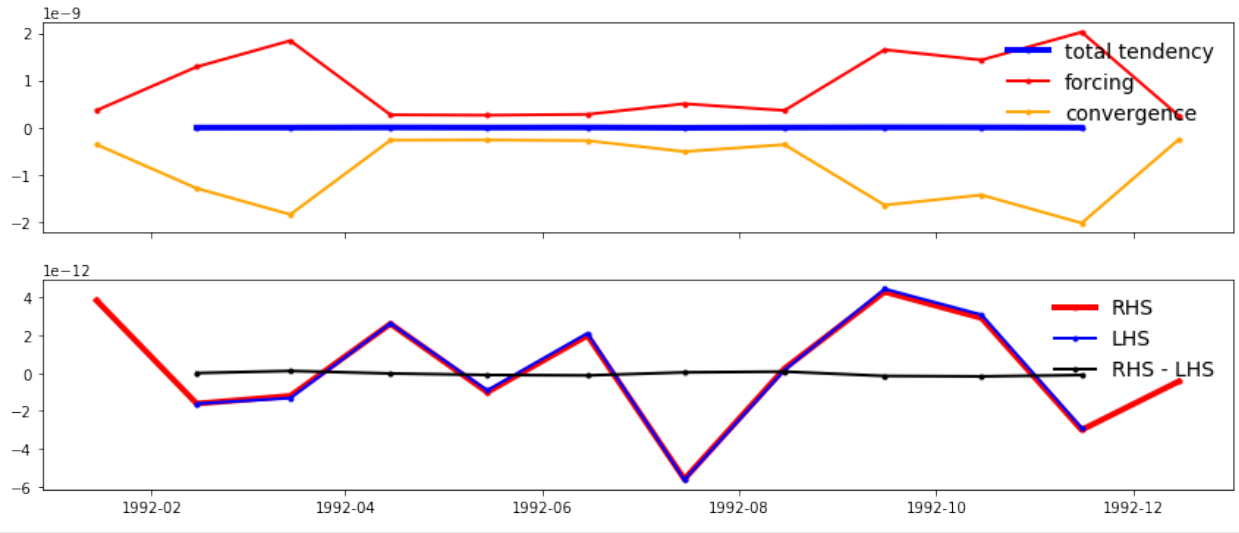
[38]:
```
<matplotlib.collections.QuadMesh at 0x2aab6ff611d0>
```



Note: The residuals are larger when doing the calculations on Unix (on Habanero) compared to Mac OS. When running the code on Unix, there is some bias in the horizontal convergence term (`hConvV`), but it is very small.

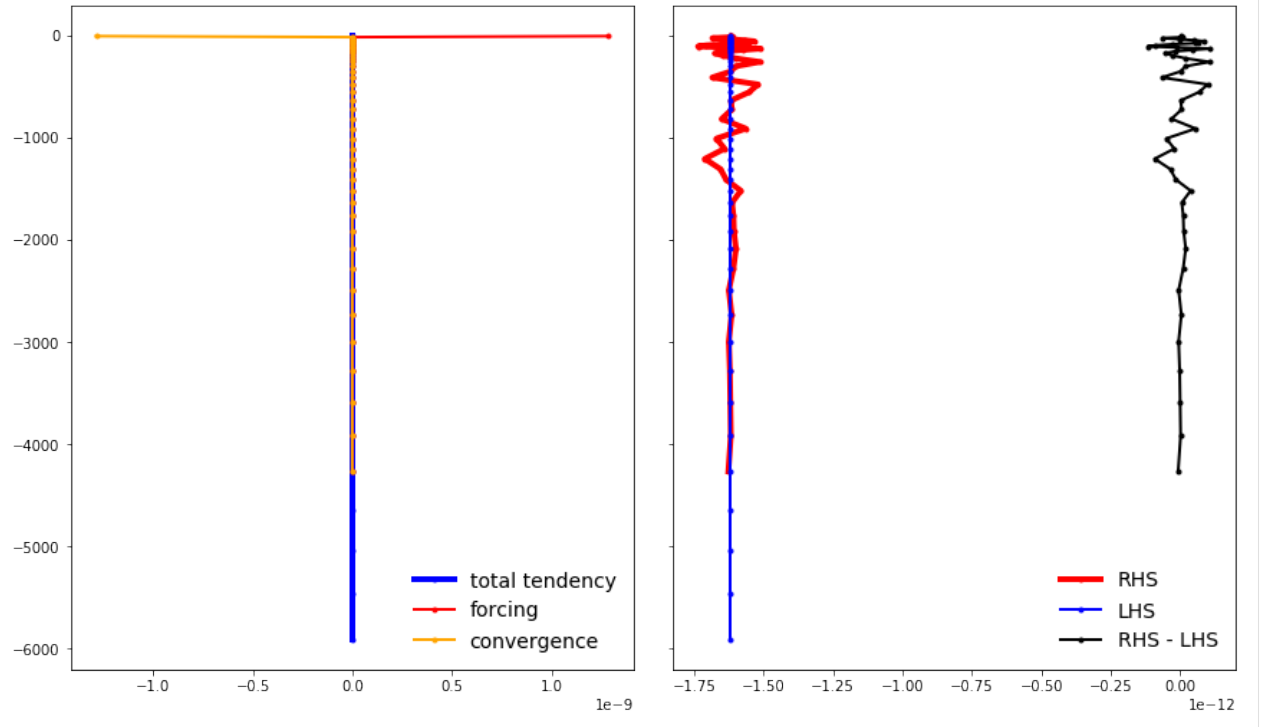
Time series for an arbitrarily chosen grid point

[39]:
```
k = 0
j = 10
i = 10

f, axes = plt.subplots(2, 1,figsize=(12,5))
f.tight_layout()

plt.subplot(2, 1, 1)
plt.plot(tendV.time, tendV[:,k,j,i], lw=4, color='blue', marker='.',label='total tendency
↪')
plt.plot(forcV.time, forcV[:,k,j,i], lw=2, color='red', marker='.',label='forcing')
plt.plot(ConvV.time, ConvV[:,k,j,i], lw=2, color='orange', marker='.',label='convergence
↪')
plt.setp(plt.gca(), 'xticklabels',[])
plt.legend(loc='upper right',frameon=False,fontsize=14)

plt.subplot(2, 1, 2)
plt.plot(totalV.time, totalV[:,k,j,i], lw=4, color='red', marker='.',label='RHS')
plt.plot(tendV.time, tendV[:,k,j,i], lw=2, color='blue', marker='.',label='LHS')
plt.plot(tendV.time, totalV[:,k,j,i]-tendV[:,k,j,i], lw=2, color='k', marker='.',label=
↪'RHS - LHS')
plt.legend(loc='upper right',frameon=False,fontsize=14)

plt.show()
```



Verical profiles for an arbitrarily chosen grid point

[40]:
```
t = 1
j = 10
i = 10

f, axes = plt.subplots(1, 2, sharey=True, figsize=(12,7))
f.tight_layout()

plt.subplot(1, 2, 1)
plt.plot(tendV[t,:,j,i],tendV.Z, lw=4, color='blue', marker='.',label='total tendency')
plt.plot(forcV[t,:,j,i],forcV.Z, lw=2, color='red', marker='.',label='forcing')
plt.plot(ConvV[t,:,j,i],ConvV.Z, lw=2, color='orange', marker='.',label='convergence')
plt.legend(loc='lower right',frameon=False,fontsize=14)

plt.subplot(1, 2, 2)
plt.plot(totalV[t,:,j,i],totalV.Z, lw=4, color='red', marker='.',label='RHS')
plt.plot(tendV[t,:,j,i],tendV.Z, lw=2, color='blue', marker='.',label='LHS')
plt.plot(totalV[t,:,j,i]-tendV[t,:,j,i],totalV.Z, lw=2, color='k', marker='.',label='RHS␣
↪- LHS')
plt.setp(plt.gca(), 'yticklabels',[])
plt.legend(loc='lower right',frameon=False,fontsize=14)

plt.show()
```



1.24.4 Evaluating the heat budget

$$G^{\theta,tot} = G^{\theta,adv} + G^{\theta,forc} + G^{\theta,diff}$$

$$\frac{\partial(s^*\theta)}{\partial t} = -\nabla_{z^*}(s^*\,\theta\,\mathbf{v_{res}}) - \frac{\partial(\theta\,w_{res})}{\partial z^*} + s^*\,F_\theta + s^*\,D_\theta$$

Total tendency

- THETA: Potential Temperature (degC)

[41]:
```
# Load snapshots of theta (here only one face is used)
THETAsnp = ds_snp.sel(face=1).THETA.load()
```

[42]:
```
# Calculate the stheta term
HCsnp = THETAsnp*(1+ETANsnp/Depth)
```

[43]:
```
# Total tendency (degC/month)
tendH_perMonth = (HCsnp.shift(time=-1)-HCsnp)[:-1]
```

[44]:
```
# Make sure time axis is the same as for the monthly variables
tendH_perMonth.time.values = ds_ave.time[1:-1].values
```

[45]:
```
# Convert tendency from 1/month to 1/s
tendH_perSec = tendH_perMonth/dt
```

[46]:
```
# Predefine tendH array with correct dimensions
tendH = xr.DataArray(np.nan*np.zeros([np.shape(tendH_perSec)[0]+2,50,90,90]),
                     coords={'time': range(np.shape(tendH_perSec)[0]+2),'k': np.
↪array(range(0,50)),
                             'j': np.array(range(0,90)),'i': np.array(range(0,90))},
↪dims=['time','k','j','i'])

# Time
tendH.time.values = ds_ave.time.values

# Add coordinates
tendH['XC'] = ds_snp.XC.sel(face=1)
tendH['YC'] = ds_snp.YC.sel(face=1)
tendH['Z'] = ds_snp.Z
```

[47]:
```
# Total tendency (degC/s)
tendH.values[1:-1] = tendH_perSec.values
```

Forcing

- TFLUX: total heat flux (match heat-content variations) (W/m^2)
- oceQsw: net Short-Wave radiation (+=down) (W/m^2)

[48]:
```
# Load monthly averages of heat flux and shortwave radiation (here only one face is used)
TFLUX = ds_ave.sel(face=1).TFLUX.load()
oceQsw = ds_ave.sel(face=1).oceQsw.load()
```

Defining terms needed for evaluating surface heat forcing

[49]:
```
Z = ds_ave.sel(face=1).Z.load()
RF = np.concatenate([ds_ave.sel(face=1).Zp1.values[:-1],[np.nan]])
```

Note: `Z` and `Zp1` are used in deriving surface heat penetration. MATLAB code uses `RF` from `mygrid` structure.

[50]:
```
q1 = R*np.exp(1.0/zeta1*RF[:-1]) + (1.0-R)*np.exp(1.0/zeta2*RF[:-1])
q2 = R*np.exp(1.0/zeta1*RF[1:]) + (1.0-R)*np.exp(1.0/zeta2*RF[1:])
```

[51]:
```
# Correction for the 200m cutoff
zCut = np.where(Z < -200)[0][0]
q1[zCut:] = 0
q2[zCut-1:] = 0
```

[52]:
```
# Save q1 and q2 as xarray data arrays
q1 = xr.DataArray(q1,coords=[Z.k],dims=['k'])
q2 = xr.DataArray(q2,coords=[Z.k],dims=['k'])
```

Compute vertically penetrating flux

[53]:
```
# Surface heat flux (below the surface)
forcH = ((q1*(mskC==1)-q2*(mskC.shift(k=-1)==1))*oceQsw).transpose('time','k','j','i')

# Reset surface layer to zero
forcH.values[:,0] = 0*forcH.values[:,0]
```

[54]:
```
# Surface heat flux (at the sea surface)
forcH[:,0] = ((TFLUX - (1-(q1[0]-q2[0]))*oceQsw)*mskC[0]).transpose('time','j','i')
```

Add geothermal heat flux

[55]:
```
# Create 3d bathymetry mask
mskC_shifted = mskC.shift(k=-1)
mskC_shifted.values[-1,:,:] = 0
mskb = mskC - mskC_shifted
```

[56]:
```
# Create 3d field of geothermal heat flux
geoflx2d = geoflx_llc.sel(face=1)
geoflx3d = geoflx2d * mskb
GEOFLX = geoflx3d.transpose('k','j','i')
```

[57]:
```
# Add geothermal heat flux to forcing field and convert from W/m^2 to degC/s
forcH = ((forcH + GEOFLX)/(rhoconst*c_p))/(hFacC*drF)
```

Advection

Horizontal convergence

- ADVx_TH: U Comp. Advective Flux of Pot.Temperature (degC m^3/s)
- ADVy_TH: V Comp. Advective Flux of Pot.Temperature (degC m^3/s)

[58]:
```
# Load monthly averaged advective fluxes (here only one face is used)
ADVx_TH = ds_ave.sel(face=1).ADVx_TH.load()
ADVy_TH = ds_ave.sel(face=1).ADVy_TH.load()
```

[59]:
```
# Convergence of horizontal advection (degC/s)
adv_hConvH = -(grid.diff(ADVx_TH, 'X', boundary='extend') + \
               grid.diff(ADVy_TH, 'Y', boundary='extend'))/vol
```

Vertical convergence

- ADVr_TH: Vertical Advective Flux of Pot.Temperature (degC m^3/s)

[60]:
```
# Load monthly averages of vertical advective flux (here only one face is used)
ADVr_TH = ds_ave.sel(face=1).ADVr_TH.load()
```

Note: The heat budget only balances when the sea surface forcing is not added to the vertical advective flux (at the air-sea interface). This is different from the volume and salinity budget.

[61]:
```
# Convergence of the vertical advection (degC m^3/s)
adv_vConvH = grid.diff(ADVr_TH, 'Z', boundary='extend')
```

Note: Convergence in the deepest layer in `adv_vConvH` needs to be replaced by minus the vertical advective flux. Otherwise, the volume budget in the deepest layer will be unbalanced. This is probably an issue with the given way `grid.diff()` calculates values at the edges.

[62]:
```
adv_vConvH[:,-1,:,:] = -ADVr_TH[:,-1,:,:]
```

[63]:
```
# Convergence of vertical advection (degC/s)
adv_vConvH = adv_vConvH/vol
```

Diffusion

Horizontal convergence

- DFxE_TH: U Comp. Diffusive Flux of Pot.Temperature (degC m^3/s)
- DFyE_TH: V Comp. Diffusive Flux of Pot.Temperature (degC m^3/s)

[64]:
```
# Load monthly averages of diffusive fluxes (here only one face is used)
DFxE_TH = ds_ave.sel(face=1).DFxE_TH.load()
DFyE_TH = ds_ave.sel(face=1).DFyE_TH.load()
```

[65]:
```
# Convergence of horizontal diffusion (degC/s)
dif_hConvH = -(grid.diff(DFxE_TH, 'X', boundary='extend') + \
               grid.diff(DFyE_TH, 'Y', boundary='extend'))/vol
```

Vertical convergence

- DFrE_TH: Vertical Diffusive Flux of Pot.Temperature (Explicit part) (degC m^3/s)
- DFrI_TH: Vertical Diffusive Flux of Pot.Temperature (Implicit part) (degC m^3/s)

[66]:
```
# Load monthly averages of vertical diffusive fluxes (here only one face is used)
DFrE_TH = ds_ave.sel(face=1).DFrE_TH.load()
DFrI_TH = ds_ave.sel(face=1).DFrI_TH.load()
```

[67]:
```
# Convergence of vertical diffusion (degC m^3/s)
dif_vConvH = grid.diff(DFrE_TH, 'Z', boundary='extend') + grid.diff(DFrI_TH, 'Z',␣
↪boundary='extend')
```

Note: Convergence in the deepest layer in `dif_vConvH` needs to be replaced by minus the vertical diffusive flux to balance the budget.

[68]:
```
dif_vConvH[:,-1,:,:] = -(DFrE_TH+DFrI_TH)[:,-1,:,:]
```

[69]:
```
# Convergence of vertical diffusion (degC/s)
dif_vConvH = dif_vConvH/vol
```

Total convergences

[70]:
```
# Total convergence of advective flux
adv_ConvH = adv_hConvH + adv_vConvH

# Total convergence of diffusive flux
dif_ConvH = dif_hConvH + dif_vConvH

# Total convergence
ConvH = adv_ConvH + dif_ConvH
```

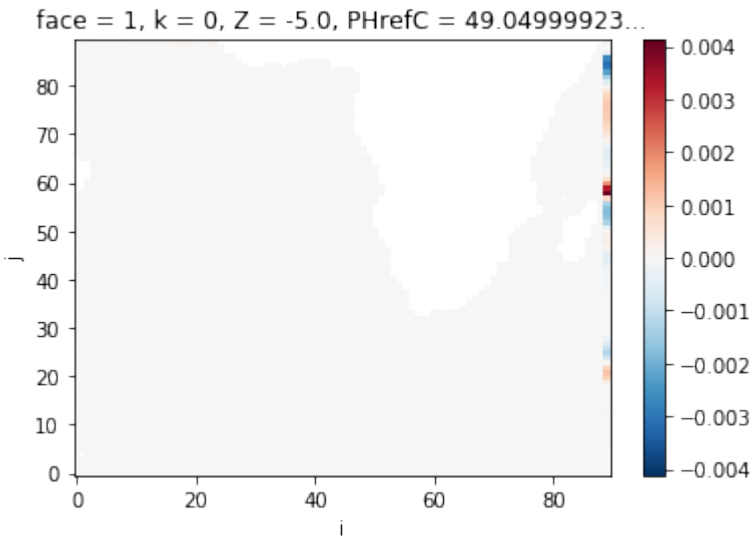
Total tendency

[71]:
```
totalH = ConvH + forcH
```

Map accumulated residual in volume budget

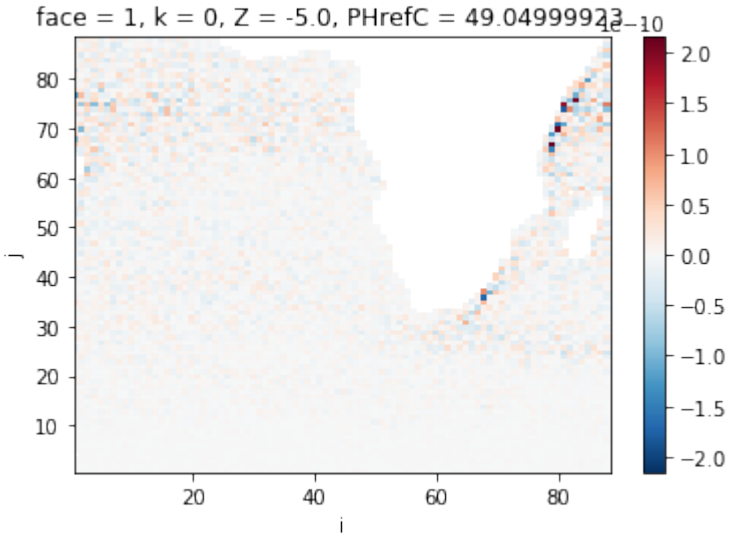
[72]:
```
((totalH-tendH).sum(dim='k').sum(dim='time')*land_mask).plot(cmap='RdBu_r')
```

[72]:
```
<matplotlib.collections.QuadMesh at 0x2aac43f29350>
```



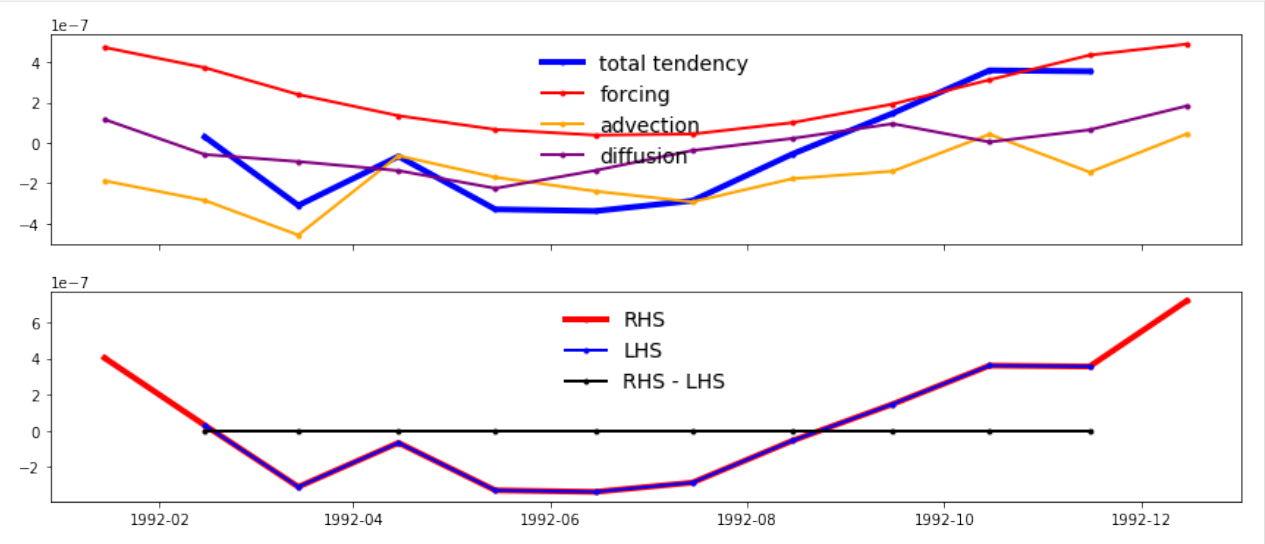
[73]:
```
# Ignoring face boundaries
((totalH-tendH).sum(dim='k').sum(dim='time')*land_mask)[1:89,1:89].plot(cmap='RdBu_r')
```

[73]:
```
<matplotlib.collections.QuadMesh at 0x2aab1804b8d0>
```



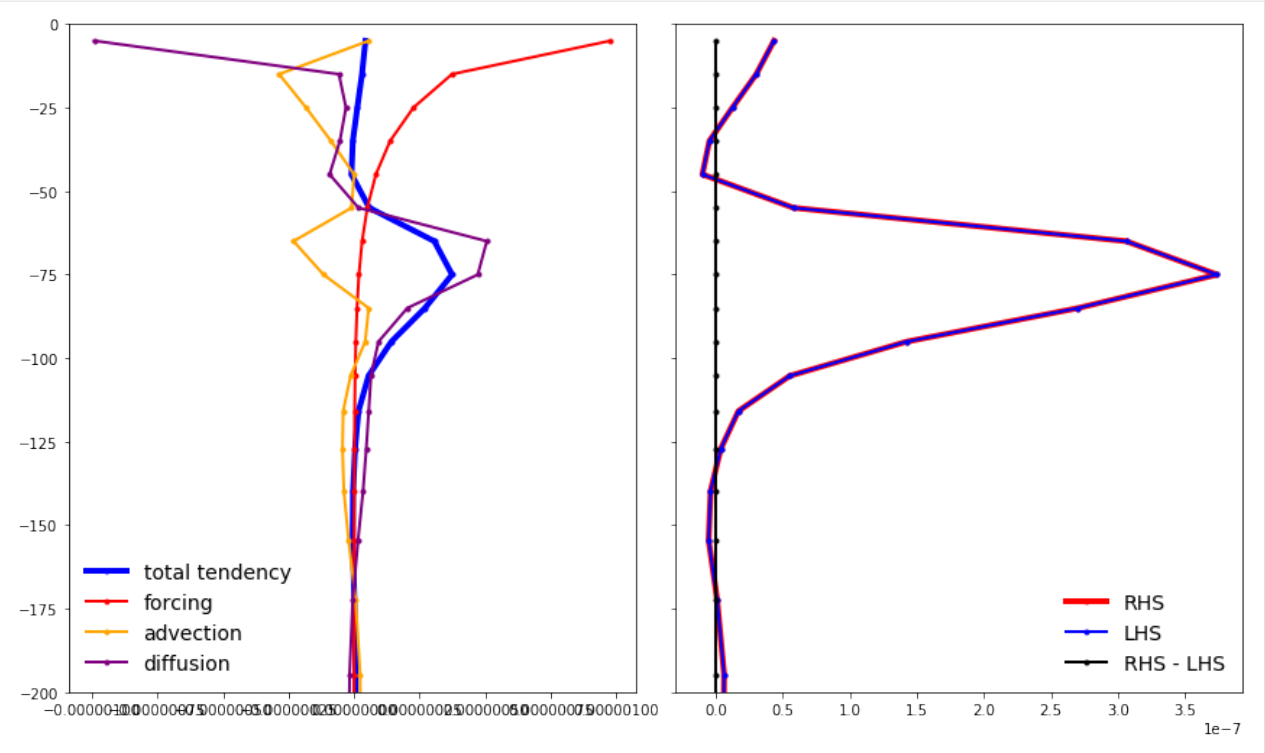
Time series for an arbitrarily chosen grid point

[74]:
```
k = 1
j = 10
i = 10

f, axes = plt.subplots(2, 1,figsize=(12,5))
f.tight_layout()

plt.subplot(2, 1, 1)
plt.plot(tendH.time, tendH[:,k,j,i], lw=4, color='blue', marker='.',label='total tendency
↪')
plt.plot(forcH.time, forcH[:,k,j,i], lw=2, color='red', marker='.',label='forcing')
plt.plot(adv_ConvH.time, adv_ConvH[:,k,j,i], lw=2, color='orange', marker='.',label=
↪'advection')
plt.plot(dif_ConvH.time, dif_ConvH[:,k,j,i], lw=2, color='purple', marker='.',label=
↪'diffusion')
plt.setp(plt.gca(), 'xticklabels',[])
plt.legend(loc='upper center',frameon=False,fontsize=14)

plt.subplot(2, 1, 2)
plt.plot(totalH.time, totalH[:,k,j,i], lw=4, color='red', marker='.',label='RHS')
plt.plot(tendH.time, tendH[:,k,j,i], lw=2, color='blue', marker='.',label='LHS')
plt.plot(tendH.time, totalH[:,k,j,i]-tendH[:,k,j,i], lw=2, color='k', marker='.',label=
↪'RHS - LHS')
plt.legend(loc='upper center',frameon=False,fontsize=14)

plt.show()
```



Verical profiles for an arbitrarily chosen grid point

[75]:
```
t = 1
j = 10
i = 10

f, axes = plt.subplots(1, 2, sharey=True, figsize=(12,7))
f.tight_layout()

plt.subplot(1, 2, 1)
plt.plot(tendH[t,:,j,i],tendH.Z, lw=4, color='blue', marker='.',label='total tendency')
plt.plot(forcH[t,:,j,i],forcH.Z, lw=2, color='red', marker='.',label='forcing')
plt.plot(adv_ConvH[t,:,j,i],adv_ConvH.Z, lw=2, color='orange', marker='.',label=
↪'advection')
plt.plot(dif_ConvH[t,:,j,i],dif_ConvH.Z, lw=2, color='purple', marker='.',label=
↪'diffusion')
plt.legend(loc='lower left',frameon=False,fontsize=14)
plt.ylim([-200,0])

plt.subplot(1, 2, 2)
plt.plot(totalH[t,:,j,i],totalH.Z, lw=4, color='red', marker='.',label='RHS')
plt.plot(tendH[t,:,j,i],tendH.Z, lw=2, color='blue', marker='.',label='LHS')
plt.plot(totalH[t,:,j,i]-tendH[t,:,j,i],tendH.Z, lw=2, color='k', marker='.',label='RHS -
↪ LHS')
plt.setp(plt.gca(), 'yticklabels',[])
plt.legend(loc='lower right',frameon=False,fontsize=14)
plt.ylim([-200,0])

plt.show()
```



1.24.5 Evaluating the salt budget

$$G^{S,tot} = G^{S,adv} + G^{S,forc} + G^{S,diff}$$

$$\frac{\partial (s^* S)}{\partial t} = -\nabla_{z^*}(s^* \, S \, \mathbf{v_{res}}) - \frac{\partial (S \, w_{res})}{\partial z^*} + s^* \, F_S + s^* \, D_S$$

Total tendency

- SALT: Salinity (psu)

[76]:
```
# Load salinity snapshot (here only one face is used)
SALTsnp = ds_snp.sel(face=1).SALT.load()
```

[77]:
```
# Calculate s*S term
sSALT = SALTsnp*(1+ETANsnp/Depth)
```

[78]:
```
# Total tendency (psu/month)
tendS_perMonth = (sSALT.shift(time=-1)-sSALT)[:-1]x
```

[79]:
```
# Make sure time axis is the same as for the monthly variables
tendS_perMonth.time.values = ds_ave.time[1:-1].values
```

[80]:
```
# Convert tendency from 1/month to 1/s
tendS_perSec = tendS_perMonth/dt
```

[81]:
```
# Predefine tendS array with correct dimensions
tendS = xr.DataArray(np.nan*np.zeros([np.shape(tendS_perSec)[0]+2,50,90,90]),
                     coords={'time': range(np.shape(tendS_perSec)[0]+2),'k': np.
↪array(range(0,50)),
                             'j': np.array(range(0,90)),'i': np.array(range(0,90))},
↪dims=['time','k','j','i'])

# Time
tendS.time.values = ds_ave.time.values

# Add coordinates
tendS['XC'] = ds_snp.XC.sel(face=1)
tendS['YC'] = ds_snp.YC.sel(face=1)
tendS['Z'] = ds_snp.Z
```

[82]:
```
# Total tendency (psu/s)
tendS.values[1:-1] = tendS_perSec.values
```

Forcing

- SFLUX: total salt flux (match salt-content variations) (g/m^2/s)
- oceSPtnd: salt tendency due to salt plume flux (g/m^2/s)

[83]:
```
# Load monthly averaged SFLUX and oceSPtnd (here only one face is used)
SFLUX = ds_ave.sel(face=1).SFLUX.load()
oceSPtnd = ds_ave.sel(face=1).oceSPtnd.load()
```

[84]:
```
# Expand SFLUX along depth axis
SFLUX3d = xr.concat(50*[SFLUX.expand_dims('k',1)],dim='k')
SFLUX3d.coords['k'] = SFLUX3d.k

# Reset SFLUX3d to zero below surface layer
SFLUX3d.values[:,1:] = 0*SFLUX3d[:,1:].values
```

Note: `SFLUX` and `oceSPtnd` is given in g/m^2/s. Dividing by density and corresponding vertical length scale (`drF`) results in g/kg/s, which is the same as psu/s.

[85]:
```
# Surface salt flux (psu/s)
forcS = (((SFLUX3d+oceSPtnd)/rhoconst)/(hFacC*drF)).transpose('time','k','j','i')
```

Advection

Horizontal convergence

- ADVx_SLT: U Comp. Advective Flux of Salinity (psu m^3/s)
- ADVy_SLT: V Comp. Advective Flux of Salinity (psu m^3/s)

[86]:
```
# Load monthly averaged advective fluxes (here only one face is used)
ADVx_SLT = ds_ave.sel(face=1).ADVx_SLT.load()
ADVy_SLT = ds_ave.sel(face=1).ADVy_SLT.load()
```

[87]:
```
# Convergence of horizontal advection (psu/s)
adv_hConvS = -(grid.diff(ADVx_SLT, 'X', boundary='extend') + \
               grid.diff(ADVy_SLT, 'Y', boundary='extend'))/vol
```

Vertical convergence

- ADVr_SLT: Vertical Advective Flux of Salinity (psu m^3/s)

[88]:
```
# Load monthly averaged vertical advective flux (here only one face is used)
ADVr_SLT = ds_ave.sel(face=1).ADVr_SLT.load()
```

Note: The salt budget only balances when the sea surface forcing is not added to the vertical salt flux (at the air-sea interface). This is different from the volume and salinity budget.

[89]:
```
# Convergence of vertical advection (psu m^3/s)
adv_vConvS = grid.diff(ADVr_SLT, 'Z', boundary='extend')
```

Note: The deepest depth layer in `adv_vConvS` needs to be replaced by minus the vertical advective flux. This is probably an issue with how `grid.diff()` calculates the edges.

[90]:
```
adv_vConvS[:,-1,:,:] = -ADVr_SLT[:,-1,:,:]
```

[91]:
```
# Convergence of vertical advection (psu/s)
adv_vConvS = adv_vConvS/vol
```

Diffusion

Horizontal convergence

- DFxE_SLT: U Comp. Diffusive Flux of Salinity (psu m^3/s)
- DFyE_SLT: V Comp. Diffusive Flux of Salinity (psu m^3/s)

[92]:
```
# Load monthly averaged horizontal diffusive fluxes (here only one face is used)
DFxE_SLT = ds_ave.sel(face=1).DFxE_SLT.load()
DFyE_SLT = ds_ave.sel(face=1).DFyE_SLT.load()
```

[93]:
```
# Convergence of horizontal diffusion (psu/s)
dif_hConvS = -(grid.diff(DFxE_SLT, 'X', boundary='extend') + \
               grid.diff(DFyE_SLT, 'Y', boundary='extend'))/vol
```

Vertical convergence

- DFrE_SLT: Vertical Diffusive Flux of Salinity (Explicit part) (psu m^3/s)
- DFrI_SLT: Vertical Diffusive Flux of Salinity (Implicit part) (psu m^3/s)

[94]:
```
# Load monthly averaged vertical diffusive fluxes (here only one face is used)
DFrE_SLT = ds_ave.sel(face=1).DFrE_SLT.load()
DFrI_SLT = ds_ave.sel(face=1).DFrI_SLT.load()
```

[95]:
```
# Convergence of vertical diffusion (psu m^3/s)
dif_vConvS = grid.diff(DFrE_SLT, 'Z', boundary='extend') + grid.diff(DFrI_SLT, 'Z',␣
↪boundary='extend')
```

Note: The deepest depth layer in `dif_vConvS` needs to be calculated seperately. This is probably an issue with the given boundary condition in `grid.diff()`.

[96]:
```
# Convergence of vertical diffusion (psu/s)
dif_vConvS = dif_vConvS/vol
```

Total convergences

[97]:
```
# Total convergence of advective flux
adv_ConvS = adv_hConvS + adv_vConvS

# Total convergence of diffusive flux
dif_ConvS = dif_hConvS + dif_vConvS

# Total convergence
ConvS = adv_ConvS + dif_ConvS
```

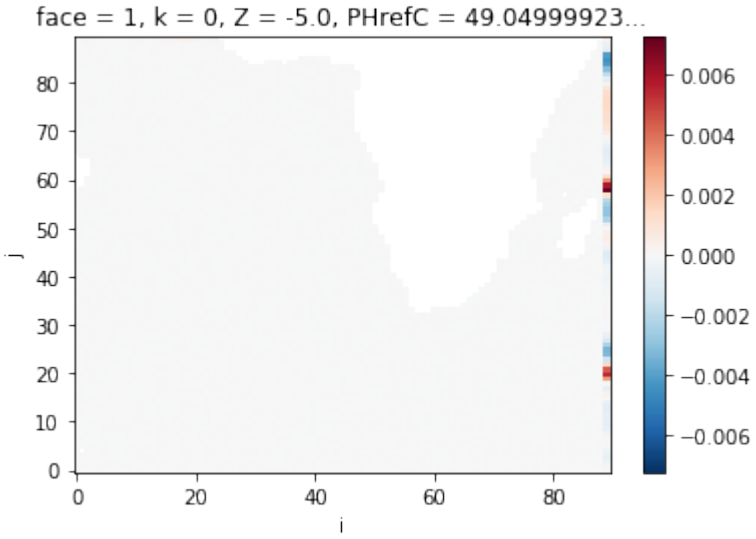
Total tendency

[98]:
```
totalS = ConvS + forcS
```

Plot accumulated residual in salt budget

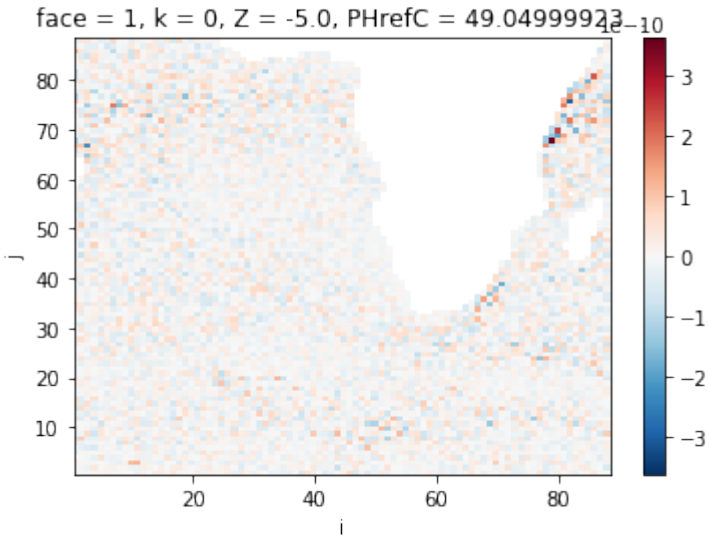
[99]:
```
((totalS-tendS).sum(dim='k').sum(dim='time')*land_mask).plot(cmap='RdBu_r')#,vmin=-5e-10,
↪vmax=5e-10)
```

[99]:
```
<matplotlib.collections.QuadMesh at 0x2aae5e3aaed0>
```



[100]:
```
# Ignoring face boundaries
((totalS-tendS).sum(dim='k').sum(dim='time')*land_mask)[1:89,1:89].plot(cmap='RdBu_r')
```

[100]:
```
<matplotlib.collections.QuadMesh at 0x2aabc3ef0090>
```



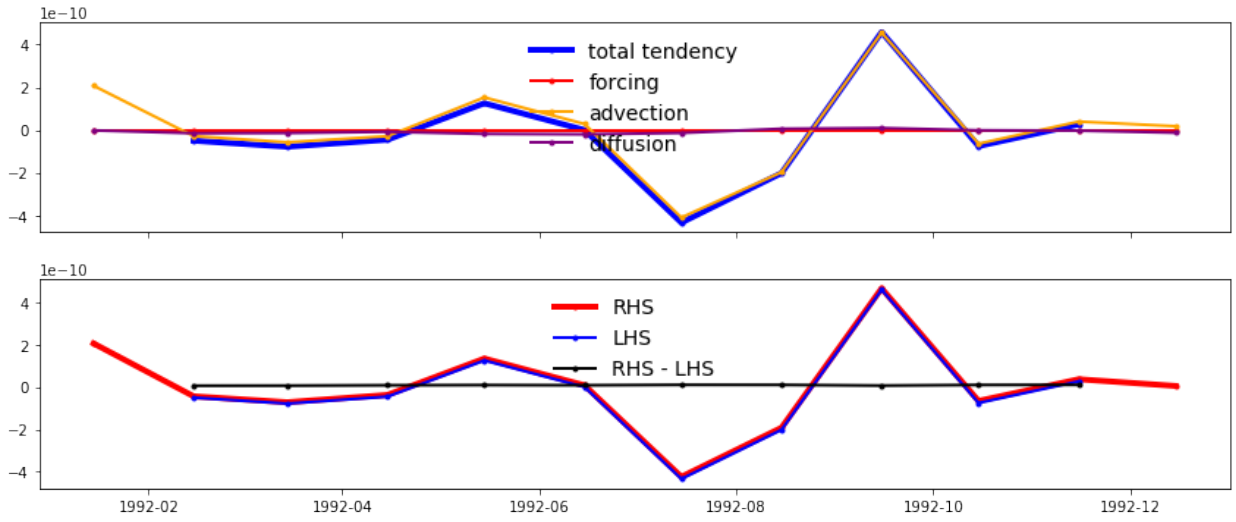
Time series for an arbitrarily chosen grid point

[101]:
```
k = -1
j = 3
i = 11

f, axes = plt.subplots(2, 1,figsize=(12,5))
f.tight_layout()

plt.subplot(2, 1, 1)
```

(continues on next page)

(continued from previous page)

```
plt.plot(tendS.time, tendS[:,k,j,i], lw=4, color='blue', marker='.',label='total tendency
↪')
plt.plot(forcS.time, forcS[:,k,j,i], lw=2, color='red', marker='.',label='forcing')
plt.plot(adv_ConvS.time, adv_ConvS[:,k,j,i], lw=2, color='orange', marker='.',label=
↪'advection')
plt.plot(dif_ConvS.time, dif_ConvS[:,k,j,i], lw=2, color='purple', marker='.',label=
↪'diffusion')
plt.setp(plt.gca(), 'xticklabels',[])
plt.legend(loc='upper center',frameon=False,fontsize=14)

plt.subplot(2, 1, 2)
#plt.axhline(y=0, xmin=0, xmax=1, linewidth=0.5, color = 'k')
plt.plot(totalS.time, totalS[:,k,j,i], lw=4, color='red', marker='.',label='RHS')
plt.plot(tendS.time, tendS[:,k,j,i], lw=2, color='blue', marker='.',label='LHS')
plt.plot(tendS.time, totalS[:,k,j,i]-tendS[:,k,j,i], lw=2, color='k', marker='.',label=
↪'RHS - LHS')
plt.legend(loc='upper center',frameon=False,fontsize=14)

plt.show()
```



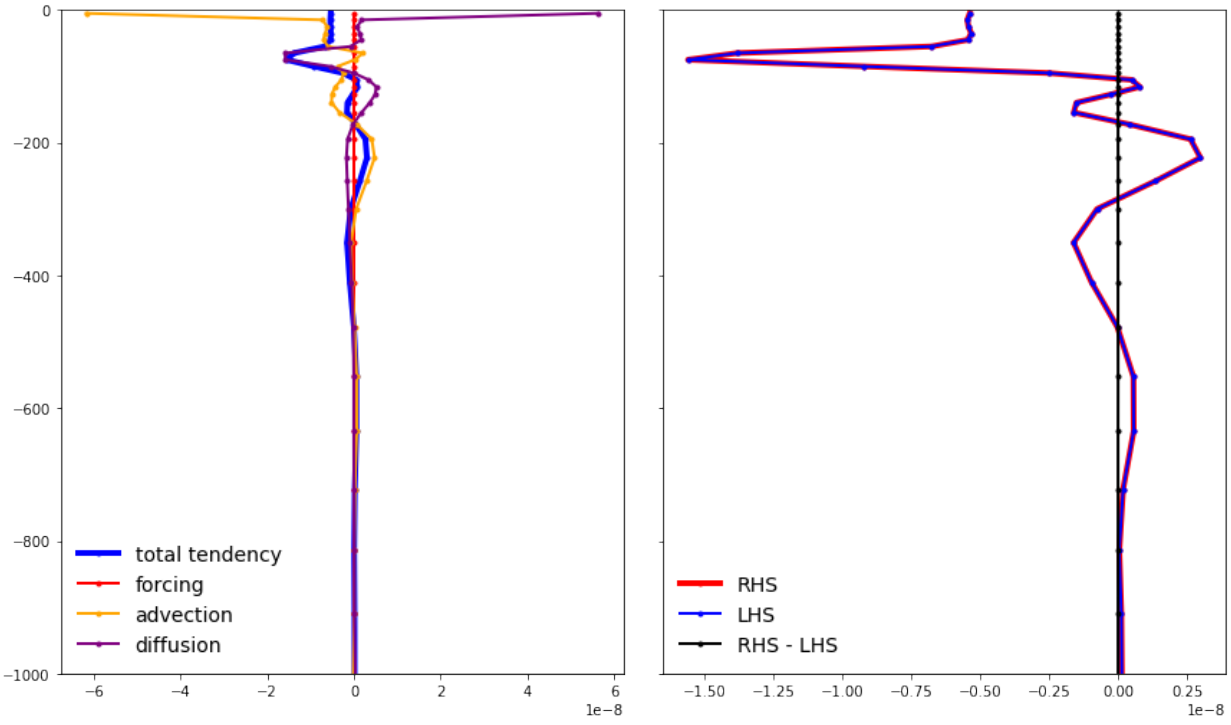
Verical profiles for an arbitrarily chosen grid point

[102]:
```
t = 1
j = 10
i = 10

f, axes = plt.subplots(1, 2, sharey=True, figsize=(12,7))
f.tight_layout()

plt.subplot(1, 2, 1)
plt.plot(tendS[t,:,j,i], tendS.Z, lw=4, color='blue', marker='.',label='total tendency')
plt.plot(forcS[t,:,j,i], forcS.Z, lw=2, color='red', marker='.',label='forcing')
plt.plot(adv_ConvS[t,:,j,i], adv_ConvS.Z, lw=2, color='orange', marker='.',label=
```

(continues on next page)

(continued from previous page)

```
↪'advection')
plt.plot(dif_ConvS[t,:,j,i], dif_ConvS.Z, lw=2, color='purple', marker='.',label=
↪'diffusion')
plt.legend(loc='lower left', frameon=False, fontsize=14)
plt.ylim([-1000,0])
#plt.xlim([-2e-10,2e-10])

plt.subplot(1, 2, 2)
plt.axvline(x=0, ymin=0, ymax=1, linewidth=0.5, color = 'k')
plt.plot(totalS[t,:,j,i], totalS.Z, lw=4, color='red', marker='.',label='RHS')
plt.plot(tendS[t,:,j,i], tendS.Z, lw=2, color='blue', marker='.',label='LHS')
plt.plot(totalS[t,:,j,i]-tendS[t,:,j,i], tendS.Z, lw=2, color='k', marker='.',label='RHS␣
↪- LHS')
plt.setp(plt.gca(), 'yticklabels',[])
plt.legend(loc='lower left',frameon=False,fontsize=14)
plt.ylim([-1000,0])
#plt.xlim([-2e-10,2e-10])

plt.show()
```



1.24.6 Evaluating the salinity budget

$$G^{Sln,tot} = G^{Sln,adv} + G^{Sln,forc} + G^{Sln,diff}$$

$$\frac{\partial S}{\partial t} = \frac{1}{s^*}\left[S\,\nabla_{z^*}(s^*\,\mathbf{v}) + S\,\frac{\partial w}{\partial z^*} - \nabla_{z^*}(s^*\,S\,\mathbf{v_{res}}) - \frac{\partial S\,w_{res}}{\partial z^*}\right] + D_S + F_S - S\,F$$

Scale factor

- Depth: Ocean_depth (m)
- ETAN: Surface Height Anomaly (m)

[103]:
```
# Load monthly averaged surface height anomaly (here only one face is used)
ETAN = ds_ave.sel(face=1).ETAN.load()
```

[104]:
```
# Scale factor
rstarfac = ((Depth + ETAN)/Depth).transpose('time','j','i')
```

Total tendency

[105]:
```
# Total tendency (psu/month)
tendSln_perMonth = (SALTsnp.shift(time=-1)-SALTsnp)[:-1]
```

[106]:
```
# Make sure time axis is the same as for the monthly variables
tendSln_perMonth.time.values = ds_ave.time[1:-1].values
```

[107]:
```
# Total tendency (psu/s)
tendSln_perSec = tendSln_perMonth/dt
```

[108]:
```
# Predefine tendSln array with correct dimensions
tendSln = xr.DataArray(np.nan*np.zeros([np.shape(tendSln_perSec)[0]+2,50,90,90]),
                       coords={'time': range(np.shape(tendSln_perSec)[0]+2),'k': np.
↪array(range(0,50)),
                               'j': np.array(range(0,90)),'i': np.array(range(0,90))},
↪dims=['time','k','j','i'])

# Time
tendSln.time.values = ds_ave.time.values

# Add coordinates
tendSln['XC'] = ds_snp.XC.sel(face=1)
tendSln['YC'] = ds_snp.YC.sel(face=1)
tendSln['Z'] = ds_snp.Z
```

[109]:
```
# Total tendency (psu/s)
tendSln.values[1:-1] = tendSln_perSec.values
```

Forcing

Note: The forcing term is comprised of both salt flux (`forcS`) and volume (i.e., freshwater) flux (`forcV`). - SALT: Salinity (psu)

[110]:
```
# Load monthly averaged salinity fields (here only one face is used)
SALT = ds_ave.sel(face=1).SALT.load()
```

[111]:
```
# Sea surface forcing for salinity (psu/s)
forcSln = (-SALT*forcV + forcS)/rstarfac
```

Advection

Horizontal convergence

[112]:
```
adv_hConvSln = (-SALT*hConvV + adv_hConvS)/rstarfac
```

Vertical convergence

[113]:
```
adv_vConvSln = (-SALT*vConvV + adv_vConvS)/rstarfac
```

Diffusion

Horizontal convergence

[114]:
```
dif_hConvSln = dif_hConvS/rstarfac
```

Vertical convergence

[115]:
```
dif_vConvSln = dif_vConvS/rstarfac
```

Total convergences

[116]:
```
# Total convergence of advective flux
adv_ConvSln = adv_hConvSln + adv_vConvSln

# Total convergence of diffusive flux
dif_ConvSln = dif_hConvSln + dif_vConvSln

# Total convergence
ConvSln = adv_ConvSln + dif_ConvSln
```

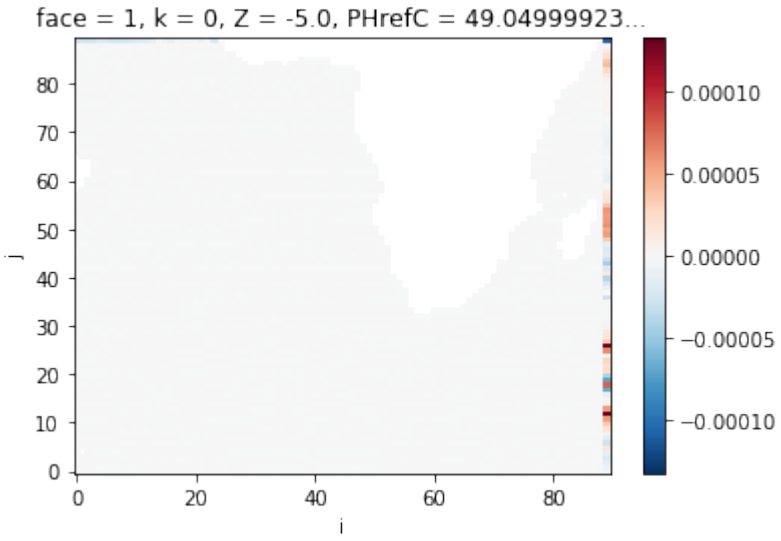
Total tendency

[117]:
```
totalSln = ConvSln + forcSln
```

Plot Accumulated residual in salinity budget

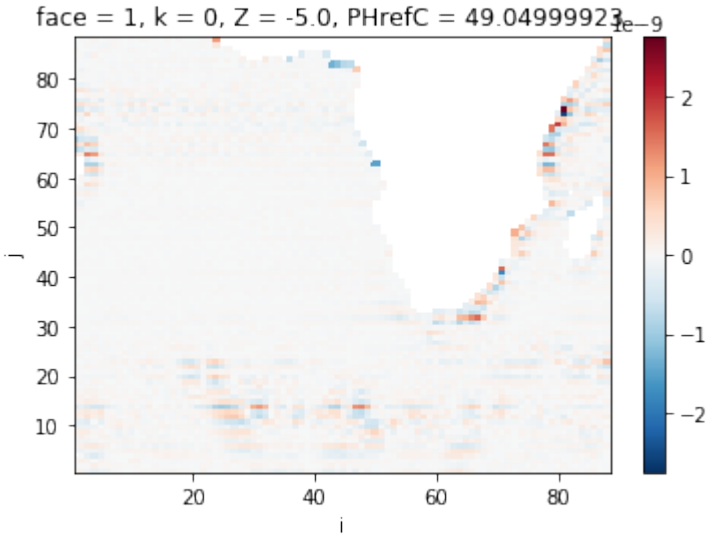
[118]:
```
((totalSln-tendSln).sum(dim='k').sum(dim='time')*land_mask).plot(cmap='RdBu_r')
```

[118]:
```
<matplotlib.collections.QuadMesh at 0x2aae75c5c750>
```



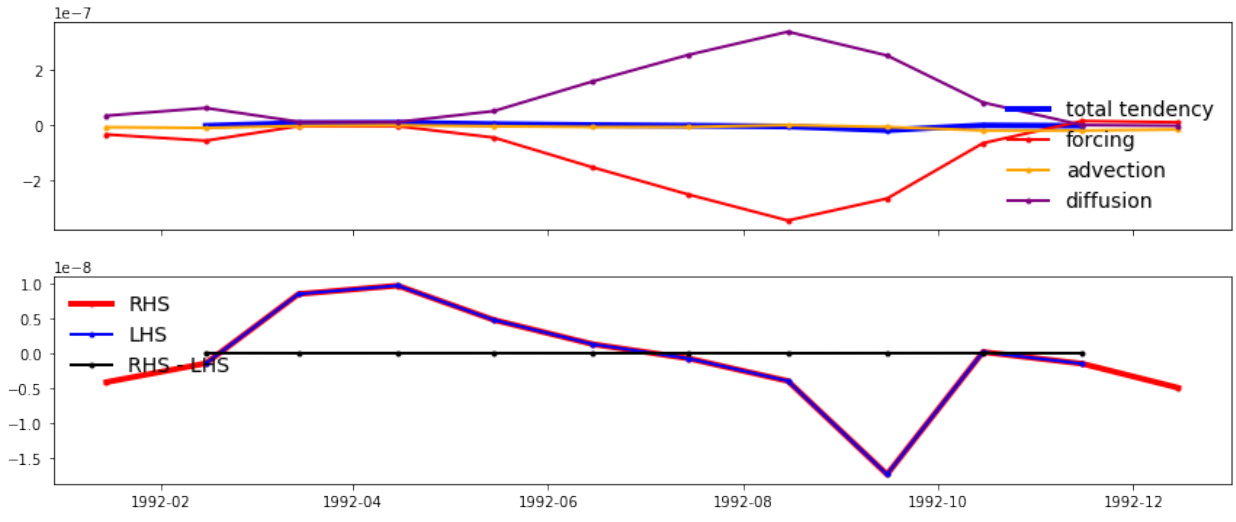
[119]:
```
# Ignoring face boundaries
((totalSln-tendSln).sum(dim='k').sum(dim='time')*land_mask)[1:89,1:89].plot(cmap='RdBu_r
↪')
```

[119]:
```
<matplotlib.collections.QuadMesh at 0x2aae5f3b2f10>
```



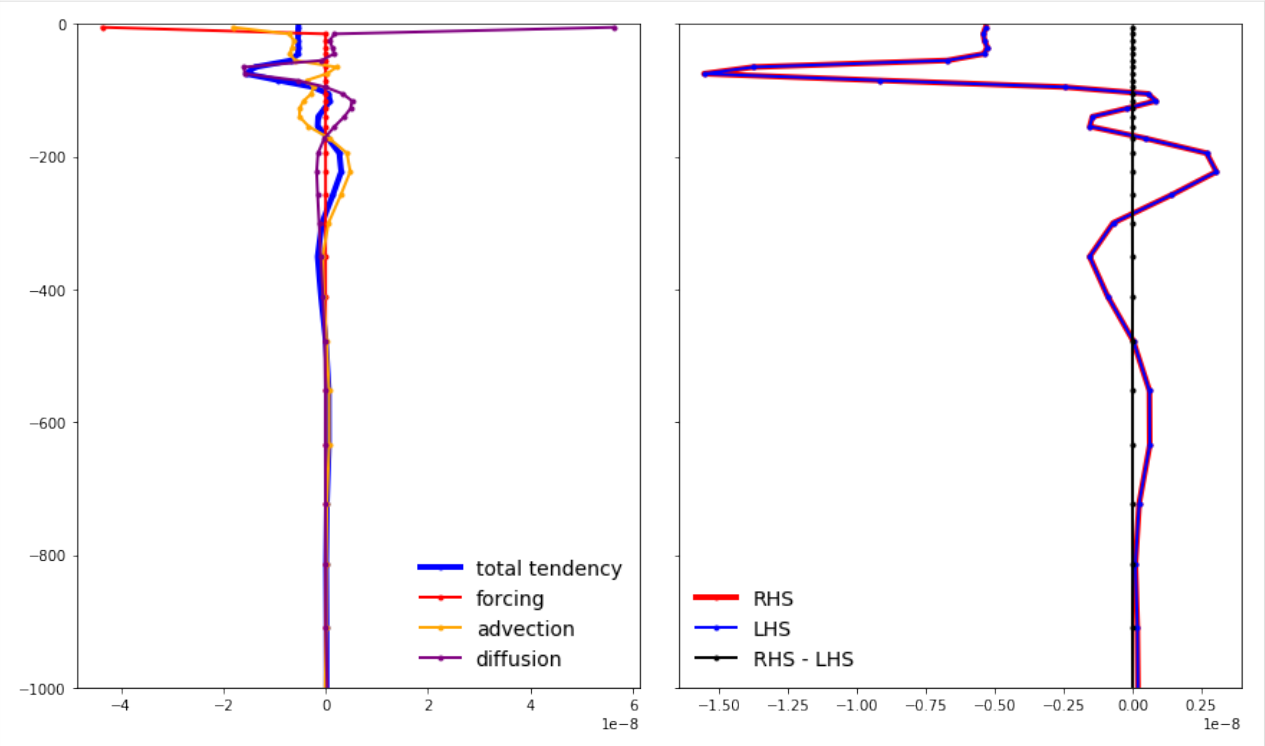
Time series for an arbitrarily chosen grid point

[120]:
```
k = 0
j = 0
i = 10

f, axes = plt.subplots(2, 1,figsize=(12,5))
f.tight_layout()

plt.subplot(2, 1, 1)
plt.plot(tendSln.time, tendSln[:,k,j,i], lw=4, color='blue', marker='.',label='total␣
↪tendency')
plt.plot(forcSln.time, forcSln[:,k,j,i], lw=2, color='red', marker='.',label='forcing')
plt.plot(adv_ConvSln.time, adv_ConvSln[:,k,j,i], lw=2, color='orange', marker='.',label=
↪'advection')
plt.plot(dif_ConvSln.time, dif_ConvSln[:,k,j,i], lw=2, color='purple', marker='.',label=
↪'diffusion')
plt.setp(plt.gca(), 'xticklabels',[])
plt.legend(loc='lower right',frameon=False,fontsize=14)

plt.subplot(2, 1, 2)
plt.plot(totalSln.time, totalSln[:,k,j,i], lw=4, color='red', marker='.',label='RHS')
plt.plot(tendSln.time, tendSln[:,k,j,i], lw=2, color='blue', marker='.',label='LHS')
plt.plot(tendSln.time, totalSln[:,k,j,i]-tendSln[:,k,j,i], lw=2, color='k', marker='.',
↪label='RHS - LHS')
plt.legend(loc='upper left',frameon=False,fontsize=14)

plt.show()
```



Verical profiles for an arbitrarily chosen grid point

[121]:
```
t = 1
j = 10
i = 10

f, axes = plt.subplots(1, 2, sharey=True, figsize=(12,7))
f.tight_layout()

plt.subplot(1, 2, 1)
plt.plot(tendSln[t,:,j,i],tendSln.Z, lw=4, color='blue', marker='.',label='total tendency
↪')
plt.plot(forcSln[t,:,j,i],forcSln.Z, lw=2, color='red', marker='.',label='forcing')
plt.plot(adv_ConvSln[t,:,j,i],adv_ConvSln.Z, lw=2, color='orange', marker='.',label=
↪'advection')
plt.plot(dif_ConvSln[t,:,j,i],dif_ConvSln.Z, lw=2, color='purple', marker='.',label=
↪'diffusion')
plt.legend(loc='lower right',frameon=False,fontsize=14)
plt.ylim([-1000,0])

plt.subplot(1, 2, 2)
plt.axvline(x=0, ymin=0, ymax=1, linewidth=0.5, color = 'k')
plt.plot(totalSln[t,:,j,i],totalSln.Z, lw=4, color='red', marker='.',label='RHS')
plt.plot(tendSln[t,:,j,i],tendSln.Z, lw=2, color='blue', marker='.',label='LHS')
plt.plot(totalSln[t,:,j,i]-tendSln[t,:,j,i],tendSln.Z, lw=2, color='k', marker='.',label=
↪'RHS - LHS')
plt.setp(plt.gca(), 'yticklabels',[])
plt.legend(loc='lower left',frameon=False,fontsize=14)
plt.ylim([-1000,0])

plt.show()
```



[ ]:

1.25 Getting Help

1.25.1 The ECCO Support Mailing List

For questions or comments please contact us via: ecco-support@mit.edu

1.25.2 Problems installing Python libraries?

Sometimes the pip Python package manager doesn't download the latest version of libraries from pypi, particlar on Mac/osX systems, because old versions of the libraries can be stored in the local cache directory. Deleting the pip cache directory and always using the "–no-cache-dir" option when downloading/installing packages solves this problem.

```
rm -fr ~/Library/Caches/pip/*
pip install ecco_v4_py --no-cache-dir
```

Sometimes libraries get updated. To update packages installed with conda use the the following command:

```
conda update anaconda
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search