# Analyzing ESM data with python Documentation

***Release 0.1***

**Alistair Adcroft**

**Feb 03, 2019**

# Contents:


**1 Introduction** **1**
- 1.1 How to use these notes . . . 1
- 1.2 Contributing . . . 2
- 1.3 Credits . . . 2

**2 Basics of accessing data** **3**
- 2.1 Synopsis . . . 3
- 2.2 About netcdf . . . 3
- 2.3 Opening a dataset . . . 4
- 2.4 Variables . . . 4
- 2.5 Reading all or part of a 1-d data array . . . 6
- 2.6 Shape of data . . . 7
- 2.7 Reading multi-dimensional data . . . 7
- 2.8 Summary . . . 8

**3 Basic visualization of 2d data** **11**
- 3.1 Synopsis . . . 11
- 3.2 pyplot from matplotlib . . . 11
- 3.3 Summary . . . 16

**4 Problems with curvilinear/spherical coordinates** **17**
- 4.1 Synopisis . . . 17
- 4.2 A look at some model output on a tri-polar grid . . . 17
- 4.3 Plot without physical coordinates (plot with data-indices) . . . 18
- 4.4 Plot using projection coordinates . . . 19
- 4.5 Plot using geographic coordinates . . . 20
- 4.6 Mesh vs data coordinates (cell bounds) . . . 21
- 4.7 Plotting cell-by-cell with geographic coordinates (to illustrate a problem) . . . 22
- 4.8 Extracting and plotting with geographic mesh coordinates . . . 24
- 4.9 Comments . . . 27
- 4.10 Summary . . . 27

**5 Visualizing spherical data using cartopy** **29**
- 5.1 Synopsis . . . 29
- 5.2 Get going with cartopy . . . 29
- 5.3 Extract geographic mesh coordinates . . . 30
- 5.4 The Plate-Carree projection . . . 30

5.5 Other projections . . . 31
5.6 Avoiding re-organization of the mesh data . . . 34
5.7 Summary . . . 35

**6 Reading and plotting with xarray** **37**
6.1 Synopsis . . . 37
6.2 xarray . . . 37
6.3 Summary . . . 40

**7 Reading and plotting with Iris** **41**
7.1 Synopsis . . . 41
7.2 Iris . . . 41
7.3 Summary . . . 43

**8 Deeper dive into pcolormesh()** **45**
8.1 Synopsis . . . 45
8.2 Some test data . . . 45
8.3 Simple pcolormesh via matplotlib.pyplot . . . 46
8.4 pcolormesh via cartopy . . . 46
8.5 pcolormesh via xarray . . . 47
8.6 Summary . . . 49

**9 Coordinates, Projections, and Grids** **51**
9.1 Synopisis . . . 51
9.2 Coordinate systems . . . 51
9.3 Projections . . . 52
9.4 ESM grids . . . 54
9.5 Contents of grid files . . . 57
9.6 Summary . . . 57

**10 Operations on a grid** **59**
10.1 Synopsis . . . 59
10.2 Summary . . . 59

**11 Indices and tables** **61**

CHAPTER 1

# Introduction

The point of these notes is to provide a crash course in using python to examine numerical Earth System Model (ESM) output. They are also an excuse to evaluate and compare various packages that you might find useful in your analysis of ESM output. These notes are *not a recommendation about which packages you should or should not use*. All packages have advantages and drawbacks and, in my humble opinion, each are ideal for the particular tasks for which they were designed. Kudos to all the respective developers! I personally tend to use all the packages encountered in theses notes - I find being versatile helpful in what I do. You might prefer to choose one package for the comfort of familiarity and become expert with that one package. My recommendation: use whatever works for you.

## 1.1 How to use these notes

These notes are written using Jupyter notebooks. You can view or use them in multiple ways; interactively or passively, downloaded or on the web.

### 1.1.1 Interactive jupyter notebooks (recommended)

To be able to run these notebooks you need python (which is the point of the notes!) and jupyter (which sits above python and works with 200+ languages). You will have to clone the repository or download and unpack a zip file.

To clone using `git`, you will obviously need `git` installed. Then

```
git clone https://github.com/adcroft/Analyzing-ESM-data-with-python.git
```

To download a zip file go to https://github.com/adcroft/Analyzing-ESM-data-with-python and click the green "Clone or download" button and then "Download ZIP", or just click https://github.com/adcroft/Analyzing-ESM-data-with-python/archive/master.zip. Once downloaded, unzip the file contents.

Once cloned, or downloaded and unpacked, launch a jupyter notebook server with

```
jupyter notebook
```

which should open a browser window in a directory view. Navigate to the location of the downloaded/cloned notebooks and click a notebook to open it.

Within a notebook, use the "Run" button to step through each cell and execute. "CTRL-enter" will also run the currently selected cell, while "SHIFT-enter" will do the same but then move and select the next cell.

### 1.1.2 Non-interactive via the web (HTML, pdf, epub)

You can view these notebooks rendered on GitHub at https://github.com/adcroft/Analyzing-ESM-data-with-python . Click on one or each of the `.ipynb` files.

The notebooks have also been converted using *nbsphinx* and are hosted in various forms on *readthedocs*. To see these just follow one of these links:

- HTML
- pdf
- epub

## 1.2 Contributing

If you find these notes helpful, or error-ridden, feel free to fork the GitHub repository, make changes and let me know with a pull request. And feel free to give your self credit in the text!

## 1.3 Credits

- The Jupyter project has transformed the way we work - for the better!
- nbsphinx is a great tool that allows these notes to written in jupyter notebooks and rendered in many other formats.
- The Princeton University undergraduate project M6ASP with many thanks to @estherelle and @karnabhonil.

CHAPTER 2

# Basics of accessing data

## 2.1 Synopsis

- How to open a netcdf file or remote dataset with netCDF4
- List variables
- Query variable shape
- Read variable data

## 2.2 About netcdf

Most data used in the ESM world is gridded and stored using the netCDF format. Unfortunatly, weather models use other formats which we will simply have to ignore here. The netCDF format is a protocol for putting data in files which are not human readable but the protocol means you can read them on any computer with the appropriate (freely available) tools.

The most widely used python package for reading and writing netCDF files is "netCDF4" (obtained with `conda install netcdf4`). There are other packages but netCDF4 seems to be the most versatile and compatible with all formats. The "scipy.io.netcdf" package is included in scipy but is only compatible with version 3 files. Version 4 files use HDF5 under-the-hood for which netCDF4 works well.

Access the netCDF4 package, using the following (usually at the top of your notebook):

[1]:
```
import netCDF4
```

Here we will walk through some specific operations using netCDF4.Dataset. To find out more about the netCDF4 package you can do

```
help(netCDF4)
```

to get extensive information.

## 2.3 Opening a dataset

Before looking at model output, let's look at some regularly gridded data. We'll use data served by the Lamont-Doherty Earth Observatory who have a good collection of products and gridded data. You will need to be connected to the Internet to do the following:

[2]:
```
nc = netCDF4.Dataset('http://iridl.ldeo.columbia.edu/SOURCES/.NOAA/.NODC/.WOA05/.Grid-
↪5x5/.Annual/.mn/.temperature/dods')
```

Several things to note:

- The argument to `netCDF4.Dataset()` is a URL, rather than a file name. Often you will be working with local files in which case the argument would be a path to the file(s) of interest.
- The cell should have executed quickly (maybe with no delay at all). This is because `netCDF4.Dataset()` fetches/reads only the meta-data and not the actual data. Usually, the meta-data is small compared to the actual data.
- We stored the results of `netCDF4.Dataset()` in the variable `nc`. The name of this variable is arbitrary - we could just as easily have used `fred = netCDF4.Dataset(...)`. Here, I chose `nc` to be short for netCDF container.

Let's look at what `netCDF4.Dataset()` returned.

[3]:
```
print(nc)
```

```
<class 'netCDF4._netCDF4.Dataset'>
root group (NETCDF3_CLASSIC data model, file format DAP2):
    Conventions: IRIDL
    dimensions(sizes): X(72), Y(36), Z(33)
    variables(dimensions): float32 Y(Y), float32 Z(Z), float32 X(X), float32␣
↪temperature(Z,Y,X)
    groups:
```

`netCDF4.Dataset()` returned an object that is a "class" defined by the netCDF4 package. We can see the following details:

- The file format is actually using the older NETCDF3 protocol so scipy.io.netcdf would work on this file.
- The conventions used in the file are "IRIDL" meaning IRI Data Library. More on conventions later.
- The data could be up to three dimensional with the dimensions being "X", "Y", and "Z", and with lengths, 72, 36 and 33, respectively.
- There are four variables "X", "Y", "Z" and "temperature".
- "X", "Y", and "Z" are one-dimensional with the lengths of their namesake dimension, i.e. X has length X or 72. This is a part of the IRIDL convention and is shared with other conventions such as CF. Such **variables that share names with a dimension and with length of that dimension are referred to as "dimension variables"**. They are special and are convenient to use as coordinates when the coordinate system is orthogonal and Cartesian.
- "temperature" has dimensions "Z,Y,X" so is three dimensional. The order of dimensions matters.

## 2.4 Variables

Here's another way to see what variables are in the dataset:

[4]:
```
print(nc.variables)
```

```
OrderedDict([('Y', <class 'netCDF4._netCDF4.Variable'>
float32 Y(Y)
    pointwidth: 5.0
    standard_name: latitude
    gridtype: 0
    units: degree_north
unlimited dimensions:
current shape = (36,)
filling off
), ('Z', <class 'netCDF4._netCDF4.Variable'>
float32 Z(Z)
    positive: down
    gridtype: 0
    units: m
unlimited dimensions:
current shape = (33,)
filling off
), ('X', <class 'netCDF4._netCDF4.Variable'>
float32 X(X)
    pointwidth: 5.0
    standard_name: longitude
    gridtype: 1
    units: degree_east
unlimited dimensions:
current shape = (72,)
filling off
), ('temperature', <class 'netCDF4._netCDF4.Variable'>
float32 temperature(Z, Y, X)
    units: Celsius_scale
    missing_value: -99.9999
    long_name: Temperature
unlimited dimensions:
current shape = (33, 36, 72)
filling off
)])
```

We see that `nc.variables` is a python dictionary (`OrderedDict`) with keys "X", "Y", "Z" and "temperature", each pointing to a `netCDF4.Variable` object. Those objects contain information such as units, data shape, and so on, which is not displayed very cleanly.

Knowing that `nc.variables` is a python dictionary we can get a more concise summary with:

[5]:
```
print(nc.variables.keys())
```

```
odict_keys(['Y', 'Z', 'X', 'temperature'])
```

or

[6]:
```
for v in nc.variables:
    print(v)
```

```
Y
Z
X
temperature
```

To access one entry of a python dictionary use the `dict[key]` syntax. For `nc.variables` the keys are 'X', 'Y', 'Z' and 'temperature'.

Let's examine the 'Z' object:

[7]:
```
print(nc.variables['Z'])
```

```
<class 'netCDF4._netCDF4.Variable'>
float32 Z(Z)
    positive: down
    gridtype: 0
    units: m
unlimited dimensions:
current shape = (33,)
filling off
```

This is what we saw before when we looked at `nc.variables` but now we have a more manageable volume of information (just the one variable). - The data type is 32-bit float (or real*4 in Fortran). - Positive Z means downward (Z is more like depth!). - Z has units of meters ("m"). - There are 33 Z values.

## 2.5 Reading all or part of a 1-d data array

So far we have only seen meta-data, either about the file or about variables within the file.

To see actual values of a variable we use python slices (i.e. things like `[:]`, or `[:2]`). Remember that when using the `[s:e]` notation, the vector starts with index "s" but stops before index "e". A missing index implies the full extent.

For an one-dimensional array or list, A, with size n: - `A[s:e]` gives elements s, s+1, ..., e-2, e-1 - `A[:e]` gives elements 0, 1, ..., e-2, e-1 - `A[s:]` gives elements s, s+1, ..., n-2, n-1 - `A[:]` gives elements 0, 1, ..., n-2, n-1 - `A[:-1]` gives elements 0, 1, ..., n-3, n-2 - `A[-3:]` gives elements n-3, n-2, n-1 - `A[-4:-1]` gives elements n-4, n-3, n-2

Only now, when we specify a slice of data, will the data values actually be read by python:

[8]:
```
nc.variables['Z'][:]
```

[8]:
```
masked_array(data=[   0.,   10.,   20.,   30.,   50.,   75.,  100.,  125.,
                    150.,  200.,  250.,  300.,  400.,  500.,  600.,  700.,
                    800.,  900., 1000., 1100., 1200., 1300., 1400., 1500.,
                   1750., 2000., 2500., 3000., 3500., 4000., 4500., 5000.,
                   5500.],
             mask=False,
       fill_value=1e+20,
            dtype=float32)
```

The returned data is in the form of a "masked_array" which is a numpy class (https://docs.scipy.org/doc/numpy/reference/maskedarray.html). This means the file/variable can support missing data. For coordinate data you more often get a simple "array" (https://docs.scipy.org/doc/numpy/reference/arrays.html).

Masked arrays have a .mask attribute which is usually a boolean array. Where True, the data is masked out or missing.

The fill_value is the value the array will appear to have where it is missing. You should never need to know this but when plotting data, if you ever see scales up to "1e+20" then the masking isn't working properly.

*Note that I omitted the ''print()'' command.* This is a convenient feature but be warned jupyter only displays the results of the last command. To be sure of seeing output it is generally safer to always use `print()`. However, the output is sometimes different as we can see here:

[9]:
```
print(nc.variables['Z'][:])
```

```
[   0.   10.   20.   30.   50.   75.  100.  125.  150.  200.  250.  300.
  400.  500.  600.  700.  800.  900. 1000. 1100. 1200. 1300. 1400. 1500.
 1750. 2000. 2500. 3000. 3500. 4000. 4500. 5000. 5500.]
```

Aside: *The display of ‘‘nc.variables[‘Z’][:]‘‘ is dependent on context: within a print or ‘‘__str__()‘‘ context then just the values are returned, but for the ‘‘__repr__()‘‘ context the underlying object is returned. This is a detail of interest only if you are building/extending classes and likely of no use to you other than as an explanation of why there is a difference between the commands ‘‘print(nc.variables[‘Z’][:])‘‘ and ‘‘nc.variables[‘Z’][:]‘‘.*

To read only the first five depths:

[10]:
```
print(nc.variables['Z'][:5])
```

```
[ 0. 10. 20. 30. 50.]
```

The last value can be found using negative indexing:

[11]:
```
print(nc.variables['Z'][-1])
```

```
5500.0
```

Python uses C-conventions so the first value is obtained with index 0:

[12]:
```
print(nc.variables['Z'][0])
```

```
0.0
```

## 2.6 Shape of data

numpy arrays have shape and the `.shape` attribute returns a tuple with the shape of the variable or data. `.shape` works on netCDF4.Variable objects without reading the data. Here are some examples:

[13]:
```
print( "nc.variables['Z'].shape =", nc.variables['Z'].shape )
print( "nc.variables['Z'][:].shape =", nc.variables['Z'][:].shape )
print( "nc.variables['Z'][:5].shape =", nc.variables['Z'][:5].shape )
print( "nc.variables['Z'][2:5].shape =", nc.variables['Z'][2:5].shape )
```

```
nc.variables['Z'].shape = (33,)
nc.variables['Z'][:].shape = (33,)
nc.variables['Z'][:5].shape = (5,)
nc.variables['Z'][2:5].shape = (3,)
```

## 2.7 Reading multi-dimensional data

Now we’ll look at the shape of slices from the three dimensional variable:

[14]:
```
nc.variables['temperature'].shape
```

[14]:
```
(33, 36, 72)
```

It is generally good practice to **not** load all data unnecessarily. The above command returned the shape without reading all the data into memory. The following reads all the data and then returns the shape:

[15]:
```
nc.variables['temperature'][:,:,:].shape
```

```
[15]: (33, 36, 72)
```

The multiple colons are explicitly saying "all elements" along each dimension for which there is a colon (three dimensions in this case).

The same can be managed with *implicit* ranges simply by omission:

```
[16]: nc.variables['temperature'][:,:].shape
[16]: (33, 36, 72)
```

```
[17]: nc.variables['temperature'][:].shape
[17]: (33, 36, 72)
```

So not specifying the slice on a dimensions defaults to the whole dimension.

Now suppose we want a multi-dimensional subset of the data. The following returns the shape of surface temperature because:

- the first dimension for "temperature" is Z;
- Z points downward so the first value is the shallowest;
- the first index in a dimension is 0.

```
[18]: nc.variables['temperature'][0,:,:].shape
[18]: (36, 72)
```

Because of implicit slices on omitted dimensions, the following give the same result:

```
[19]: nc.variables['temperature'][0,:].shape
[19]: (36, 72)
```

```
[20]: nc.variables['temperature'][0].shape
[20]: (36, 72)
```

We can do the same along other dimensions. To sample all data corresponding to the 5'th Y element:

```
[21]: nc.variables['temperature'][:,4,:].shape
[21]: (33, 72)
```

```
[22]: nc.variables['temperature'][:,4].shape
[22]: (33, 72)
```

As an extra, if you want implicit ":" for all but the last dimensions then the following works:

```
[23]: nc.variables['temperature'][...,0].shape
[23]: (33, 36)
```

## 2.8 Summary

- Open a file with `nc = netCDF4.Dataset(filename)` or a remote dataset with `nc = netCDF4.Dataset(url)`

- Look at the list of variables with `print(nc.variables)` or `print(nc.variables.keys()`
- Access a variable with `var = nc.variables['var']`
- Look at the shape of a variables with `var.shape` or `nc.variables['var'].shape`
- Read the variable data with `var[:]` or `nc.variables['var'][2,:]`

# CHAPTER 3

# Basic visualization of 2d data

## 3.1 Synopsis

- Import matplotlib.pyplot
- Contour and shade 2d data

[1]:
```
import netCDF4
import numpy
```

## 3.2 pyplot from matplotlib

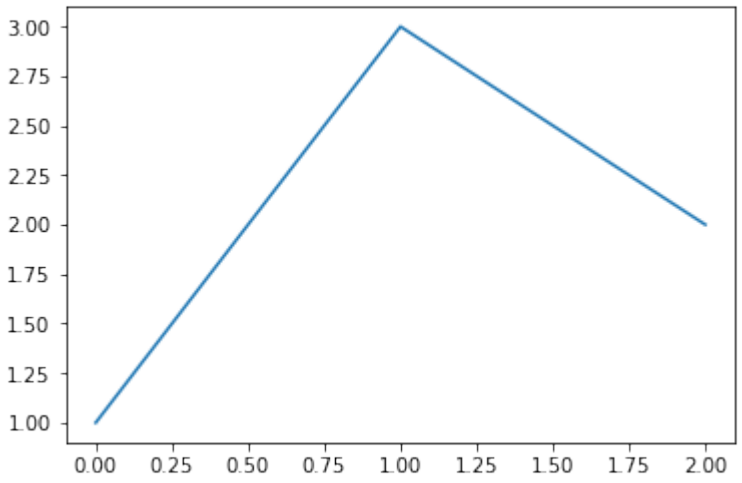
Now let's visualize the data we now know how to read. We've already imported the netCDF4 package above. Now we will import pyplot which is a sub-package of matplotlib.

We could just import the package like this

[2]:
```
import matplotlib.pyplot
```

but to use it is rather cumbersome

[3]:
```
matplotlib.pyplot.plot([1,3,2]);
```




So we will import pyplot with an alias "plt" as follows:

[4]:
```
import matplotlib.pyplot as plt
```

Here, plt is a shorthand for pyplot that *is widely used* but you can use whatever alias you like.

Normally, we would put this import statement at the top of the notebook with the other imports.

Let's look at the 2005 World Ocean Atlas temperature data on a 5°x5° grid, served from IRIDL:

[5]:
```
nc = netCDF4.Dataset('http://iridl.ldeo.columbia.edu/SOURCES/.NOAA/.NODC/.WOA05/.Grid-
↪5x5/.Annual/.mn/.temperature/dods')
```

And let's load the coordinates and sea-surface temperature into variables:

[6]:
```
lon = nc.variables['X'][:]
lat = nc.variables['Y'][:]
sst = nc.variables['temperature'][0,:,:]
```

Note that the `[:]` forced the reading of the data. Leaving the `[:]` out would have returned an object and deferred reading of the data which is generally considered better form. Here, I chose to force read so that the data is not repeatedly fetch across the Internet connection.

There are two main ways of looking at 2D data.

1. Contours

- `plt.contour()` Draw contours of constant values. Contours are colored by default.
- `plt.contourf()` Draws bands of constant color between contours of constant value.

Use `plt.clabel()` to label contours. 2. Psuedo-color shading of value at data locations. - `plt.pcolor()` Colored pixels for each data value. - `plt.pcolormesh()` is optimized for quadrilateral data and thus faster. It also understands curvilinear grids better than pcolor().

Use `plt.colorbar()` to add a color bar.

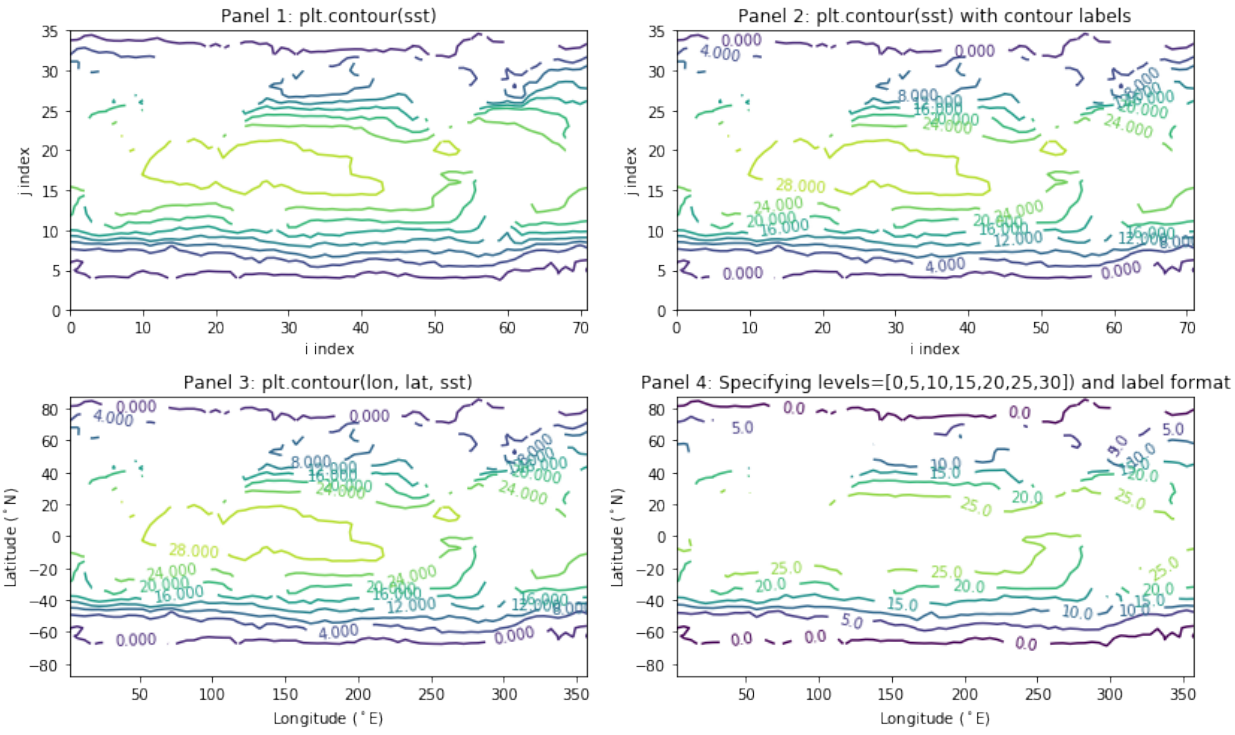
Here are four variants using `plt.contour()` with different arguments (given in the title of each pane):

[7]:
```
plt.figure(figsize=(12,7)); # Makes figure large enough to see four panels (optional)

# 1. Contour without coordinate value
```

(continues on next page)

(continued from previous page)

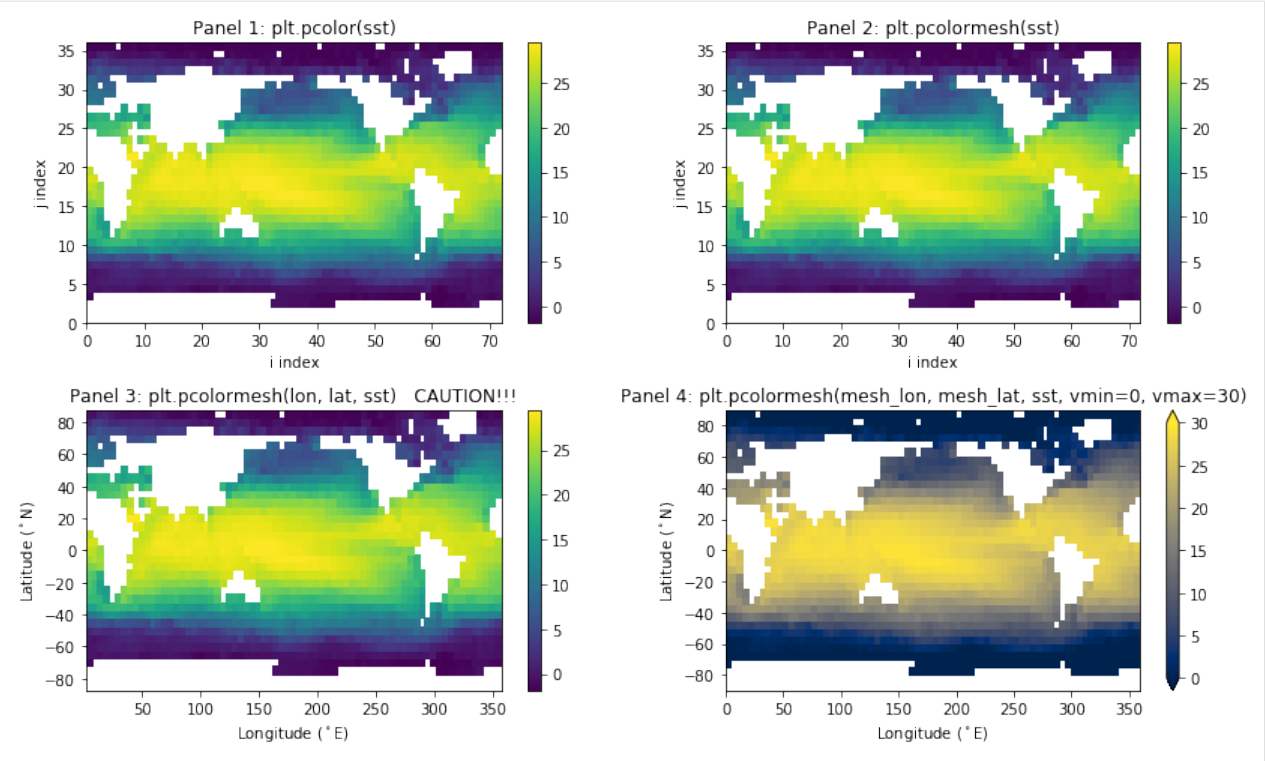
```
plt.subplot(221);
plt.contour(sst);
plt.xlabel('i index');
plt.ylabel('j index');
plt.title('Panel 1: plt.contour(sst)');

# 2. As above but with contour labels
plt.subplot(222);
c = plt.contour(sst);
plt.clabel(c);
plt.xlabel('i index');
plt.ylabel('j index');
plt.title('Panel 2: plt.contour(sst) with contour labels');

# 3. Contour with coordinates
plt.subplot(223);
c = plt.contour(lon, lat, sst);
plt.clabel(c);
plt.xlabel('Longitude ($^\circ$E)');
plt.ylabel('Latitude ($^\circ$N)');
plt.title('Panel 3: plt.contour(lon, lat, sst)');

# 4. Contour with coordinates and specific contour levels
plt.subplot(224);
c = plt.contour(lon, lat, sst, levels=[0,5,10,15,20,25,30]);
plt.clabel(c, fmt='%.1f');
plt.xlabel('Longitude ($^\circ$E)');
plt.ylabel('Latitude ($^\circ$N)');
plt.title('Panel 4: Specifying levels=[0,5,10,15,20,25,30]) and label format');

plt.tight_layout()
```



And here are variants of `plt.pcolor()` and `plt.pcolormesh()`.

In the following examples you will not notice any difference between pcolor and pcolormesh but pcolormesh is the preferred method for i) efficiency and ii) flexibility with curvilinear grids.

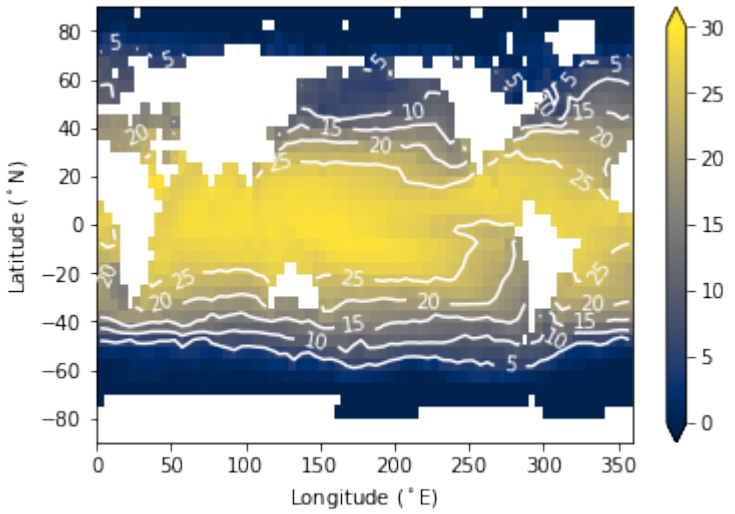
[8]:
```
plt.figure(figsize=(12,7)); # Makes figure large enough to see four panels (optional)

# 1. Simple pcolor
plt.subplot(221);
plt.pcolor(sst);
plt.colorbar();
plt.xlabel('i index');
plt.ylabel('j index');
plt.title('Panel 1: plt.pcolor(sst)');

# 2. Simple pcolormesh
plt.subplot(222);
plt.pcolormesh(sst);
plt.colorbar();
plt.xlabel('i index');
plt.ylabel('j index');
plt.title('Panel 2: plt.pcolormesh(sst)');

# 3. pcolormesh with cell-centered coordinate value
plt.subplot(223);
plt.pcolormesh(lon, lat, sst);
plt.colorbar();
plt.xlabel('Longitude ($^\circ$E)');
plt.ylabel('Latitude ($^\circ$N)');
plt.title('Panel 3: plt.pcolormesh(lon, lat, sst)   CAUTION!!!');

# 4. pcolormesh with mesh coordinates value
plt.subplot(224);
mesh_lon = numpy.linspace(0,360,lon.shape[0]+1)
mesh_lat = numpy.linspace(-90,90,lat.shape[0]+1)
plt.pcolormesh(mesh_lon, mesh_lat, sst, vmin=0, vmax=30, cmap=plt.cm.cividis);
plt.colorbar(extend='both');
plt.xlabel('Longitude ($^\circ$E)');
plt.ylabel('Latitude ($^\circ$N)');
plt.title('Panel 4: plt.pcolormesh(mesh_lon, mesh_lat, sst, vmin=0, vmax=30)');

plt.tight_layout()
```




**VERY IMPORTANT NOTE**: In panel 3 above, we pass the coordinates of the data locations to pcolormesh and you should **notice that the top row and right column of values are not plotted!!!** This is because pcolormesh expects the coordinates of the mesh, or cell boundaries, which should have one extra column and row of values than does the data. In panel 4, I make up appropriate mesh coordinates using `numpy.linspace()`.

To finish up, we can also combine pcolormesh and contour plots...

[9]:
```
mesh_lon = numpy.linspace(0,360,lon.shape[0]+1)
mesh_lat = numpy.linspace(-90,90,lat.shape[0]+1)
plt.pcolormesh(mesh_lon, mesh_lat, sst, vmin=0, vmax=30, cmap=plt.cm.cividis);
plt.colorbar(extend='both');
c = plt.contour(lon, lat, sst, levels=[5,10,15,20,25], colors='w')
plt.clabel(c, fmt='%.0f')
plt.xlabel('Longitude ($^\circ$E)');
plt.ylabel('Latitude ($^\circ$N)');
```




## 3.3 Summary

- Import matplotlib.pyplot with `import matplotlib.pyplot as plt`
- Contour with `plt.contour()` and `plt.contourf()`
- Shade with `plt.pcolormesh()`
- Also used `plt.colorbar()`, `plt.xlabel()`, `plt.ylabel()`, `plt.clabel()`

# CHAPTER 4

# Problems with curvilinear/spherical coordinates

## 4.1 Synopisis

- Illustrate problem when naively plotting data on curvilinear grids or in spherical coordinates.
- Describes difference between model and geographic coordinates.
- Describes need for mesh-coordinate vs data coordinates.
- Extraction of compact mesh coordinates from CF-standard cell bounds.

## 4.2 A look at some model output on a tri-polar grid

[1]:
```
import netCDF4
import numpy
import matplotlib.pyplot as plt

# These two packages will be used for illustrating the an inefficient cell-by-cell␣
↪plotting approach
import matplotlib.patches
import matplotlib.collections
```

Above I have imported the packages we encountered already which you will likely use often. The last two imports you will not often need and I import them here for the purposes of the following discussion.

Let's look at some model output that resides on the native curvilinear grid of the model.

[2]:
```
nc = netCDF4.Dataset('http://esgf-data2.diasjp.net/thredds/dodsC/esg_dataroot/CMIP6/
↪CMIP/MIROC/MIROC6/1pctCO2/r1i1p1f1/Omon/tos/gn/v20181212/tos_Omon_MIROC6_1pctCO2_
↪r1i1p1f1_gn_330001-334912.nc')
```

which contains the variables

[3]:
```
print( nc.variables.keys() )
```

```
odict_keys(['time', 'time_bnds', 'y', 'y_bnds', 'x', 'x_bnds', 'vertices_latitude',␣
↪'vertices_longitude', 'latitude', 'longitude', 'tos'])
```

This is a CMIP6 file and so contains only one model dynamic variable per file (here 'tos' is CMIP lingo for "Temperature of Ocean Surface") so all the other variables listed are forms of coordinates. Why so many? That is unfortunately part of the convention which is poorly designed for data on curvilinear meshes.

Here is a list of the variables with their symbolic netcdf dimensions and their actual shape:

[4]:
```
for v in nc.variables:
    print('{:21} dimensions = {!s:25} shape = {!s}'.format(v, nc.variables[v].
↪dimensions, nc.variables[v].shape))
```

```
time                  dimensions = ('time',)                shape = (600,)
time_bnds             dimensions = ('time', 'bnds')         shape = (600, 2)
y                     dimensions = ('y',)                   shape = (256,)
y_bnds                dimensions = ('y', 'bnds')            shape = (256, 2)
x                     dimensions = ('x',)                   shape = (360,)
x_bnds                dimensions = ('x', 'bnds')            shape = (360, 2)
vertices_latitude     dimensions = ('y', 'x', 'vertices')   shape = (256, 360, 4)
vertices_longitude    dimensions = ('y', 'x', 'vertices')   shape = (256, 360, 4)
latitude              dimensions = ('y', 'x')               shape = (256, 360)
longitude             dimensions = ('y', 'x')               shape = (256, 360)
tos                   dimensions = ('time', 'y', 'x')       shape = (600, 256, 360)
```

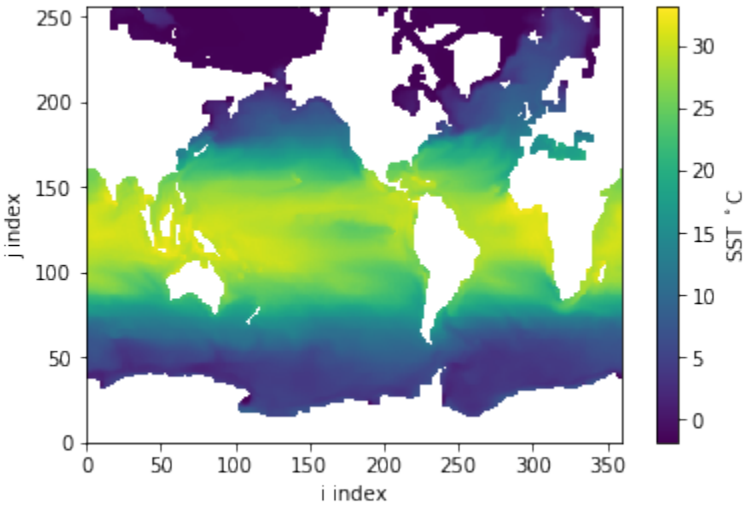
There are five netcdf dimensions:

- 'x', 'y' and 'time', are the shape of the data.
- 'time' is model time, usually given as a days since a date.
- 'x' and 'y' is a "model" coordinate and **NOT** geographic longitude and latitude, even if the units and values make it seem so.
- 'bnds' and 'verticies' are two other dimensions needed to provide extra information to deal with the deficiency of the data storage designed for providing data at locations rather than on finite volume mesh.
- 'bnds' is equal to 2 and is used in the *_bnds variables.
- 'vertices' is equal to 4 (this is a quadrilateral mesh) and used to provide the latitude and longitude of the corners of each cell.

## 4.3 Plot without physical coordinates (plot with data-indices)

Let's quickly look at the first time-slice of the variable of interest

[5]:
```
tos = nc.variables['tos'][0] # Pre-load data which will be re-used later

plt.pcolormesh( tos ); # Plotting without coordinate (for now)

plt.colorbar(label='SST $^\circ$C')
plt.xlabel('i index');
plt.ylabel('j index');
```




At first glance this looks like the planet's ocean *but* notice how Canada and Northern Russia reach the top edge? This data resides on a tri-polar curvilinear mesh so the Northern hemisphere is highly distorted if plotted naively (as done here).

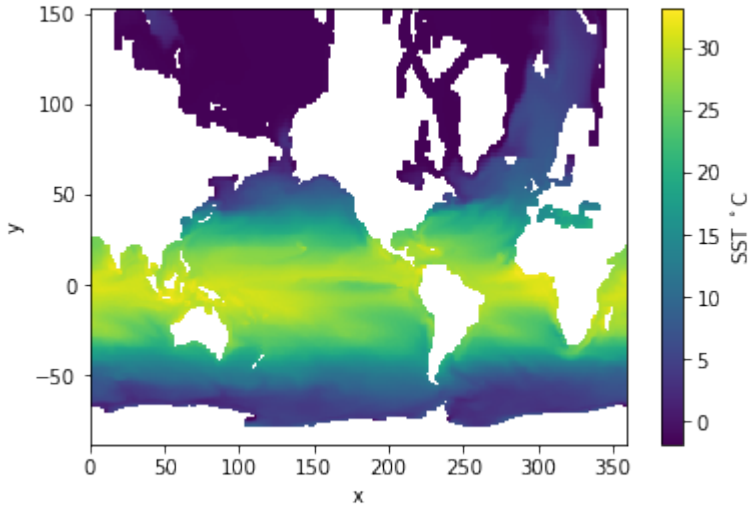
## 4.4 Plot using projection coordinates

What about those 1-dimensional coordinate variables 'x' and 'y'? Let's try them...

[6]:
```
print( nc.variables['x'] )
print( nc.variables['y'] )
```

```
<class 'netCDF4._netCDF4.Variable'>
float64 x(x)
    bounds: x_bnds
    units: degrees
    axis: X
    long_name: x coordinate of projection
    standard_name: projection_x_coordinate
unlimited dimensions:
current shape = (360,)
filling off

<class 'netCDF4._netCDF4.Variable'>
float64 y(y)
    bounds: y_bnds
    units: degrees
    axis: Y
    long_name: y coordinate of projection
    standard_name: projection_y_coordinate
unlimited dimensions:
current shape = (256,)
filling off
```

[7]:
```
plt.pcolormesh( nc.variables['x'][:], nc.variables['y'][:], tos ); # Plotting with
↪"projection coordinate"

plt.colorbar(label='SST $^\circ$C')
plt.xlabel('x');
plt.ylabel('y');
```




Using the "projection coordinate" has merely stretched things out. In principle, if we had code to map from a projection coordinate to geographic longitude and latitude then we could plot properly.

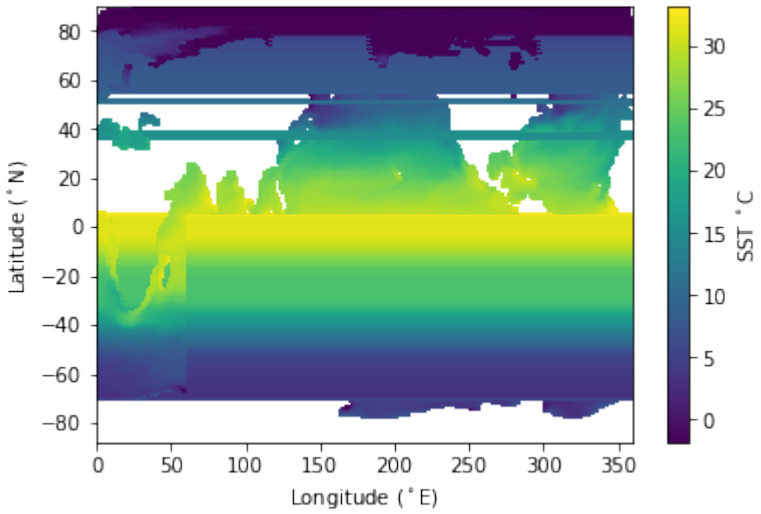
*Comment: Model meshes rarely use analytic projections but normally have numerically generated meshes so the notion of a providing projection coordinates for the user is somewhat useless in practice.*

## 4.5 Plot using geographic coordinates

The file also contains the geographic coordinates "longitude" and "latitude" which are each 2D data. Note that they have the same shape as each time-level of "tos" which means it will suffer from the missing row/column problem mention previously.

[8]:
```
# Pre-load data which will be re-used later
lon = nc.variables['longitude'][:]
lat = nc.variables['latitude'][:]
```

[9]:
```
plt.pcolormesh(lon, lat, tos)

plt.colorbar(label='SST $^\circ$C')
plt.xlabel('Longitude ($^\circ$E)');
plt.ylabel('Latitude ($^\circ$N)');
```




Obviously this didn't work well! I'll explain what's going wrong later but first we know should be using the mesh coordinates (cell corners) rather data coordinates (cell centers) so let's try that.

## 4.6 Mesh vs data coordinates (cell bounds)

Before moving onto using the actual mesh coordinates that are provided, a brief explanation about cell bounds versus coordinates. 'x_bnds' are the cell bounds for the 'x' coordinate variable:

[10]:
```
nc.variables['x_bnds']
```

[10]:
```
<class 'netCDF4._netCDF4.Variable'>
float64 x_bnds(x, bnds)
    _ChunkSizes: [360   2]
unlimited dimensions:
current shape = (360, 2)
filling off
```

[11]:
```
print( nc.variables['x'][:5] )
```

```
[0.5 1.5 2.5 3.5 4.5]
```

[12]:
```
print( nc.variables['x_bnds'][:5] )
```

```
[[0. 1.]
 [1. 2.]
 [2. 3.]
 [3. 4.]
 [4. 5.]]
```

We see that the first five values of 'x' are monotonically increasing and that each value of 'x' falls in between the bounds of the cell 'x_bnds'.

_Comment: There is redundancy in the 'x__bnds' data. There are 360 x 2 = 720 data values but only 361 distinct values because the cells are space-filling or contiguous - the right edge of one cell is the left edge of the next. The data could have been more compactly stored as a one dimensional list of 361 values. For efficient plotting in two dimensions, the more compact coordinate data will prove more useful._

Now look at the geographic longitude and the associated vertex data:

[13]:
```
print( nc.variables['longitude'] )
```

```
<class 'netCDF4._netCDF4.Variable'>
float32 longitude(y, x)
    standard_name: longitude
    long_name: longitude
    units: degrees_east
    missing_value: 1e+20
    _FillValue: 1e+20
    bounds: vertices_longitude
    _ChunkSizes: [256 360]
unlimited dimensions:
current shape = (256, 360)
filling off
```

[14]:
```
print( nc.variables['vertices_longitude'] )
```

```
<class 'netCDF4._netCDF4.Variable'>
float32 vertices_longitude(y, x, vertices)
    units: degrees_east
    missing_value: 1e+20
    _FillValue: 1e+20
    _ChunkSizes: [256 360   4]
unlimited dimensions:
current shape = (256, 360, 4)
filling off
```

Analogous to how pairs of 'x_bnd' reflect the cells surronding 'x', quads of 'vertices_longitude' describe the cells centered on 'longitude'. The same will be true for 'vertices_latitude' and 'latitude'. Here's some actual values for cell with index j=7,i=3:

[15]:
```
print( 'Longitude =', nc.variables['longitude'][7,3], 'Latitude =', nc.variables[
↪'latitude'][7,3] )
```

```
Longitude = 63.5 Latitude = -82.75
```

[16]:
```
print( nc.variables['vertices_longitude'][7,3,:], nc.variables['vertices_latitude'][7,
↪3,:] )
```

```
[63. 64. 64. 63.] [-83.  -83.  -82.5 -82.5]
```

Again we see that for a particular cell the longitude,latitude coordinates fall within the bounds of the cells given as the cell vertices.

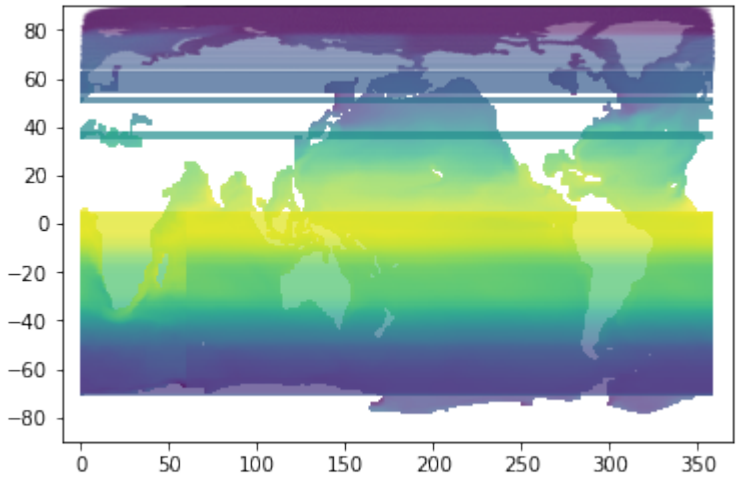
Observe that the vertices are given in an anti-clockwise order starting in the south-west (bottom-left) corner. We will later use this to construct a more compact mesh description.

## 4.7 Plotting cell-by-cell with geographic coordinates (to illustrate a problem)

The next two cells of this notebook use the vertex data to plot a polygon for each cell of data. **THIS IS NOT THE RECOMMENDED WAY TO PLOT** but is provided to show how tedious it is to use the mesh information in this format, *and* to illustrate some problems with the curvilinear mesh itself.

[17]:
```
# For efficiency, load relevant data into memory
vlon = nc.variables['vertices_longitude'][:]
vlat = nc.variables['vertices_latitude'][:]
```

[18]:
```
# A tedious way to plot, cell-by-cell
fig, ax = plt.subplots();
plt.xlim(-10,370);
plt.ylim(-90,90);
patches = []; vals = []
for j in range(nc.variables['y'].shape[0]):
    for i in range(nc.variables['x'].shape[0]):
        if not tos.mask[j,i]:
            llon = vlon[j,i,:]
            llat = vlat[j,i,:]
            xy = numpy.vstack([llon,llat])
            patches.append( matplotlib.patches.Polygon(xy.T) )
            vals.append( tos[j,i] )
p = matplotlib.collections.PatchCollection( patches, alpha=0.7)
p.set_array( numpy.array(vals) )
ax.add_collection(p);
```



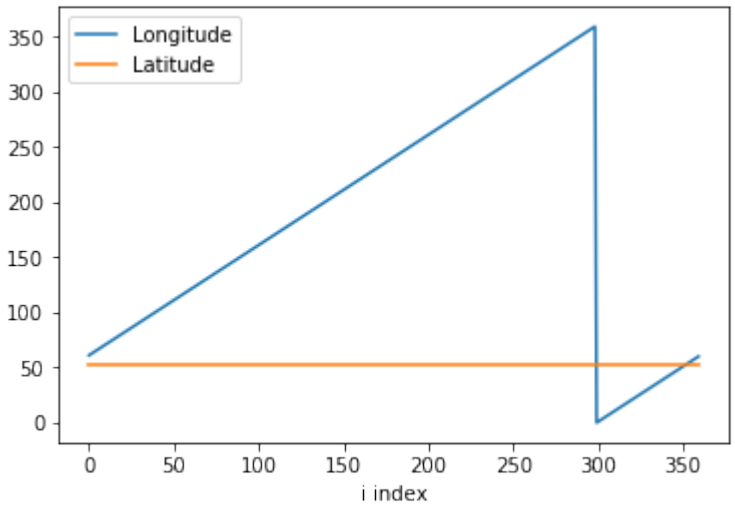

You might notice this took a while to plot - plotting cell-by-cell is very inefficient.

And despite all that code, the plot still looks no better than the `plt.pcolormesh(lon, lat, tos)` we tried above.

Here, I used a partially transparent "alpha" so you can see the continent outline. The continents are being overdrawn by cells whose longitudes "wrap around" from 360 to 0 and so span the entire plot. The longitudes in this mesh appear to be constrained to the range 0..360.

To see what is going wrong, here's the longitude,latitude (on the same vertical scale) for the NE vertex of cells at a constant j-index (of 190):

[19]:
```
plt.plot( vlon[190,:,2], label='Longitude' );
plt.plot( vlat[190,:,2], label='Latitude' );
plt.xlabel('i index');
plt.legend();
```




The latitudes are constant (for this j-index) and the longitudes increase linearly until about 360 where they jump down to near zero before increasing again. This is the periodicity of the longitudinal coordinate being manifest. 360 degrees east is equivalent to 0 degree east on the sphere but not on our 2D plots. So long as we are making the plots ourselves (as opposed to using a mapping package) we need to fix the coordinates.

## 4.8 Extracting and plotting with geographic mesh coordinates

Here's how we can try.

We'll make the following assumptions: - Starting vertex is the same for each cell (in this dataset it is the SW corner). - Direction of vertexes is the same for all cells (in this dataset it is counter-clockwise).

And we will make the following adjustments: - Create contiguous coordinate arrays for mesh vertexes without redundancy. - Adjust longitudes to not decrease with increasing i-index so as to not cross previous patches

[20]:
```
# First create the latitude array for the mesh (which needs no value adjustments)
flat = numpy.zeros( (nc.variables['y'].shape[0]+1, nc.variables['x'].shape[0]+1) ) #␣
↪1 larger than cell shape
flat[1:,1:] = vlat[:,:,2] # Copy NE corner values to mesh
flat[0,1:] = vlat[0,:,1] # Copy SE corner for bottom row only
flat[1:,0] = vlat[:,0,3] # Copy NW corner for left column only
flat[0,0] = vlat[0,0,0] # Copy SW corner for SW cell only
```

[21]:
```
# Now create the longitude array for the mesh
flon = numpy.zeros( (nc.variables['y'].shape[0]+1, nc.variables['x'].shape[0]+1) )
flon[1:,1:] = vlon[:,:,2] # Copy NE corner values to mesh
flon[0,1:] = vlon[0,:,1] # Copy SE corner for bottom row only
flon[1:,0] = vlon[:,0,3] # Copy NW corner for left column only
flon[0,0] = vlon[0,0,0] # Copy SW corner for SW cell only

# The following lines "fix" the longitude values so plotting works
for i in range(nc.variables['x'].shape[0]):
    dlon = flon[:,i+1] - flon[:,i]
    dlon = numpy.mod( dlon, 360)
    flon[:,i+1] = flon[:,i] + dlon # Monotonically increase
flon[-1,:] = numpy.round( (flon[-2,:]-60)/180 )*180+60 # Top row of longitudes need␣
↪special values!
```

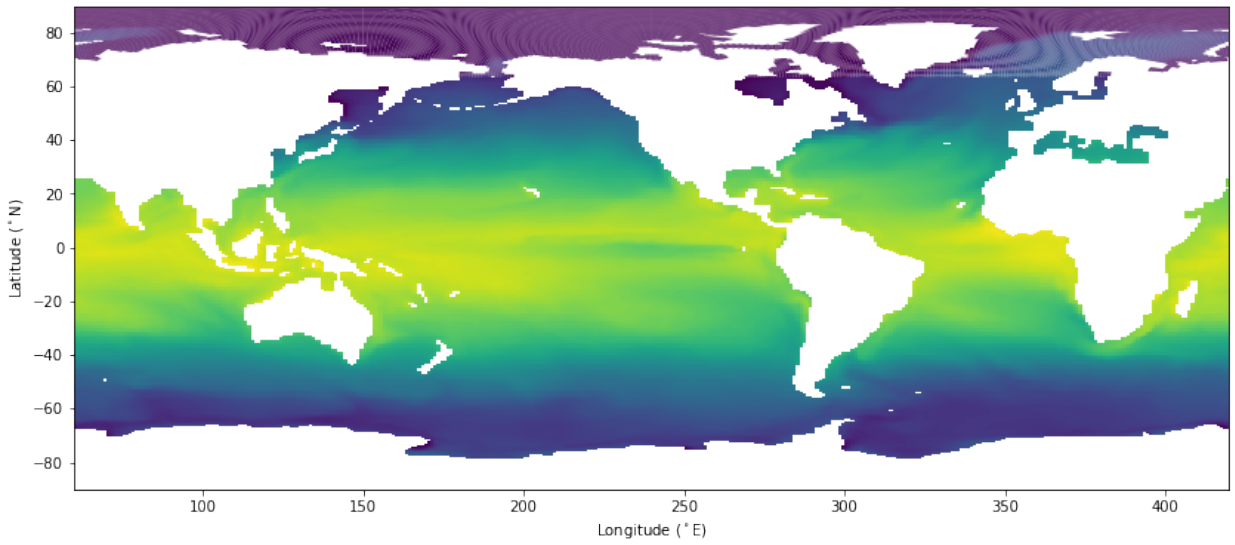
(continues on next page)

(continued from previous page)

Here, I made sure that longitude increase monotonically in the i-direction.

The following plots cell-by-cell with the adjusted coordinates... (**still not recommended as a plotting approach**).

[22]:
```
# Still a tedious way to plot, cell-by-cell
fig, ax = plt.subplots(figsize=(14,6));
plt.xlim(60,420);
plt.ylim(-90,90);
patches = []; vals = []
for j in range(nc.variables['y'].shape[0]):
    for i in range(nc.variables['x'].shape[0]):
        if not tos.mask[j,i]:
            llon = numpy.array([flon[j,i], flon[j,i+1], flon[j+1,i+1], flon[j+1,i]])
            llat = numpy.array([flat[j,i], flat[j,i+1], flat[j+1,i+1], flat[j+1,i]])
            xy = numpy.vstack([llon,llat])
            patches.append( matplotlib.patches.Polygon(xy.T) )
            vals.append( tos[j,i] )
p = matplotlib.collections.PatchCollection( patches, alpha=1)
p.set_array( numpy.array(vals) )
ax.add_collection(p);

plt.xlabel('Longitude ($^\circ$E)');
plt.ylabel('Latitude ($^\circ$N)');
```




The mis-coloring in the Arctic is a plotting artifact of the polygons, not a coordinate problem.

*Comment: There is no color bar shown because the code to create a color scale for the polygon shading requries too much code and would be obviscating.*

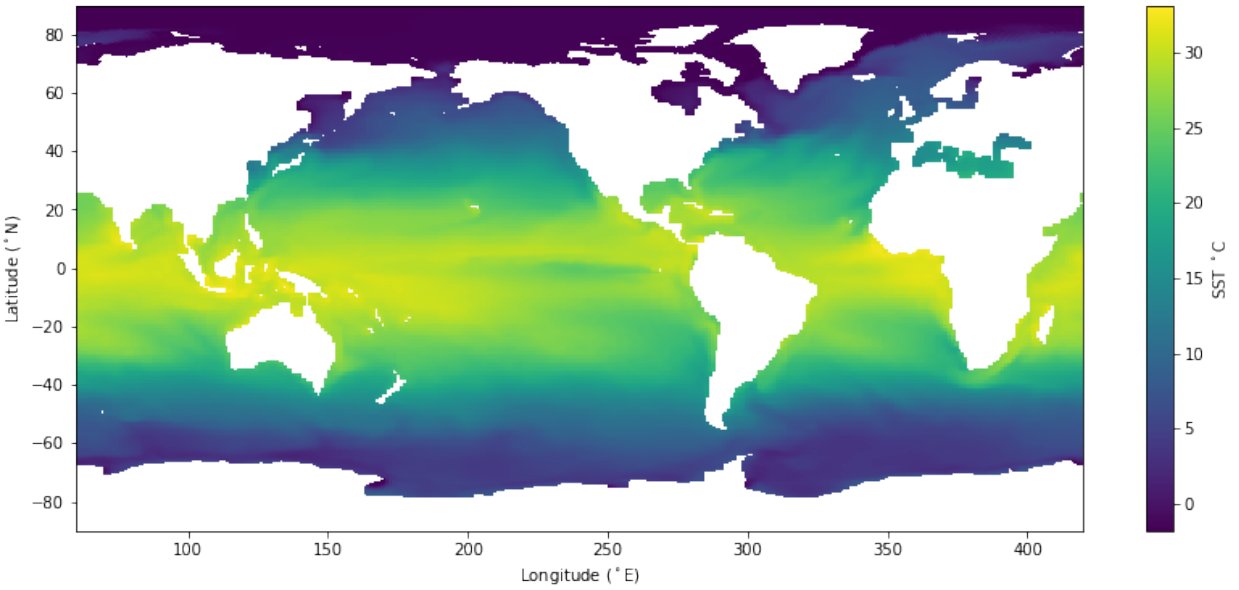
Here's the same plot but now using `pcolormesh()` that is much much more efficient (and is artifact free).

[23]:
```
# A much quicker way to make the same plot
plt.figure(figsize=(14,6))

plt.pcolormesh( flon, flat, tos);
```

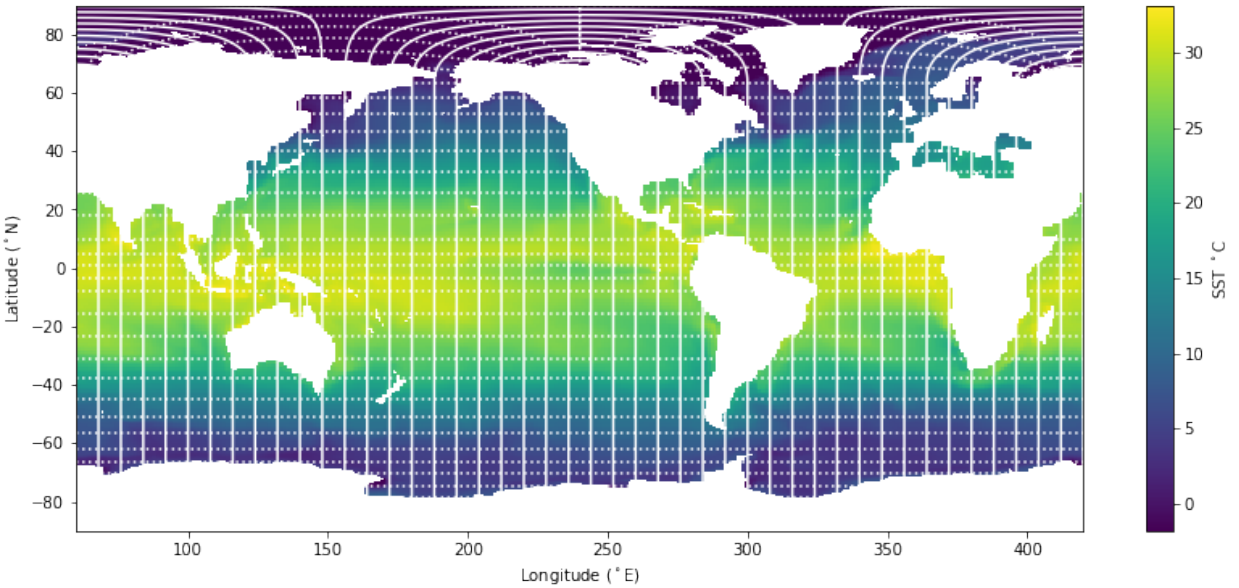
(continues on next page)

(continued from previous page)

```
plt.colorbar(label='SST $^\circ$C')
plt.xlabel('Longitude ($^\circ$E)');
plt.ylabel('Latitude ($^\circ$N)');
```




[24]:
```
# As above but with the mesh over drawn
plt.figure(figsize=(14,6))

plt.pcolormesh( flon, flat, tos);

plt.colorbar(label='SST $^\circ$C')
plt.xlabel('Longitude ($^\circ$E)');
plt.ylabel('Latitude ($^\circ$N)');

plt.plot( flon[:,::8], flat[:,::8], 'w-'); # Lines of constant i-index
plt.plot( flon[::8,:].T, flat[::8,:].T, 'w:'); # Lines of constant j-index
```




## 4.9 Comments

- "Fixing the coordinates" so that plotting works took two steps: i) extracting a compact set of coordinate data and ii) correcting values that can be plotted without over-drawing of data.
- The latter problem is a result of trying to plot spherical data on a 2D plane. This will be better dealt with using projections of the sphere onto a plane for which the cartopy package is designed (see next notebook).
- The reshaping of coordinate data is a necessary step because the CMOR file format is poorly designed. We will continue to reshape the vertex data in subsequent examples.

## 4.10 Summary

- Curvilinear meshes used by ocean models can be awkward stored in files, and to plot.
- Data in spherical coordinates plotted in 2d is often messed up by periodic longitude values.
- Creating some cleaned-up mesh coordinates is often necessary to use `plt.pcolormesh`.

CHAPTER 5

# Visualizing spherical data using cartopy

## 5.1 Synopsis

- Use projections to create 2d plots of spherical data

## 5.2 Get going with cartopy

[1]:
```
import netCDF4
import numpy
import matplotlib.pyplot as plt

# This notebook uses "cartopy" for which the core package is the "Coordinate␣
↪Reference System" or "crs"
# Obtain cartopy with `conda install -c conda-forge cartopy`

import cartopy.crs

# Many cartopy examples use this alias
#import cartopy.crs as ccrs
```

Here we we illustrate how easy it is to use the cartopy package to solve the problem of visualizating spherical data. The documentation for cartopy is at https://scitools.org.uk/cartopy/docs/latest/ .

Let's look at the same model output as the previous noteboook, that resides on the native curvilinear grid of the model.

[2]:
```
nc = netCDF4.Dataset('http://esgf-data2.diasjp.net/thredds/dodsC/esg_dataroot/CMIP6/
↪CMIP/MIROC/MIROC6/1pctCO2/r1i1p1f1/Omon/tos/gn/v20181212/tos_Omon_MIROC6_1pctCO2_
↪r1i1p1f1_gn_330001-334912.nc')
```

We'll pre-load the data for efficiency. Again, this is only to avoid repeated transfers and usually we would use deferred reading.

[3]:
```
# For efficiency, load relevant data into memory
tos = nc.variables['tos'][0] # SST at a particular time
vlon = nc.variables['vertices_longitude'][:] # Vertex longitudes in inefficient (:,:,
↪:) form
vlat = nc.variables['vertices_latitude'][:] # Vertex latitudes in inefficient (:,:,:)␣
↪form
```

The following two notebook cells extract the contiguous vertex coordinates and store the mesh data in arrays `flon`, `flat`.

## 5.3 Extract geographic mesh coordinates

This is the exact same code we used to previously to re-arrange the vertex data into a more usable shape.

[4]:
```
# First create the latitude array for the mesh (which needs no value adjustments)
flat = numpy.zeros( (nc.variables['y'].shape[0]+1, nc.variables['x'].shape[0]+1) ) #␣
↪1 larger than cell shape
flat[1:,1:] = vlat[:,:,2] # Copy NE corner values to mesh
flat[0,1:] = vlat[0,:,1] # Copy SE corner for bottom row only
flat[1:,0] = vlat[:,0,3] # Copy NW corner for left column only
flat[0,0] = vlat[0,0,0] # Copy SW corner for SW cell only
```

[5]:
```
# Now create the longitude array for the mesh
flon = numpy.zeros( (nc.variables['y'].shape[0]+1, nc.variables['x'].shape[0]+1) )
flon[1:,1:] = vlon[:,:,2] # Copy NE corner values to mesh
flon[0,1:] = vlon[0,:,1] # Copy SE corner for bottom row only
flon[1:,0] = vlon[:,0,3] # Copy NW corner for left column only
flon[0,0] = vlon[0,0,0] # Copy SW corner for SW cell only
```

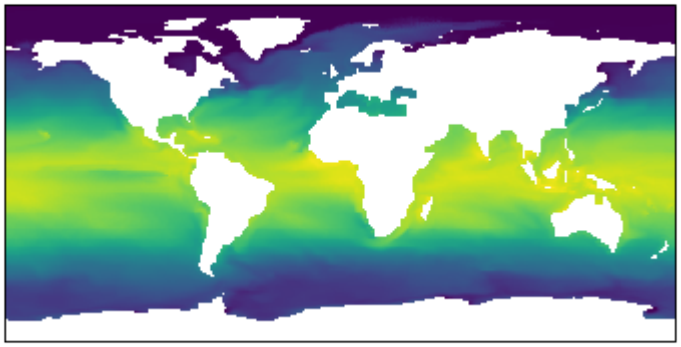
## 5.4 The Plate-Carree projection

To use cartopy, you declare the projection you wish to use in the visualization in the axes (i.e. here I use `plt.axes()` below). When you call pcolormesh for those axes (i.e. with `ax.pcolormesh()`) you specify the nature of the coordinates of the data with `transform=`. Most ESM output provides the coordinates as geographic latitude and longitude, or spherical coordinates. Coordinates are distinct from the nature of the grid or mesh. The mesh could be a convoluted

The "Platte Carree" projection is a simply a "latitude-longitude" plot: x and y in the plot correspond linearly to longitude and latitude. So we would normally use `transform=cartopy.crs.PlateCarree()`. When you ask for a "lat-lon" plot

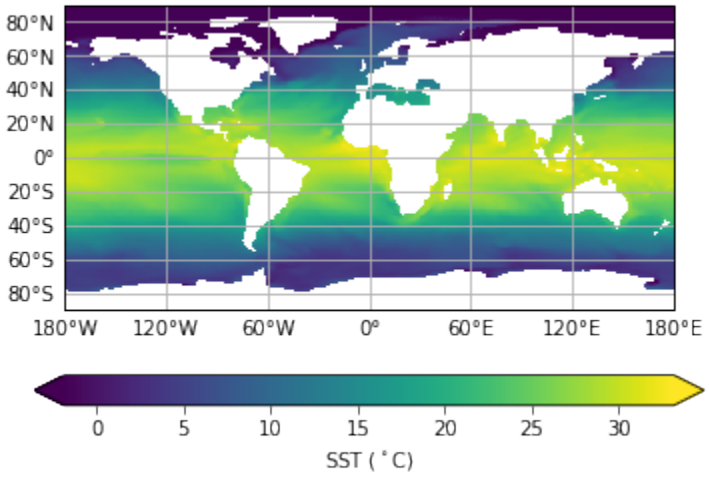
Now to make a rudimentary "lat-lon" plot we use `projection=cartopy.crs.PlateCarree()` as follows:

[6]:
```
# Create axes and register a "projection" for subsequent plot commands to use
ax = plt.axes(projection=cartopy.crs.PlateCarree());

# Plot the data
ax.pcolormesh( flon, flat, tos );
```



which looks artifact free! And to add more information to the plot ...

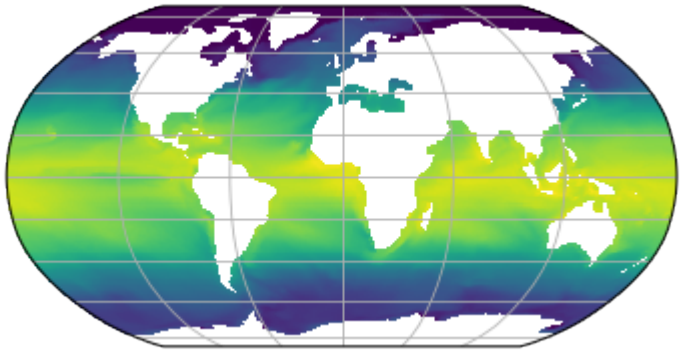
[7]:
```
# Create axes and register a "projection" for subsequent plot commands to use
ax = plt.axes(projection=cartopy.crs.PlateCarree());

pcm = ax.pcolormesh( flon, flat, tos );

# Add a color scale
plt.colorbar(pcm, ax=ax, orientation='horizontal', extend='both', label='SST ($^\circ$C)')

# Draw coordinate grid
gl = ax.gridlines(draw_labels=True);
# Control labelling
gl.xlabels_top = False
gl.ylabels_right = False
gl.xformatter = cartopy.mpl.gridliner.LONGITUDE_FORMATTER
gl.yformatter = cartopy.mpl.gridliner.LATITUDE_FORMATTER
```



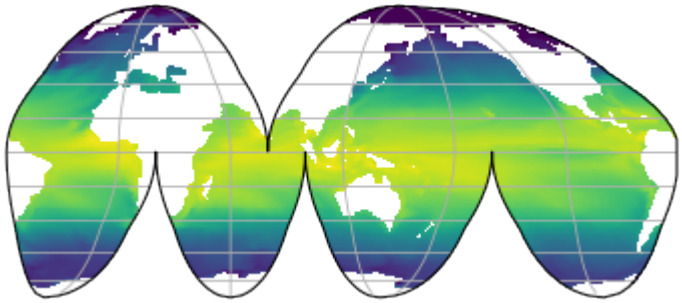

## 5.5 Other projections

The great thing about *cartopy* is it let's us use other projections very easily...

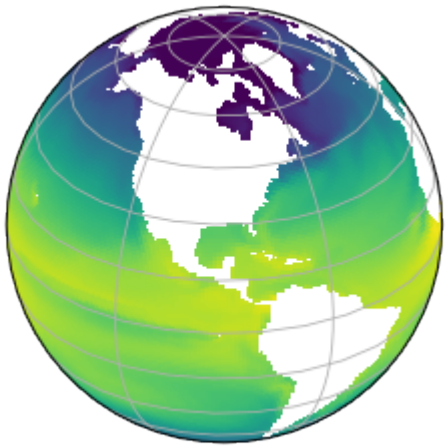
[8]:
```
# Create axes and register a "projection" for subsequent plot commands to use
ax = plt.axes(projection=cartopy.crs.Robinson());

# Plot the data
ax.pcolormesh( flon, flat, tos, transform=cartopy.crs.PlateCarree() );

# Draw coordinate grid
gl = ax.gridlines();
```



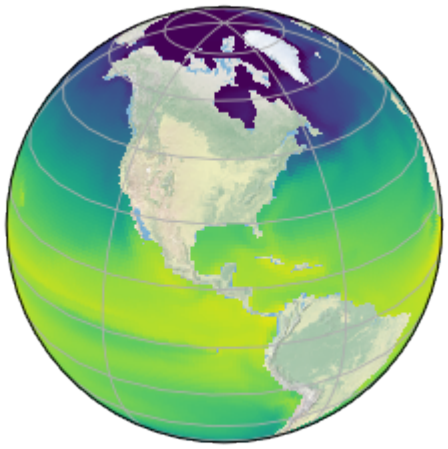
[9]:
```
# Create axes and register a "projection" for subsequent plot commands to use
ax = plt.axes(projection=cartopy.crs.InterruptedGoodeHomolosine(central_
↪longitude=120));

# Plot the data
ax.pcolormesh( flon, flat, tos, transform=cartopy.crs.PlateCarree() );

# Draw coordinate grid
gl = ax.gridlines();
```



[10]:
```
# Create axes and register a "projection" for subsequent plot commands to use
ax = plt.axes(projection=cartopy.crs.Orthographic(central_longitude=-90, central_
↪latitude=30));

# Plot the data
ax.pcolormesh( flon, flat, tos, transform=cartopy.crs.PlateCarree() );

# Draw coordinate grid
gl = ax.gridlines();
```



[11]:
```
# Create axes and register a "projection" for subsequent plot commands to use
ax = plt.axes(projection=cartopy.crs.NearsidePerspective(central_longitude=-90,␣
↪central_latitude=30));

# Plot the data
ax.pcolormesh( flon, flat, tos, transform=cartopy.crs.PlateCarree() );

# Stock image of planet will be seen through "missing data" holes (white)
ax.stock_img()

# Draw coordinate grid
gl = ax.gridlines();
```



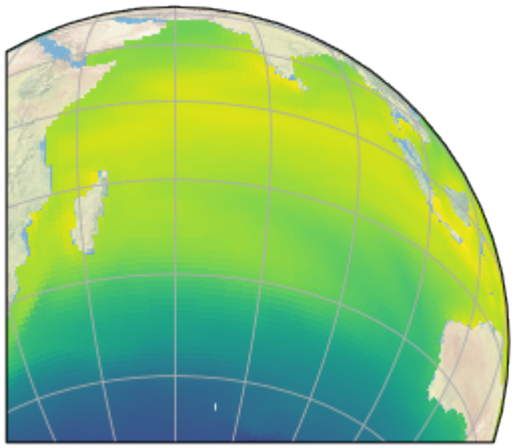
[12]:
```
# Create axes and register a "projection" for subsequent plot commands to use
ax = plt.axes(projection=cartopy.crs.NearsidePerspective(central_longitude=60,␣
↪central_latitude=-40));

# Plot the data
ax.pcolormesh( flon, flat, tos, transform=cartopy.crs.PlateCarree() );

# Stock image of planet will be seen through "missing data" holes (white)
ax.stock_img()
```

(continues on next page)

(continued from previous page)

```
# Draw coordinate grid
gl = ax.gridlines();

#ax.set_extent([-80,80,-20,80], crs=cartopy.crs.PlateCarree() )

RR = ax.get_xlim()[1]
ax.set_xlim((-0.5*RR,RR))
ax.set_ylim((-0.3*RR,RR))
```

[12]:
```
(-1642900.829389604, 5476336.097965347)
```



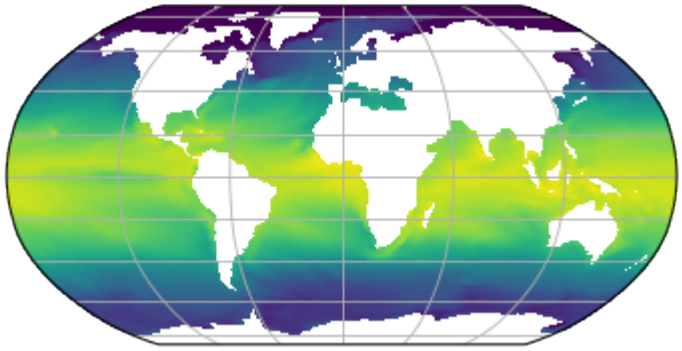
## 5.6 Avoiding re-organization of the mesh data

Big reveal! I started above by re-sampling the vertex mesh data into a usable shape. That is the right way to plot things. However, for most purposes the following shortcut works, *but only with cartopy*.

Even though previously we saw that using the cell-center coordinates did *not* work using pcolormesh, with cartopy projections the wrap around is handled properly so you can use pcolormesh with cell-center coordinates. Just remember it is skipping a row and column of data! You normally will not notice the difference. However, strictly speaking this approach is not showing you everything - for instance, if the extrema in the data were in the missing row/column then the automatic color scales would be affected.

[13]:
```
lon = nc.variables['longitude'][:] # Cell-center longitude coordinate of data
lat = nc.variables['latitude'][:] # Cell-center latitude coordinate of data
```

[14]:
```
# Create axes and register a "projection" for subsequent plot commands to use
ax = plt.axes(projection=cartopy.crs.Robinson());

# Plot the data
ax.pcolormesh( lon, lat, tos, transform=cartopy.crs.PlateCarree() );

# Draw coordinate grid
gl = ax.gridlines();
```



## 5.7 Summary

- cartopy allows us to look at data in spherical coordinates.
- Mesh coordinates are still needed but short-cutting with data-coordinates is often good enough.
- Numerous projections available.

# CHAPTER 6

# Reading and plotting with xarray

## 6.1 Synopsis

- Use xarray to read and plot spherical data in only a few lines

## 6.2 xarray

We now turn to the package **xarray** whose documentation you can find at http://xarray.pydata.org/en/stable/index.html.

Install with

```
conda install -c conda-forge xarray
```

although you should check their documentation for other dependencies such as cartopy (we've used the critical ones in earlier notebooks so if you've been following along you should be alright).

Here's the usual import cell:

[1]:
```
import xarray
import cartopy.crs
import matplotlib.pyplot as plt
```

In terms of what we've already covered, xarray plays the role of the netCDF4 and matplotlib.pyplot packages combined. It allows us to open a dataset and plot it with minimal intermediate steps.

First, we'll open the same dataset that we earlier accessed using `netCDF4.Dataset()` but this time with `xarray.open_dataset()`:

[2]:
```
ds = xarray.open_dataset('http://esgf-data2.diasjp.net/thredds/dodsC/esg_dataroot/
↪CMIP6/CMIP/MIROC/MIROC6/1pctCO2/r1i1p1f1/Omon/tos/gn/v20181212/tos_Omon_MIROC6_
↪1pctCO2_r1i1p1f1_gn_330001-334912.nc')
```

```
C:\Users\ajadc\Miniconda3\envs\py3\lib\site-packages\xarray\coding\times.py:122:␣
↪SerializationWarning: Unable to decode time axis into full numpy.datetime64 objects,
↪ continuing using dummy cftime.datetime objects instead, reason: dates out of range
  result = decode_cf_datetime(example_value, units, calendar)
C:\Users\ajadc\Miniconda3\envs\py3\lib\site-packages\xarray\coding\variables.py:69:␣
↪SerializationWarning: Unable to decode time axis into full numpy.datetime64 objects,
↪ continuing using dummy cftime.datetime objects instead, reason: dates out of range
  return self.func(self.array)
```

If you see pink-background warnings, don't worry - that's a result of the choice of dataset. The dataset was opened and we can query it by looking at the variable, `ds`, we created:

[3]:
```
print(ds)
```

```
<xarray.Dataset>
Dimensions:              (bnds: 2, time: 600, vertices: 4, x: 360, y: 256)
Coordinates:
  * time                 (time) object 3300-01-16 12:00:00 ... 3349-12-16 12:00:00
  * y                    (y) float64 -88.0 -85.75 -85.25 ... 148.6 150.5 152.4
  * x                    (x) float64 0.5 1.5 2.5 3.5 ... 356.5 357.5 358.5 359.5
    latitude             (y, x) float32 ...
    longitude            (y, x) float32 ...
Dimensions without coordinates: bnds, vertices
Data variables:
    time_bnds            (time, bnds) object ...
    y_bnds               (y, bnds) float64 ...
    x_bnds               (x, bnds) float64 ...
    vertices_latitude    (y, x, vertices) float32 ...
    vertices_longitude   (y, x, vertices) float32 ...
    tos                  (time, y, x) float32 ...
Attributes:
    _NCProperties:                  version=1|netcdflibversion=4.6.1|hdf5lib...
    Conventions:                    CF-1.7 CMIP-6.2
    activity_id:                    CMIP
    branch_method:                  standard
    branch_time_in_child:           0.0
    branch_time_in_parent:          0.0
    creation_date:                  2018-11-30T13:31:08Z
    data_specs_version:             01.00.28
    experiment:                     1 percent per year increase in CO2
    experiment_id:                  1pctCO2
    external_variables:             areacello
    forcing_index:                  1
    frequency:                      mon
    further_info_url:               https://furtherinfo.es-doc.org/CMIP6.MIR...
    grid:                           native ocean tripolar grid with 360x256 ...
    grid_label:                     gn
    history:                        2018-11-30T13:31:08Z ; CMOR rewrote data...
    initialization_index:           1
    institution:                    JAMSTEC (Japan Agency for Marine-Earth S...
    institution_id:                 MIROC
    mip_era:                        CMIP6
    nominal_resolution:             100 km
    parent_activity_id:             CMIP
    parent_experiment_id:           piControl
    parent_mip_era:                 CMIP6
    parent_source_id:               MIROC6
    parent_time_units:              days since 3200-1-1
```

(continues on next page)

(continued from previous page)

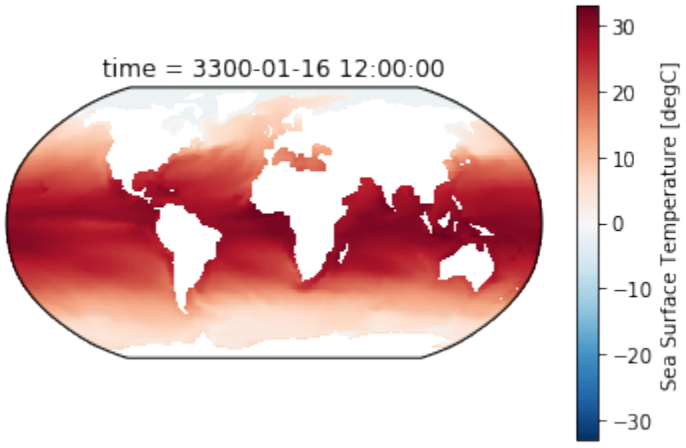
```
    parent_variant_label:           r1i1p1f1
    physics_index:                  1
    product:                        model-output
    realization_index:              1
    realm:                          ocean
    source:                         MIROC6 (2017): \naerosol: SPRINTARS6.0\n...
    source_id:                      MIROC6
    source_type:                    AOGCM AER
    sub_experiment:                 none
    sub_experiment_id:              none
    table_id:                       Omon
    table_info:                     Creation Date:(06 November 2018) MD5:072...
    title:                          MIROC6 output prepared for CMIP6
    variable_id:                    tos
    variant_label:                  r1i1p1f1
    license:                        CMIP6 model data produced by MIROC is li...
    cmor_version:                   3.3.2
    tracking_id:                    hdl:21.14100/636faabd-555e-414a-bbba-e25...
    DODS_EXTRA.Unlimited_Dimension: time
```

This is the same information we saw when we printed the netCDF4.Dataset object except that this time it is much easier to read and at the same time is more concise!

The coordinates section is of particular interest. Dimension variables (defined in *Basics of accessing data*) are indicated with an asterisk ("*") but there are some 2D variables in the coordinate section too.

We still have to use cartopy as before to handle spherical coordinates but to make a plot we need only two statements: 1. Specify the projection (otherwise how would python know how you want to look at the data?) 2. Specify the data to plot (`ds.tos[0]`)

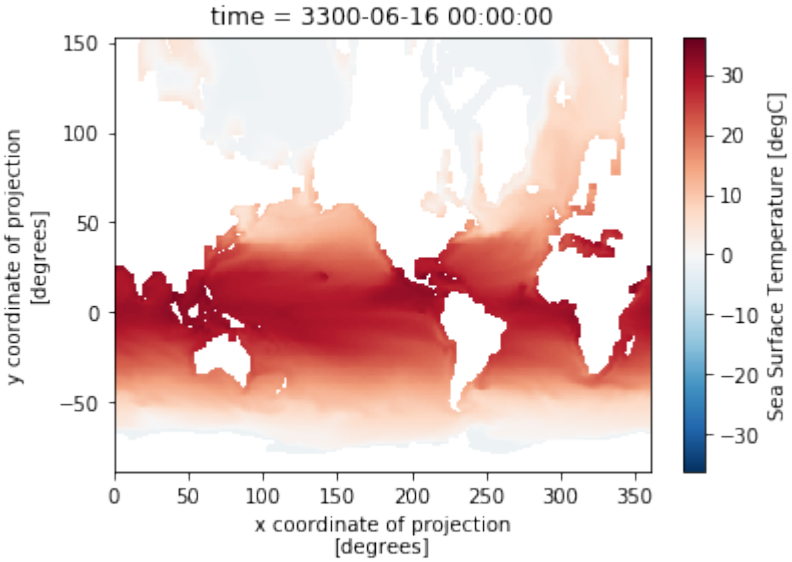
[4]:
```
ax = plt.axes(projection=cartopy.crs.Robinson());
ds.tos[0].plot(transform=cartopy.crs.PlateCarree(), x='longitude', y='latitude');
```




Notice how the time of the dataset was kindly added to the plot? This is an example of the convenience of a high-level package relative to the lower-level operations we've been using via netCDF4 and matplotlib.pyplot. xarray is designed for intuitive analysis of N-dimensional data.

To be complete, if you wish to skip the projection and plot using the projection coordinates, viewing the data with fully labeled axes and colorbar is as easy as this one line:

[5]:
```
ds.tos[5].plot();
```




## 6.3 Summary

- xarray is a high-level layer that sits on top of netCDF4 (or other APIs) and matplotlib.
- Used `.plot()` with cartopy to plot in just two lines.

# CHAPTER 7

# Reading and plotting with Iris

## 7.1 Synopsis

- Use Iris to read and plot spherical data in only a few lines

## 7.2 Iris

We now turn to the package **Iris** whose documentation you can find at https://scitools.org.uk/iris/docs/latest/.

Install with

```
conda install -c conda-forge iris
```

although you should check their documentation for other dependencies such as cartopy (we've used the critical ones in earlier notebooks so if you've been following along you should be alright).

Here's the usual import cell which imports both `iris` and some plotting modules provided within Iris:

[1]:
```
import iris
import iris.plot
import iris.quickplot
import cartopy.crs
import matplotlib.pyplot as plt
```

In terms of what we've already covered, Iris plays the role of the netCDF4 and matplotlib.pyplot packages combined, much like xarray. It allows us to open a dataset and plot it with minimal intermediate steps.

First, we'll open the same dataset that we accessed before using `netCDF4.Dataset()` and `xarray.open_dataset()` but this time with `iris.load_cube()`:

[2]:
```
tos = iris.load_cube('http://esgf-data2.diasjp.net/thredds/dodsC/esg_dataroot/CMIP6/
↪CMIP/MIROC/MIROC6/1pctCO2/r1i1p1f1/Omon/tos/gn/v20181212/tos_Omon_MIROC6_1pctCO2_
↪r1i1p1f1_gn_330001-334912.nc')
```

```
C:\Users\ajadc\Miniconda3\envs\py3\lib\site-packages\iris\fileformats\cf.py:798:␣
↪UserWarning: Missing CF-netCDF measure variable 'areacello', referenced by netCDF␣
↪variable 'tos'
  warnings.warn(message % (variable_name, nc_var_name))
```

If you see pink-background warnings, don't worry - that's because Iris sees a reference to a variable (*areacello*) that is in a different file/location. The dataset, or more precisely the "cube of data" was opened.

Note: we used `iris.load_cube()` but there is also an `iris.load()`. `iris.load()` loads a dataset, more like `xarray.open_dataset()` and would have returned a list with one element corresponding to the result of `iris.load_cube()`.

[3]:
```
tos
```

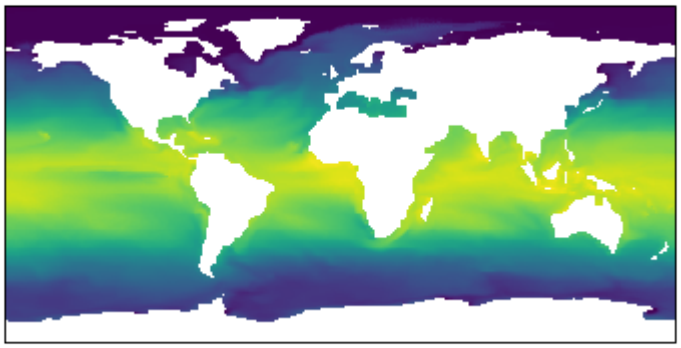
[3]:
```
<iris 'Cube' of sea_surface_temperature / (degC) (time: 600; -- : 256; -- : 360)>
```

This is the same information that we saw with netCDF4.Dataset or xarray.Dataset objects and is nicely displayed (if we had used `print(tos)` we would see an ascii form of the above html).

Notice that the coordinates are indicated as 2D and there is no mention of the 1D spatial dimension variables. This is great because it means Iris has properly interpreted the metadata following the CF convention and knows that the data resides an a non-trivial grid.

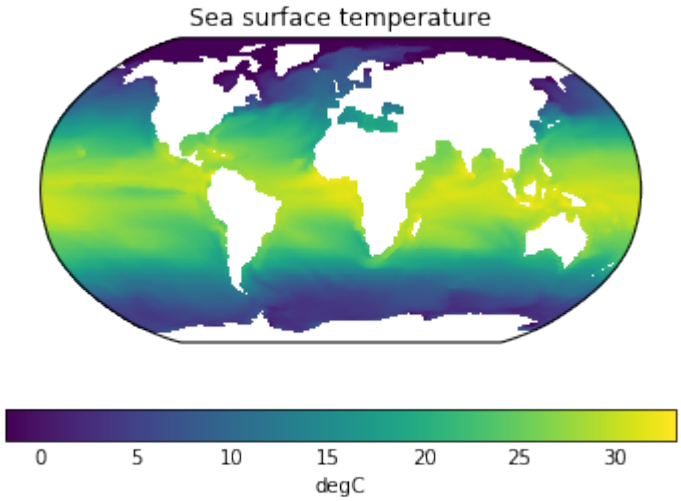
Do we still have to use cartopy (as with netCDF4 or xarray) to handle spherical coordinates?

[4]:
```
iris.plot.pcolormesh( tos[0] );
```



This has plotted things correctly and seems to be in a Plate-Carree (lat-lon) projection. Notice that this call to pcolormesh() did not add labels or a colorbar automatically. For that, Iris provide an alternative to their *plot* module called *quickplot*. Here's an examples also using a different cartopy projection.

[5]:
```
ax = plt.axes(projection=cartopy.crs.Robinson());
iris.quickplot.pcolormesh( tos[0] );
```

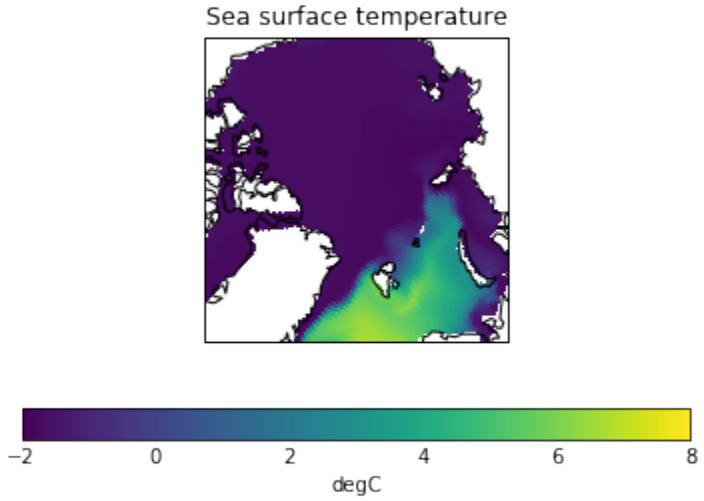



Unlike xarray, the time of the dataset was not added to the title but otherwise the plot is fairly complete and useful.

Notice that the `transform=` argument to `pcolormesh()` is not needed because Iris provides that information for you.

As a last test, let's see if Iris knows about the finite-volume nature of the data and can plot the Arctic properly without a seam...

[6]:
```
ax = plt.axes(projection=cartopy.crs.NorthPolarStereo());
iris.quickplot.pcolormesh( tos[0], vmin=-2, vmax=8 );
w=2e6; plt.xlim(-w,w); plt.ylim(-w,w); # Zoom into Arctic
plt.gca().coastlines();
```




## 7.3 Summary

- Iris is a high-level layer that understands the CF convention in netCDF files.
- Iris can plot ocean-model data correctly out-of-the-box!

# CHAPTER 8

# Deeper dive into pcolormesh()

## 8.1 Synopsis

- An examination of how pcolormesh expects cell boundaries, and what happens when the cell boundaries are *not* used.

[1]:
```
import matplotlib.pyplot as plt
import xarray
import numpy
import cartopy
```

## 8.2 Some test data

We'll make up some simple data to examine how pcolormesh works.

- The mesh will be in longitude-latitude coordinates.
- The mesh will be irregularly spaced along each coordinate direction.
- There will be 6x4 data values in 6x4 cells.
- vlon, vlat will be the 1D coordinates of cell boundaries.
- Think "V" for vertex.
- clon, clat will be the 1D coordinates of the cell centers, where the notionally data resides.
- Think "C" for cell center.

[2]:
```
# Irregularly space vertex longitudes
vlon = numpy.cumsum([0,60,60,60,45,45,90])-180
# Irregularly space vertex latitudes
vlat = numpy.cumsum([0,45,45,30,60])-90
# Cell centers defined using finite volume interpretation, i.e. middle of cell based␣
↪on cell bounds
```

(continues on next page)

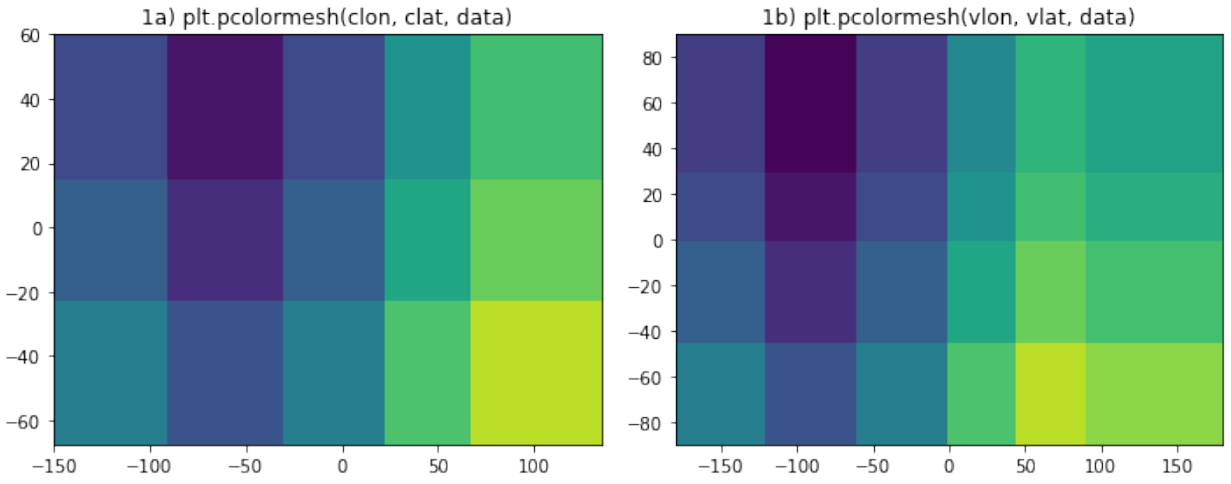
(continued from previous page)

```
clon = (vlon[:-1]+vlon[1:])/2 # Longitudes of column centers
clat = (vlat[:-1]+vlat[1:])/2 # Latitudes of row centers
# Some arbitrary data to plot
CLON2d, CLAT2d = numpy.meshgrid(clon, clat)
data = (1+numpy.sin(numpy.pi*CLON2d/180)+(0.5-CLAT2d/180)**2)/3
del CLAT2d, CLON2d
```

## 8.3 Simple pcolormesh via matplotlib.pyplot

First will use the pcolormesh directly from matplotlib.pyplot.

[3]:
```
plt.figure(figsize=(10,4))

plt.subplot(121)
plt.pcolormesh(clon, clat, data, vmin=0, vmax=1);
plt.title('1a) plt.pcolormesh(clon, clat, data)');

plt.subplot(122)
plt.pcolormesh(vlon, vlat, data, vmin=0, vmax=1);
plt.title('1b) plt.pcolormesh(vlon, vlat, data)');

plt.tight_layout()
```




There should be 6x4 cells of data showing.

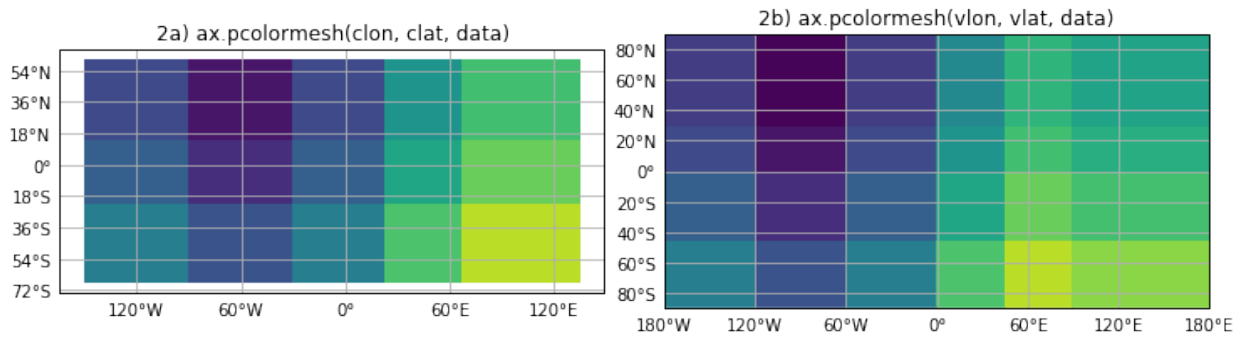
- Panel 1a shows only 5x3 values. Moreover, the locations of the cells are wrong.
- The domain spans from -180<x<180 and -90<y<90 but the figure shows significantly less of the domain.
- The interface between the 3rd and 4th columns should be at x=0, but is off to the right. Similarly the boundary between the 2nd and 3rd rows should be at y=0 but it to far north. All cell boundaries are wrong.
- Panel 1b correctly shows 6x4 cells with correctly placed boundaries.

## 8.4 pcolormesh via cartopy

Now let's use cartopy to make the plot.

To use cartopy, you declare the projection you wish to use in the visualization in the axes (i.e. where I use `plt.subplot()` below). When you call pcolormesh for those axes (i.e. with `ax.pcolormesh()`) you normally specify the nature of the coordinates for the data with `transform=`. Since most ESMs provide geographic coordinates then you tend to use `transform=cartopy.crs.PlateCarree()`.

[4]:
```
plt.figure(figsize=(10,4))

ax = plt.subplot(121, projection=cartopy.crs.PlateCarree())
ax.pcolormesh(clon, clat, data, transform=cartopy.crs.PlateCarree(), vmin=0, vmax=1);
gl = ax.gridlines(draw_labels=True); gl.xlabels_top,gl.ylabels_right = False,False
gl.xformatter,gl.yformatter = cartopy.mpl.gridliner.LONGITUDE_FORMATTER, cartopy.mpl.
↪gridliner.LATITUDE_FORMATTER
plt.title('2a) ax.pcolormesh(clon, clat, data)');

ax = plt.subplot(122, projection=cartopy.crs.PlateCarree())
ax.pcolormesh(vlon, vlat, data, transform=cartopy.crs.PlateCarree(), vmin=0, vmax=1);
gl = ax.gridlines(draw_labels=True); gl.xlabels_top,gl.ylabels_right = False,False
gl.xformatter,gl.yformatter = cartopy.mpl.gridliner.LONGITUDE_FORMATTER, cartopy.mpl.
↪gridliner.LATITUDE_FORMATTER
plt.title('2b) ax.pcolormesh(vlon, vlat, data)');

plt.tight_layout()
```




Apart from some handy decoartion of the plot (labels), the behavior for `ax.pcolormesh()` is no different than for `plt.pcolormesh()`:

- Panel 2a shows the wrong number of cells, with the wrong cell bondaries, and does not fill the domain.
- Panel 2b is drawn correctly.

## 8.5 pcolormesh via xarray

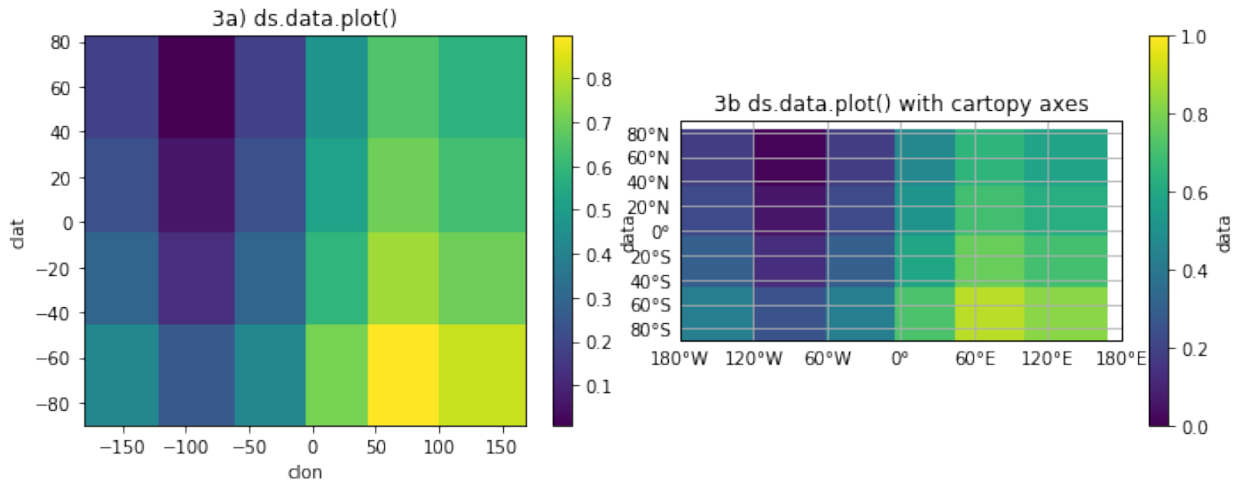
Now to examine how pcolormesh behaves for an xarray `DataSet` (or `DataArray`) we need to define a Dataset. Here, I'm indicating that vlon and vlat could also be coordinates.

[5]:
```
ds = xarray.Dataset({'data': (['clat', 'clon'], data)},
                    coords={
                        'clon': (['clon'], clon),
                        'clat': (['clat'], clat),
                        'vlon': (['vlon'], vlon),
                        'vlat': (['vlat'], vlat),
                    }
                   )
ds
```

[5]:
```
<xarray.Dataset>
Dimensions:  (clat: 4, clon: 6, vlat: 5, vlon: 7)
Coordinates:
  * clon     (clon) float64 -150.0 -90.0 -30.0 22.5 67.5 135.0
  * clat     (clat) float64 -67.5 -22.5 15.0 60.0
  * vlon     (vlon) int32 -180 -120 -60 0 45 90 180
  * vlat     (vlat) int32 -90 -45 0 30 90
Data variables:
    data     (clat, clon) float64 0.4219 0.2552 0.4219 ... 0.4702 0.6506 0.5783
```

Here's see what happens when we use the `.plot()` method and the `.pcolormesh()` method on a cartopy created axes:

[6]:
```
plt.figure(figsize=(10,4))

plt.subplot(121)
ds.data.plot();
plt.title('3a) ds.data.plot()')

ax = plt.subplot(122, projection=cartopy.crs.PlateCarree())
ds.data.plot(transform=cartopy.crs.PlateCarree(), vmin=0, vmax=1);
gl = ax.gridlines(draw_labels=True); gl.xlabels_top,gl.ylabels_right = False,False
gl.xformatter,gl.yformatter = cartopy.mpl.gridliner.LONGITUDE_FORMATTER, cartopy.mpl.
↪gridliner.LATITUDE_FORMATTER
plt.title('3b ds.data.plot() with cartopy axes')

plt.tight_layout()
```



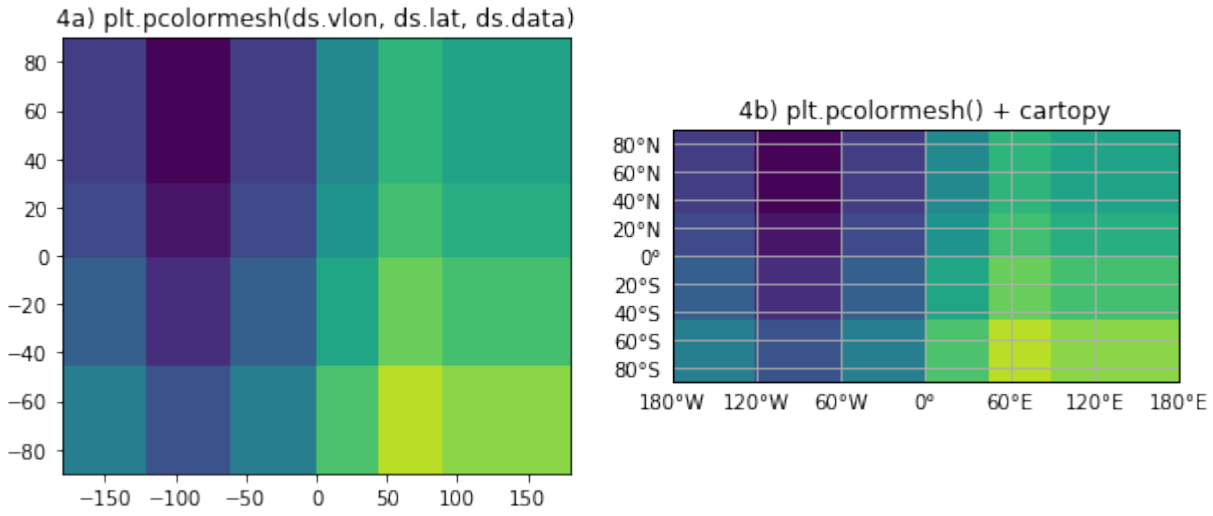

Good and bad news!

- On the up side, there are 6x4 cells shown in both panels (3a and 3b).
- On the down side, the cell boundaries are not in the correct locations.
- Also, more of the domain is shown than for 1a or 2a but the domain is still not completely filled.

What has happened here is that the good folks behind xarray, knowing that pcolormesh expects cell boundary locations, have calculated the cell boundaries from the cell-center coordinates. It's better than ignoring the problem but unfortunately only gives the correct result for a uniformly spaced mesh.

At this time, I'm unaware of a clean fix for this using xarray so calling `plt.pcolormesh()` with explicit reference to the the cell boundaries, rather than using the help `.plot()` function, seems like the only way to get a correct plot.

[7]:
```
# It would be nice if something like this worked...
# ds.data.plot.pcolormesh(x='vlon',y='vlat',infer_intervals=False)

plt.figure(figsize=(10,4))

plt.subplot(121)
plt.pcolormesh(ds.vlon, ds.vlat, ds.data, vmin=0, vmax=1);
plt.title('4a) plt.pcolormesh(ds.vlon, ds.lat, ds.data)');

ax = plt.subplot(122, projection=cartopy.crs.PlateCarree())
plt.pcolormesh(ds.vlon, ds.vlat, ds.data, vmin=0, vmax=1, transform=cartopy.crs.
↪PlateCarree());
gl = ax.gridlines(draw_labels=True); gl.xlabels_top,gl.ylabels_right = False,False
gl.xformatter,gl.yformatter = cartopy.mpl.gridliner.LONGITUDE_FORMATTER, cartopy.mpl.
↪gridliner.LATITUDE_FORMATTER
plt.title('4b) plt.pcolormesh() + cartopy');
```




## 8.6 Summary

- We've seen that plotting with pcolormesh only gives the wrong right results when the cell boundary coordinates are provided.
- When data coordinate are used (i.e. not cell boundaries):
- not all data is plotted;
- location of data is wrong.
- The xarray helper function, `.plot()`, plots all the data but not necessarily in the right location
- xarray `.plot()` is still better than raw pcolormesh.

# CHAPTER 9

# Coordinates, Projections, and Grids

## 9.1 Synopisis

- Review of coordinate systems and projections of the sphere onto the 2d placne
- Discussion about lengths and areas in finite volume grids used by ESMs

[1]:
```
%run -i figure-scripts/init.py
```

## 9.2 Coordinate systems

A coordinate system allows us to (uniquely) specify points in space. You should be familiar with the Cartesian 2d coordinate system where, by convention, (x,y) are the normal distances, measured in the same units, from two perpendicular lines through the origin.

We live in a 3d world (referring to spatial dimensions) and so require 3 numerical values to label a point in space. In Cartesian coordinates they might be (x,y,z) referenced to the center of the Earth, with z being height above the equatorial plane (positive in the direction of the North pole), y being the distance from a plane through the poles and a reference meridian, and x the distance from the plane perpendicular to both other planes. The equations of motion used by many models are often derived starting in this Cartesian coordinate system. However, these Cartesian coordinates are inconvenient to use in practice because we live on the surface of a sphere and "up", as defined by gravity, is sometimes increasing z (at the North Pole), and sometimes changing x or y (at the Equator).

### 9.2.1 Spherical coordinates

In ESMs we typically use spherical coordinates, $\lambda$, $\phi$ and $r$, where $\lambda$ is "longitude", a rotation angle eastward around the poles starting at a reference meridian; $\phi$ is "latitude", an elevation angle from the Equatorial plane (positive in Northern hemisphere), and $r$ is the radial distance from the center of the Earth. $\lambda, \phi, r$ are related to Cartesian $x, y, z$

by some simple relations:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} r\cos\phi\cos\lambda \\ r\cos\phi\sin\lambda \\ r\sin\phi \end{pmatrix}$$

Note that $r, x, y, z$ are all in the same units (eg. kilometets or meters) and $\lambda, \phi$ are angles usually given in degrees or radians.

**Coordinate singularities:** At the North and South poles of the coordinate system, $\phi = \pm\pi/2 = \pm 90°$, all values of longitude refer to the same point. There is no "east" when you are positioned at the pole. This has many consequences, but one of the more fundamental is that spherical coordinates are not a good coordinate to use to design a discretization of the spherical domain.

**Periodic coordinates:** While a tuple of longitude, latitude and radius unambiguously define a point in space, given a point in space you there are multiple valid longitudes that refer to the same point. Longitude is cyclic ($\pm 360°$ is equivalent to $0°$). This can cause problems in practice, particularly for plotting spherical data for which effort is sometimes needed to handle the periodicity.

### 9.2.2 Geographic coordinates

We live on the surface of the Earth and to precisely refer to points near the Earth's surface requires a properly defined geographic coordinate system. A common choice of coordinates is latitude, longitude and altitude, where altitude is height above a particular surface. Unfortunately the Earth is not spherical and that reference surface is better approximated as an ellipsoidal.

In order to be unambiguous about the definition of coordinates, map-makers choose a reference ellipse with a agreed upon scale and orientation. They then choose the most appropriate mapping of the spherical coordinate system onto that ellipsoid, called a *geodetic datum*. A widely used global datum includes the World Geodetic System (WGS 84), the default datum used for the Global Positioning System. When you are given a latitude-longitude pair of values, strictly speaking without the geodetic datum, the is some ambiguity about the actual physical point being referred to. For ESMs, the datum is rarely provided and this is because ESMs almost universally approximate the Earth as a sphere and use spherical coordinates for referencing locations. This means some approximation is required when comparing real-world positions and model positions.

The latitude and longitude using these horizontal datums are the spherical coordinates of the point on an ellipse. It you draw a straight line from the point on the ellipsoidal to the center, if passes through all spheres co-centered with the same latitude and longitude.

Different datum have different reference points and scales, and so longitude and latitude can differ between geodetic datum.

## 9.3 Projections

To view data covering the surface of a sphere, or the Earth, we have to project that 3d surface into 2d. Imagine peeling the rind off an orange in one piece and then trying to flatten it onto a table top; the curvature in the peel requires you to distort the rind or make cuts, in order to flatten it fully. This is the function of the map projections and distortion is inevitable. Some projection preserve properties such as relative angles between lines, or relative area, but there is no projection of the surface of the sphere that can avoid distortion of some form.

A projection maps the longitude and latitude of spherical coordinates into a new coordinate system. Very confusingly, sometimes the projection coordinates will be called longitude and latitude too! The projection coordinates are meaningless unless you know what the projection is so you often find a reference to the projection in the meta-data of coordinates; it means the longitude and latitude are not spherical coordinate but projection coordinates.

[2]:
```
%run -i figure-scripts/some-projections.py
```

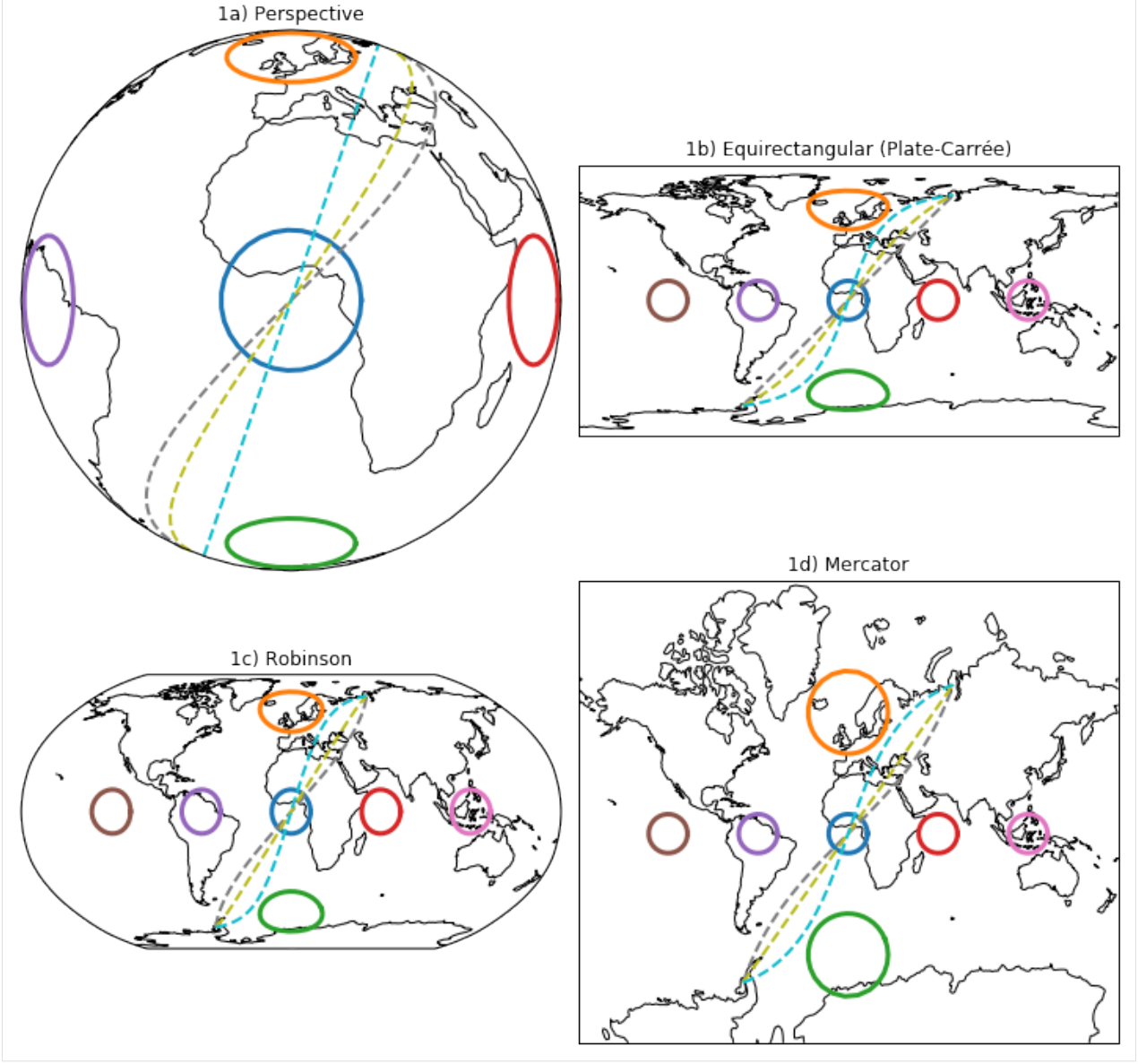



Figure 1: The colored circles in these plots are circles in the tangent plane to the sphere and projected onto the surface. The various projections can distort the circles. The circles are separated zonally or meridionally by 60°. In 1a, a perspective image of the sphere, the circles appear non-circular because of the viewing angle. The blue circle appears circular because we are viewing it from directly overhead. The projection in 1b is the easy to use Plate-Carrée projection, a "lat-lon" plot, in which circles are stretched zonally with distance from the equator. 1d shows the Mercator projection in which circles remain circles but are expanded in size away from th equator. 1c shows the Robinson projection which compromises between the two. The purple dashed lines is a straight line in latitude-longitude coordinates, and the yellow dashed line is a straight line in the Mercator coordinates. The cyan dashed line is a great arc, and is straight in the perspective view because we are viewing it from directly overhead.

The two most useful projections are the equirectangular and Mercator projections.

### 9.3.1 Equirectangular projection

This is the simplest projection, sometimes thought of a non-projection which is incorrect. In general it takes the form

$$x = R\,(\lambda - \lambda_0)\cos\phi_0 \tag{9.1}$$
$$y = R\,(\phi - \phi_0) \tag{9.2}$$

The origin of the plot, $(x, y) = (0, 0)$ corresponds to $(\lambda, \phi) = (\lambda_0, \phi_0)$. The $\cos\phi_0$ term is a constant and the most common choice of $\phi_0 = 0$ gives the plate carrée projection, which means "flat square" in French. In this case, the projection is simply

$$x = R\,(\lambda - \lambda_0) \tag{9.3}$$
$$y = R\phi \tag{9.4}$$

Distances in the y-direction are proportional to the meridional direction on the sphere, but the x-direction are stretched zonally, more so further from the equator. This is apparent in the orange and green circles in figure 1b, where the heights or the loops are the same as circles on the equator but the width is markedly increased.

In the cartopy package, this projection is called "Plate-Carrée" which is French for flat square. Other names for this projection are equidistant cylindrical projection and geographic projection. See https://en.wikipedia.org/wiki/Equirectangular_projection.

### 9.3.2 Mercator projection

The Mercator projection has the same stretching in the x-direction as the equirectangular projection but, in order to preserve shape, it also stretches the y direction so that infinitesimal elements are stretched isotropically (the y-stretching is equal to the x-stretching).

$$x = R\,(\lambda - \lambda_0) \tag{9.5}$$
$$y = R\tanh^{-1}(\sin\phi) \tag{9.6}$$

At the polar singularities, the x-stretching is infinite so y becomes infinite and the Mercator projection can never show the poles. See https://en.wikipedia.org/wiki/Mercator_projection.

### 9.3.3 Lines

A length of a line between two points is a function of the path taken. On the surface of sphere, the shortest path between two given points is a great arc. A great arc does not appear straight in many projections. Unfortunately, many grid calculations use a great arc for the length of a line between nodes on a model grid, which can be inconsistent with the constraints or assumptions about the grid.

The dashed curves in figure 1 are "straight" lines between two points in various projections. The cyan dashed curve is a great arc. The purple dashed curve is a straight line in the Plate-Carree projection (latitude-longitude space) and the yellow dashed curve is a straight line in the Mercator projection. All are curved in most other projections. To describe a straight line in some projection then *the projection must be known*, irrespective of the coordinate system defining the end points. That is, we can define the end points of the line in latitude-longitude coordinates but say a line is straight in the Mercator projection, and by so doing unambiguously define that line.

In the Mercator projection, the length of a line is $\frac{R}{\cos\alpha}\Delta\phi$ where $\tan\alpha = \frac{\Delta y}{\Delta x}$

## 9.4 ESM grids

Many ESMs use quadrilateral grids to discretize the surface of the sphere. The following discussion also applies to fully unstructured grids built from polygons but here we use quadrilateral grids for simplicity. There are also grids that

have cuts and joins but here we'll stick to space-filling grids that are logically rectangular, meaning they can be stored in rectangular arrays in computer memory and referenced with a pair of indices ($i, j$ by convention).

A quadrilateral grid is a rectangular mesh of adjacent quadrilateral cells that share edges and vertexes. Although the mesh and the cell are logically rectangular they might be physically curvilinear. From the grid we require positions of nodes, distances along edges, and areas of cells.

If we choose a coordinate system with which to record the locations of mesh nodes, say spherical latitude-longitude with appropriate definitions, then we can unambiguously define those node locations. We could describe the exact same grid using a different coordinate system, say 3D Cartesian coordinates. The physical positions of the nodes of the grids are part of what define the grid, but the choice of coordinates with which we describe those positions does not change the grid.

The edges of each cell are a curve between two adjacent nodes but the particular path of the curve has to be defined. Different paths will have different lengths. Similarly, the particular paths of the cell edges will determine the cell area. Thus the path of the cell edges is a fundamental component of a model grid needed for calculating the lengths and areas on a grid.

### 9.4.1 Simple spherical coordinate grid

Before we discuss the best choice for defining a curve between points, let's briefly define a simple spherical-coordinate grid. The mesh is formed of lines of constant longitude and lines of constant latitude.

Let $i \in 0, 1, \ldots, n_i$ and $j \in 0, 1, \ldots, n_j$, then node $i, j$ is at longitude $\lambda_i$ and latitude $\phi_j$ where $\lambda_i = \lambda_0 + i\Delta\lambda$, $\phi_j = \phi_0 + j\Delta\phi$.

Here, $\Delta\lambda$ and $\Delta\phi$ are grid spacings. In practice, these can be smooth functions of $i$ and $j$ respectively but here we treat them as constant.

An example simple spherical grid is shown below. The red dots are the nodes of the mesh with positions $\lambda_i, \phi_j$. The dashed lines are the cell edges that for a regular net. Notice that in the Plate-Carrée projection the grid is regular because the grid-spacing in constant in longitude-latitude coordinates.

The lengths and areas of the grid are measured on the surface of sphere. We defined the edges to be either lines of constant longitude or latitude. Using spherical geometry, the length of a meridionally-oriented (constant longitude) cell edge is $R\Delta\phi$. For a zonally-oriented edge at constant latitude $\phi_j$, the length is $R\Delta\lambda\cos\phi_j$. The area of a cell labelled $i + \frac{1}{2}, j + \frac{1}{2}$ bounded by four edges is $R^2\Delta\lambda\left(\sin\phi_{j+1} - \sin\phi_j\right)$.

The metric factors for this grid are the same as for a Plate-Carrée projection because we are defining the paths of the cell edges to be straight in the Plate-Carrée projection. The use of the Plate-Carrée coordinates for position, namely longitude and latitude, is a happy coincidence which means everything, positions and metrics, are defined by one projection.

[3]:
```
%run -i figure-scripts/simple-spherical-grid.py
```

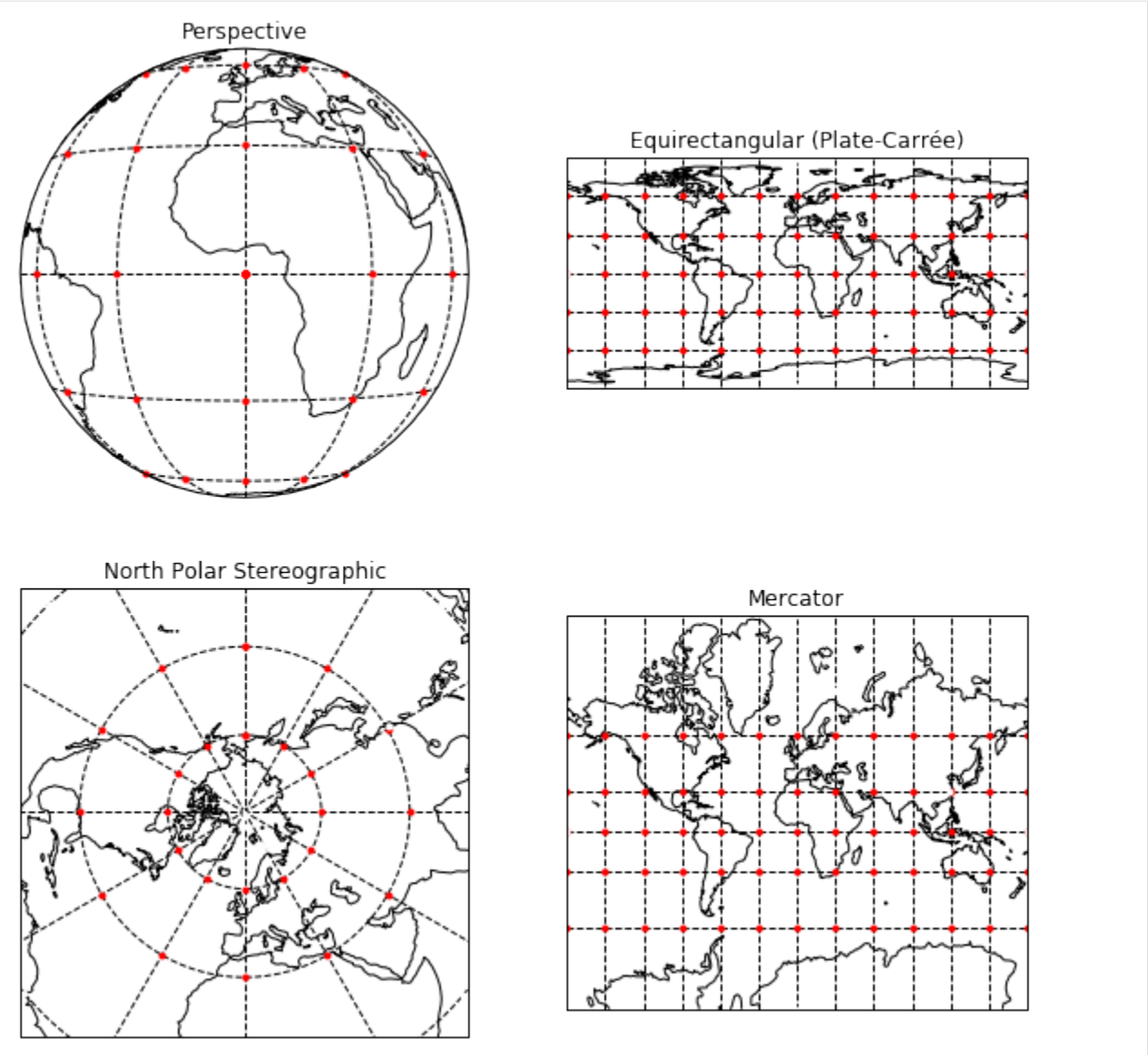



Figure 2: A simple spherical grid view in different projections. The red dots are the nodes of the mesh. The dashed lines are the grid of cell edges. Here the grid spacing is constant in latitude-longitude coordinates which is why the grid looks regular in the Plate-Carrée projection.

In the above example of a simple spherical grid we used spherical geometry combined with the specification of the paths of cell edges to calculate everything. If we were given just the positions of nodes a spherical grid, we could re-calculate the grid lengths and areas given the knowledge that the cell edges are straight in the Plate-Carrée projection.

### 9.4.2 Not to use great arcs

If is quite common for the projection for the cell edges to be omitted from a grid file. Sometimes, you will encounter some software that was written to handle arbitrary grids that does not know about the projections. In this instance, developers often make a choice for how to calculate a length between two points. There are two immediately simple choices we can make: i) the shortest curve, or ii) choose a projection in which to draw a straight line. There are other ways to choose and define the curve forming edges but they are not as useful for out purposes.

As discussed above, the shortest line is a great arc but it turns out that choosing great arcs does not result in orthogonal

meshes without a particular distributions of nodes. For example, if you consider the example spherical grid defined above, using a great arc to approximate the distance along a latitude circle will underestimate the distance. If you were to then calculate the actual paths implied by the great arcs, the grid is not orthogonal at the nodes.

If no projection is given, then straight lines in the Plate-Carrée projection make a reasonable approximation that at least works for Plate-Carrée and Mercator. Many grids a composed of patches of grids that used different projections but have to match at the joins and it seems the Plate-Carrée is often the common denominator.

## 9.5 Contents of grid files

As mentioned earlier, because most ESM treat the Earth as spherical, they rarely bother to choose a datum for the geographic coordinate system. Further, the longitudes and latitudes *may not be spherical coordinates* but instead may be the result of a projection.

The habit of not specifying the projection for cell edges is somewhat alleviated by the general pattern of providing the numerical values for lengths and areas as part of the grid specification. For example, see https://www.researchgate.net/publication/228641121_Gridspec_A_standard_for_the_description_of_grids_used_in_Earth_System_models. For many calculations, including integrals and gradients of scalars, having the numerical values of lengths and areas is sufficient. For vector operations, the angle of grid-lines is needed.

The CF convention requires the grid to be provided in all files. That convention supports specification of mappings (projections) but does not distinguish the projection for paths from the projections for coordinates. by default, longitude and latitude are the true geographic coordinates. And the required grid information is limited to the node positions. Further the nodes correspond to the data locations which is a finite-difference perspective of a mesh, and quite different from the finite volume perspective that most ESMs assume.

## 9.6 Summary

CHAPTER 10

# Operations on a grid

## 10.1 Synopsis

## 10.2 Summary

CHAPTER 11

# Indices and tables

- genindex
- modindex
- search