Basemap tutorial Documentation

Release 0.1

Roger Veciana i Rovira

Jan 26, 2018

Contents

Basemap is a great tool for creating maps using python in a simple way. It's a matplotlib extension, so it has got all its features to create data visualizations, and adds the geographical projections and some datasets to be able to plot coast lines, countries, and so on directly from the library.

Basemap has got some documentation, but some things are a bit more difficult to find. I started this documentation to extend a little the original documentation and examples, but it grew a little, and now covers many of the basemap possibilities.

I started writing Basemap examples for a talk given during the 2014 SIGTE meetings and I posted some examples in my blog (geoexamples.com). I would really appreciate some feedback if somebody finds an error, or thinks that some section should be added. You can leave a comment at the blog post announcing this tutorial.

CHAPTER 1

Getting started

1.1 Installation

The objective of this tutorial was to show how to use Basemap, since the installation part is very well documented at the official docs.

In my case, I used Basemap in several Linux configurations.

- Using Ubuntu I used the Synaptic to install the library and its dependencies.
- Using an older Suse I used the source code

There is a blog post explaining all the options

1.1.1 Downloading the tutorial

All the files, including the documentation and the sample files are at GitHub: https://github.com/rveciana/BasemapTutorial

If you have GIT installed, you can clone the project by typing

git clone https://github.com/rveciana/BasemapTutorial.git

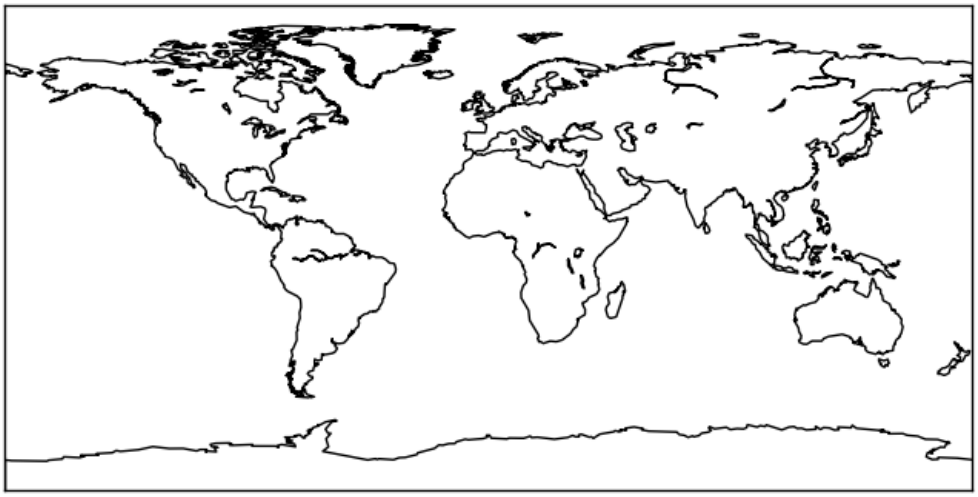
1.2 Drawing the first map

Let's create a the simplest map:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap()

map.drawcoastlines()
```

```
plt.show()
plt.savefig('test.png')
```

1. The first two lines include the Basemap library and matplotlib. Both are necessary
2. The map is created using the Basemap class, which has many options. Without passing any option, the map has the Plate Carrée projection centered at longitude and latitude = 0
3. After setting the map, we can draw what we want. In this case, the coast lines layer, which comes already with the library, using the method *drawcoastlines()*
4. Finally, the map has to be shown or saved. The methods from mathplotlib are used. In this example, *plt.show()* opens a window to explore the result. *plt.savefig('file_name')* would save the map into an image.

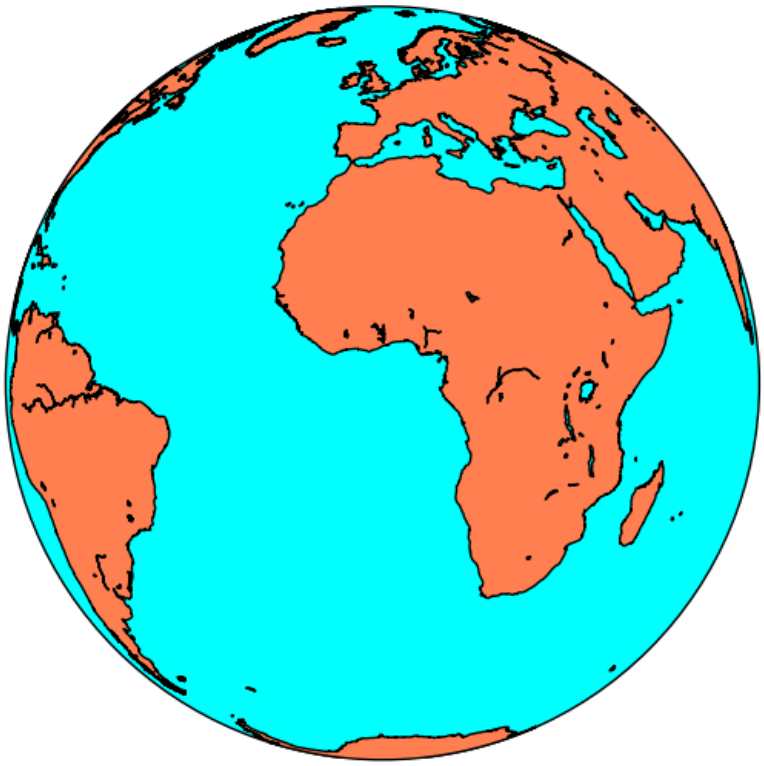


Changing the projection is easy, just add the *projection* argument and *lat_0* and *lon_0* to the *Basemap* contructor.

Even with the new projection, the map is still a bit poor, so let's fill the oceans and continents with some colors The methods *fillcontinents()* and *drawmapboundary()* will do it:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='ortho',
              lat_0=0, lon_0=0)
```

```
#Fill the globe with a blue color
map.drawmapboundary(fill_color='aqua')
#Fill the continents with the land color
map.fillcontinents(color='coral',lake_color='aqua')

map.drawcoastlines()

plt.show()
```



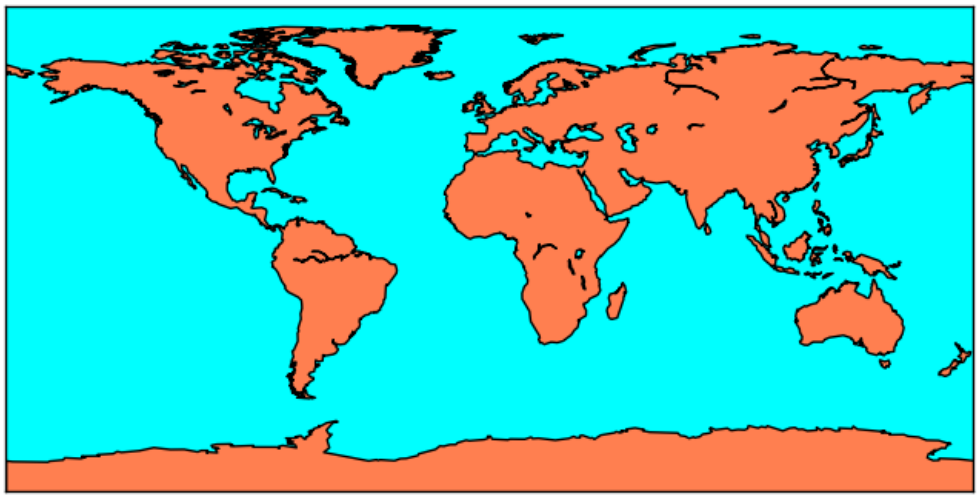
1.3 Managing projections

All maps must have a projection. The projection and its features are all assigned when the object *Basemap* is created. The way to do it is quite different from other libraries (i.e. GDAL), so understanding this point is very important for working with Basemap.

1.3.1 Projection

The projection argument sets the map projection to be used:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='cyl')

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()

plt.show()
```

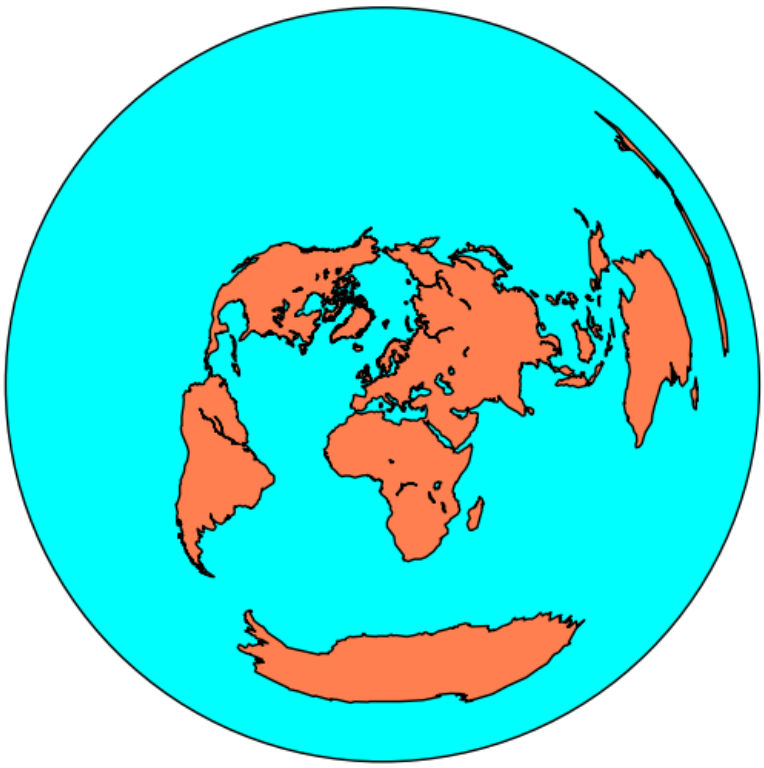


The default value is *cyl*, or Cylindrical Equidistant projection, also known as *Equirectangular projection* or *Plate Carrée*

Many projections require extra arguments:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='aeqd', lon_0 = 10, lat_0 = 50)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()
```

```
plt.show()
```



The map has now an equidistant projection centered at longitude = 10 and latitude = 50, which is over Europe. Some projections require more parameters, described in each projection page at the manual.

The Basemap object has the field *proj4string* that has the string to be used with proj4 for calculating the projection parameters without using Basemap.

Using epsg to set the projection

The EPSG code is a standard to name projections using a numerical code. Basemap allows to create the maps using this notation, but only in certain cases. To use it, pass the *epsg* argument to the *Basemap* constructor with the code.

The epsg codes supported by Basemap are at the file *<python_packages_path>/mpl_toolkits/basemap/data/epsg*. Even if the desired epsg appears in the file, sometimes the library can't use the projection, raising the error

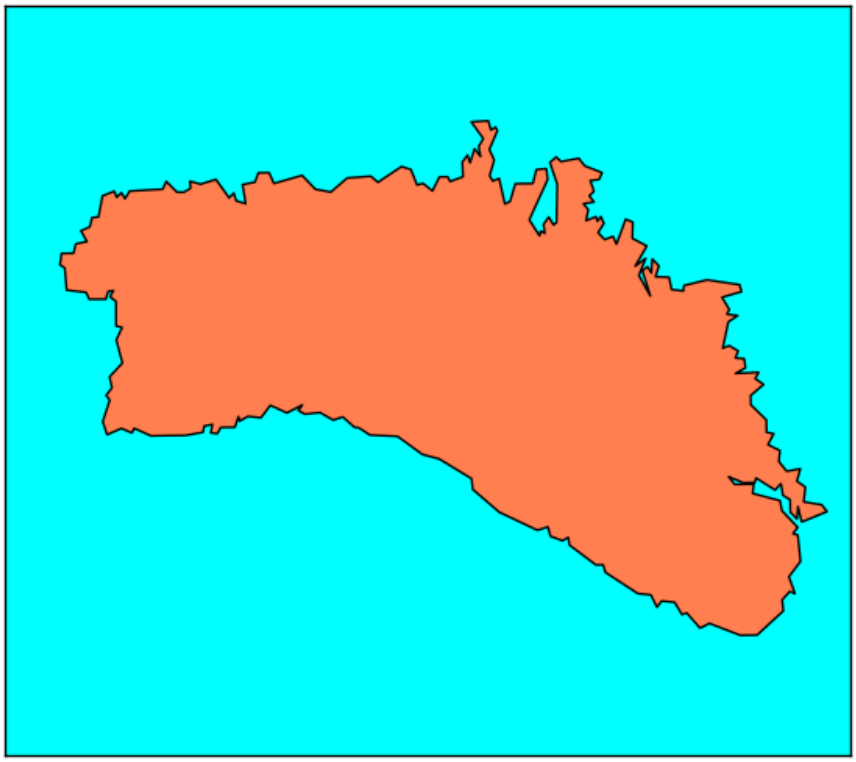
> ValueError: 23031 is not a supported EPSG code

Projections with the name "utm" are not well supported (i.e. 23031 or 15831), but the ones named tmerc can be used. The way I found to do it was opening the file and looking for a suitable option.

This example shows the island of Menorca using the UTM projection, zone 31N.

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
```

```
map = Basemap(llcrnrlon=3.75,llcrnrlat=39.75,urcrnrlon=4.35,urcrnrlat=40.15,␣
↪resolution = 'h', epsg=5520)

map.drawmapboundary(fill_color='aqua')
#Fill the continents with the land color
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()
plt.show()
```



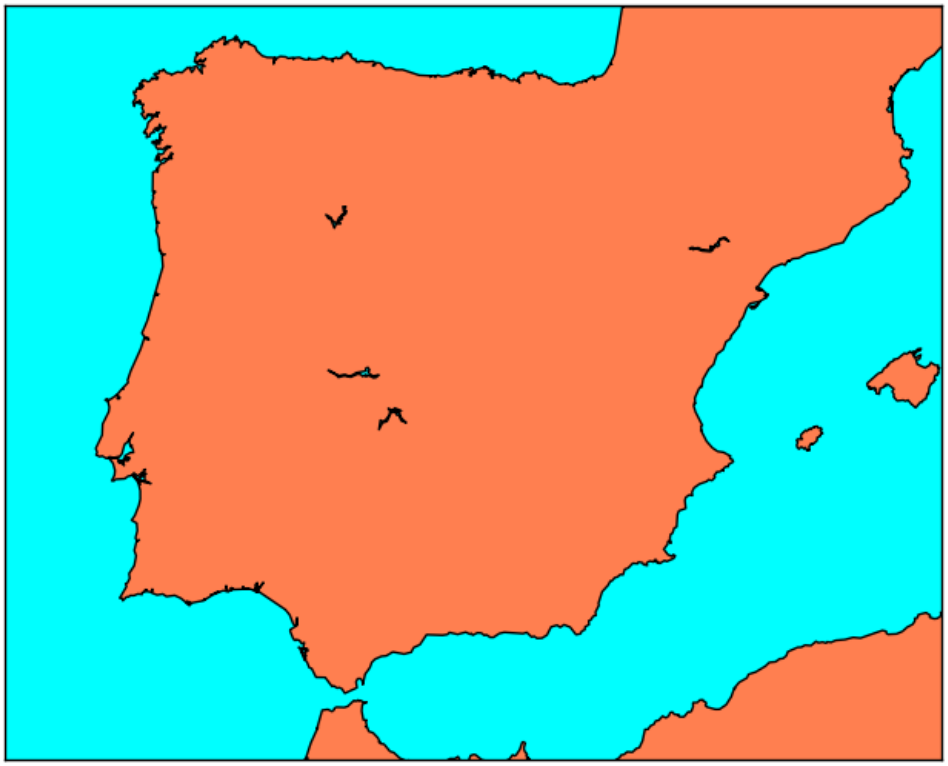
1.3.2 Extension

All the examples until now take the whole globe. Drawing only a region can be done either passing a bounding box or the center of the map and the map size. The official docs say that both methods can be used most of the times, but there are many exceptions.

Note: Using cyl, merc, mill, cea and gall projections, the corners are assumed to be -180, -90, 180, 90 (all the globe) by default if they are not set. The other projection need the extension to be set by one of the three methods.

Passing the bounding box

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(llcrnrlon=-10.5,llcrnrlat=35,urcrnrlon=4.,urcrnrlat=44.,
              resolution='i', projection='tmerc', lat_0 = 39.5, lon_0 = -3.25)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()

plt.show()
```



The lower-left and upper-right corners are past as parameters, in longitude-latitude units, not in map units. This is the reason why in some projections fails, since a square in longitude-latitude may not give a good bounding box in the projected units.

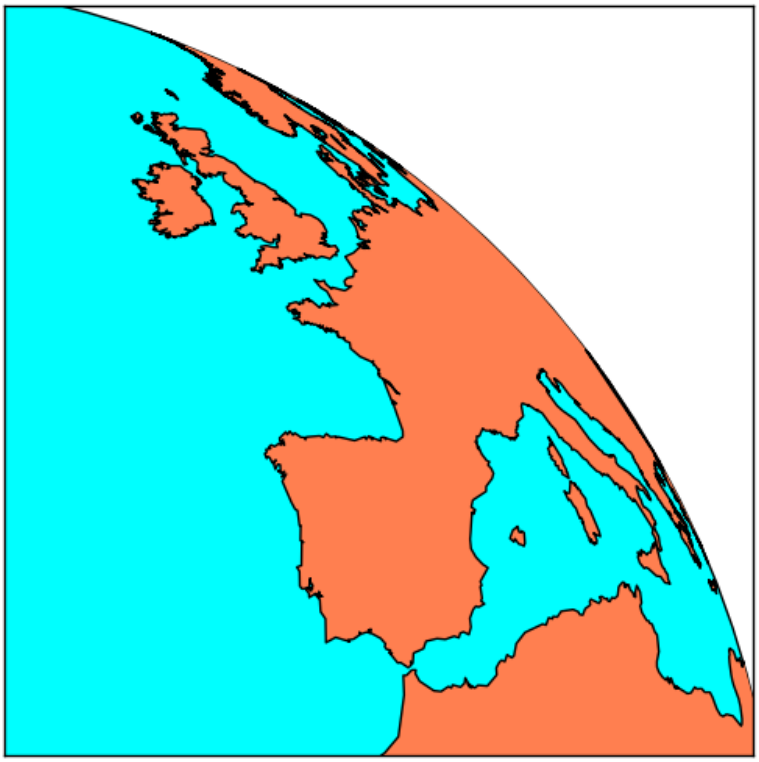
In the example, the UTM (Transverse Mercator) projection is used. The bounding box method is easier in this case, since calculating the width in UTM units from the center of the map is much more difficult.

Note: Using sinu, moll, hammer, npstere, spstere, nplaea, splaea, npaeqd, spaeqd, robin, eck4, kav7, or mbtfpq projections, this method can't be used. either because all the globe is plotted or because the extension can't be

calculated from the geographical coordinates.

Passing the bounding box in the map coordinates

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(resolution='l',
              satellite_height=3000000.,
              projection='nsper',
              lat_0 = 30., lon_0 = -27.,
              llcrnrx=500000.,llcrnry=500000.,urcrnrx=2700000.,urcrnry=2700000.
             )

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()

plt.show()
```

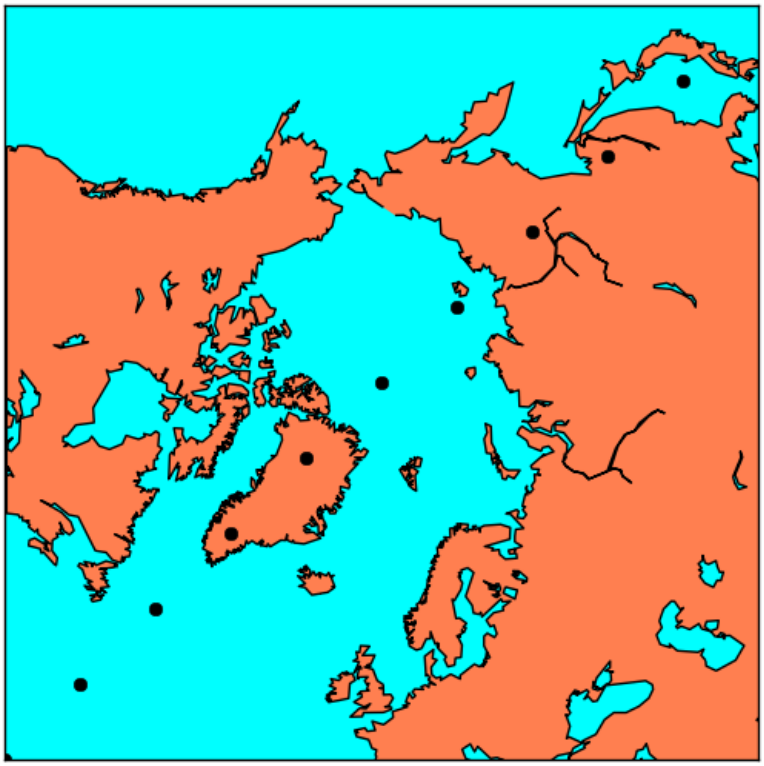


Some projections (the ones that look *like a satellite image*) accept setting the extension using the map coordinates. The projection parameters must be set (center point), and then the zone to show can be only a part of the globe.

Note: Only ortho, geos and nsper projections can use this method to set the map extension

Passing the center, width and height

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='aeqd',
              lon_0 = 0,
              lat_0 = 90,
              width = 10000000,
              height = 10000000)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()

for i in range(0, 10000000, 1000000):
    map.plot(i, i, marker='o',color='k')

plt.show()
```



In this case, the center of the projection, the width and height of the projection are passed as parameters.

The center is easy, just pass it in longitude-latitude. The size is a bit more tricky:

The units are the projection units in meters. The point *(0, 0)* is the lower left corner, and the point *(width, height)* is the upper right. So the origin of the positions is *not* the one defined by the projection as in GDAL. The projection just defines the size of the units used, not the origin.

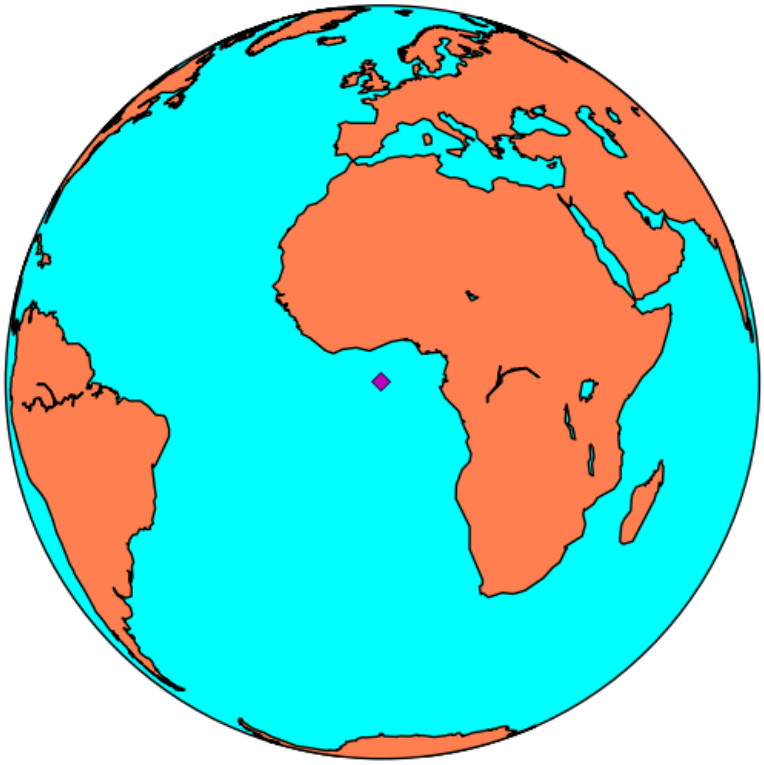
The example shows the position of several points using the *plot* function to show how the coordinates range from 0 to width and height.

1.4 Basic functions

1.4.1 Drawing a point in a map

Drawing a point in a map is usually done using the plot method:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='ortho',
              lat_0=0, lon_0=0)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()

x, y = map(0, 0)

map.plot(x, y, marker='D',color='m')

plt.show()
```

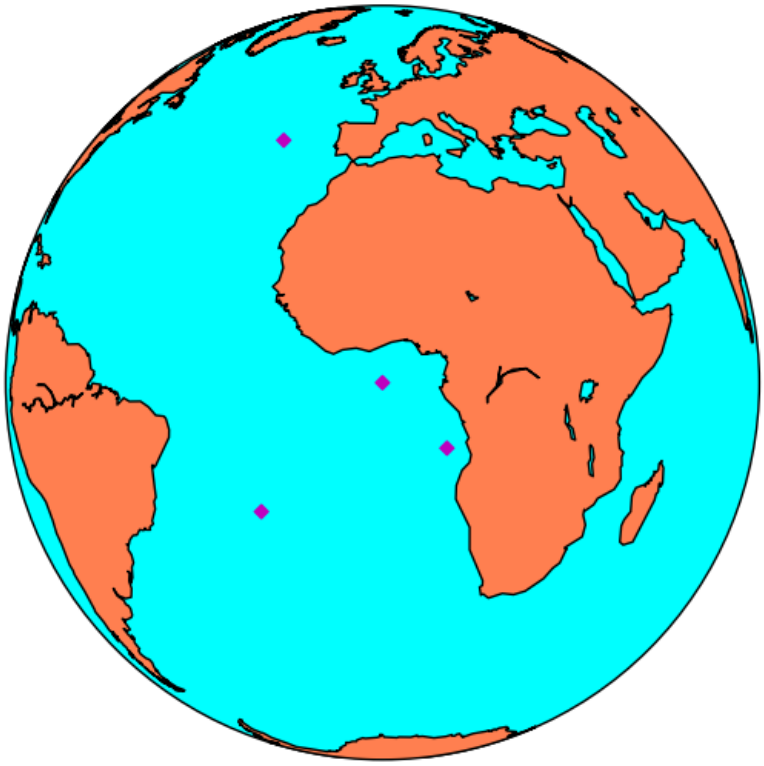


- **Use the Basemap instance to calculate the position of the point in the map coordinates when you have the longitude and la**
  - If latlon keyword is set to True, x,y are interpreted as longitude and latitude in degrees. Won't work in old *basemap* versions
- **The *plot* method needs the x and y position in the map coordinates, the marker and the color**
  - By default, the marker is a point. This page explains all the options
  - By default, the color is black (k). This page explains all the color options

If you have more than one point, you may prefer the scatter method. Passing an array of point into the plot method creates a line connecting them, which may be interesting, but is not a point cloud:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='ortho',
              lat_0=0, lon_0=0)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()

lons = [0, 10, -20, -20]
lats = [0, -10, 40, -20]
```

```
x, y = map(lons, lats)

map.scatter(x, y, marker='D',color='m')

plt.show()
```



- Remember that calling the *Basemap* instance can be done with lists, so the coordinate transformation is done at once
- The format options in the *scatter* method are the same as in plot

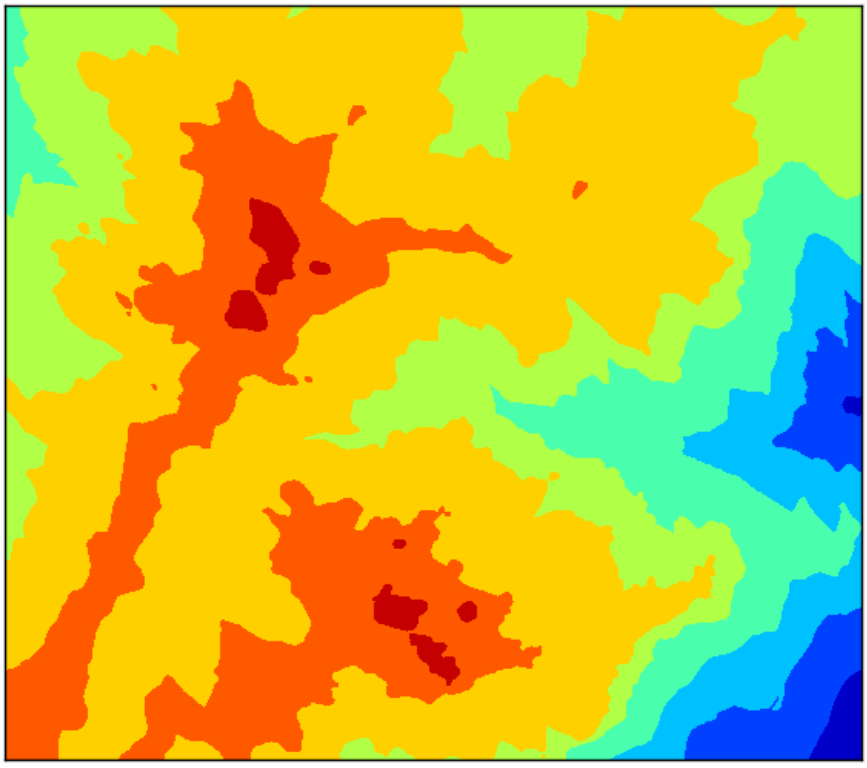
1.4.2 Plotting raster data

There are two main methods for plotting a raster, *contour/contourf*, that plots contour lines or filled contour lines (isobands) and *pcolor/pcolormesh*, that creates a pseudo-color plot.

contourf

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from osgeo import gdal
```

```
from numpy import linspace
from numpy import meshgrid

map = Basemap(projection='tmerc',
              lat_0=0, lon_0=3,
              llcrnrlon=1.819757266426611,
              llcrnrlat=41.583851612359275,
              urcrnrlon=1.841589961763497,
              urcrnrlat=41.598674173123)

ds = gdal.Open("../sample_files/dem.tiff")
data = ds.ReadAsArray()

x = linspace(0, map.urcrnrx, data.shape[1])
y = linspace(0, map.urcrnry, data.shape[0])

xx, yy = meshgrid(x, y)

map.contourf(xx, yy, data)

plt.show()
```

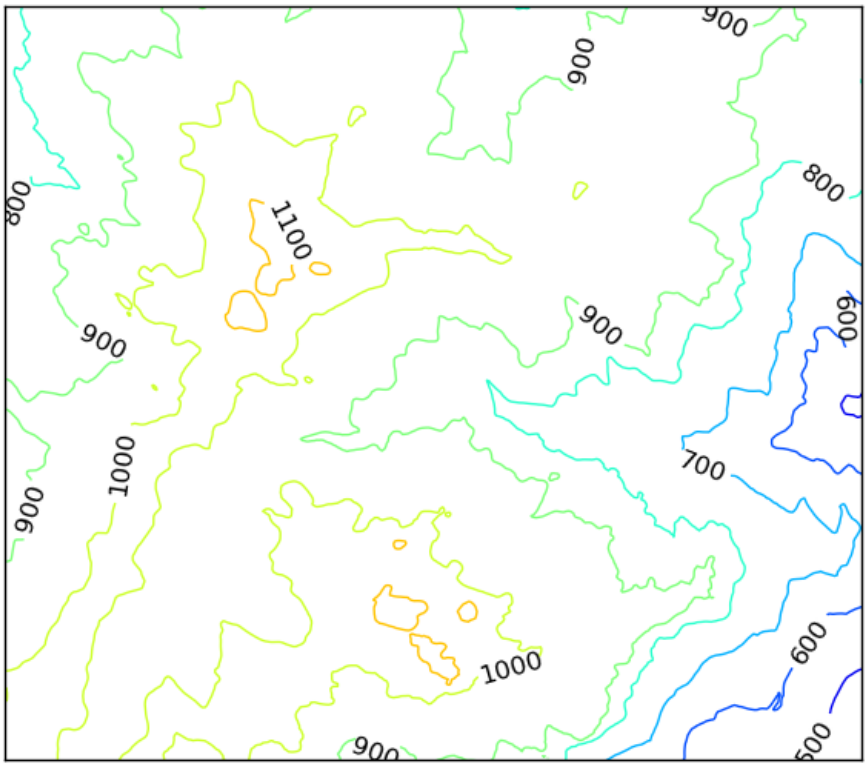


- The map is created with the same extension of the raster file, to make things easier.

- **Before plotting the contour, two matrices have to be created, containing the positions of the x and y coordinates for each p**
  - linspace is a numpy function that creates an array from an initial value to an end value with n elements. In this case, the map coordinates go from 0 to *map.urcrnrx* or *map.urcrnry*, and have the same size that the data array *data.shape[1]* and *data.shape[0]*
  - meshgrid is a numpy function that take two arrays and create a matrix with them. This is what we need, since the *x* coordinates repeat in every column, and the *y* in every line
- The contourf method will take the *x*, *y* and *data* matrices and plot them in the default *colormap*, called jet, and an automatic number of levels
- **The number of levels can be defined after the data array, as you can see at the section *contour*. This can be done in two wa**
  - An integer indicating the number of levels. The extreme values of the data array will indicate the extremes of the color scale
  - A list with the values for each level. The range function is useful to set them i.e. range(0, 3000, 100) to set a level each 100 units

contour

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from osgeo import gdal
from numpy import linspace
from numpy import meshgrid

map = Basemap(projection='tmerc',
              lat_0=0, lon_0=3,
              llcrnrlon=1.819757266426611,
              llcrnrlat=41.583851612359275,
              urcrnrlon=1.841589961763497,
              urcrnrlat=41.598674173123)

ds = gdal.Open("../sample_files/dem.tiff")
data = ds.ReadAsArray()

x = linspace(0, map.urcrnrx, data.shape[1])
y = linspace(0, map.urcrnry, data.shape[0])

xx, yy = meshgrid(x, y)

cs = map.contour(xx, yy, data, range(400, 1500, 100), cmap = plt.cm.cubehelix)
plt.clabel(cs, inline=True, fmt='%1.0f', fontsize=12, colors='k')

plt.show()
```

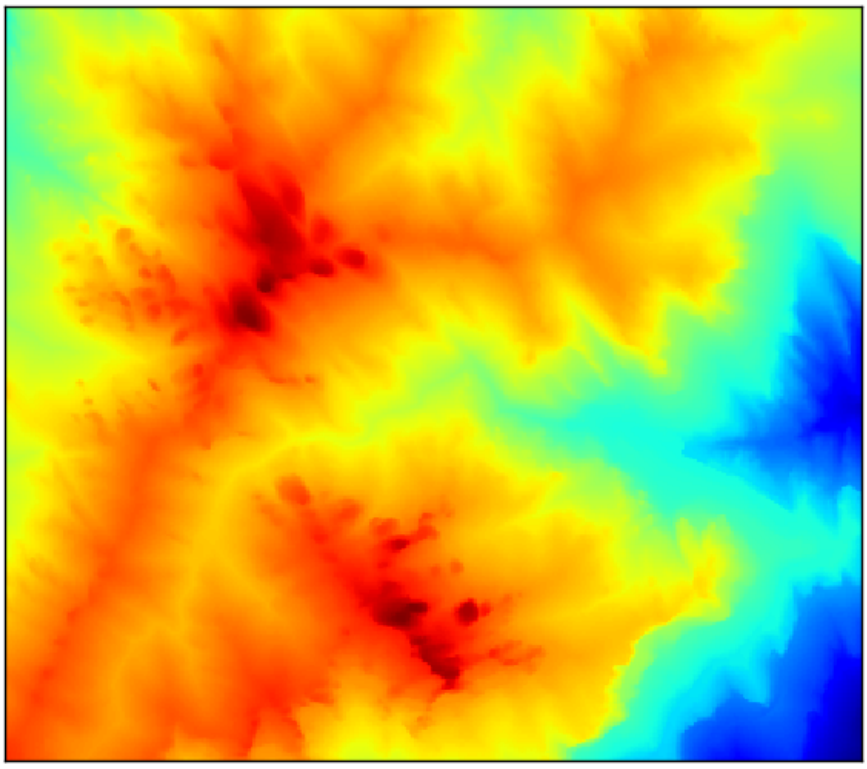


- The data must be prepared as in the *contourf* case
- The levels are set using the range function. We are dealing with altitude, so from 400 m to 1400 m , a contour line is created each 100 m
- The colormap is not the default jet. This is done by passing to the cmap argument the cubehelix colormap
- **The labels can be set to the contour method (but not to contourf)**
  - inline makes the contour line under the line to be removed
  - fmt formats the number
  - fontsize sets the size of the label font
  - colors sets the color of the label. By default, is the same as the contour line

pcolormesh

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from osgeo import gdal
from numpy import linspace
from numpy import meshgrid

map = Basemap(projection='tmerc',
```

```
              lat_0=0, lon_0=3,
              llcrnrlon=1.819757266426611,
              llcrnrlat=41.583851612359275,
              urcrnrlon=1.841589961763497,
              urcrnrlat=41.598674173123)

ds = gdal.Open("../sample_files/dem.tiff")
data = ds.ReadAsArray()

x = linspace(0, map.urcrnrx, data.shape[1])
y = linspace(0, map.urcrnry, data.shape[0])

xx, yy = meshgrid(x, y)

map.pcolormesh(xx, yy, data)

plt.show()
```



- The data must be prepared as in the *contourf* case
- The colormap can be changed as in the *contour* example

Note: pcolor and pcolormesh are very similar. You can see a good explanation here

1.4.3 Calculating the position of a point on the map

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='aeqd', lon_0 = 10, lat_0 = 50)

print map(10, 50)
print map(20015077.3712, 20015077.3712, inverse=True)
```

The output will be:

(20015077.3712, 20015077.3712) (10.000000000000002, 50.000000000000014)

When inverse is False, the input is a point in longitude and latitude, and the output is the point in the map coordinates. When inverse is True, the behavior is the opposite.

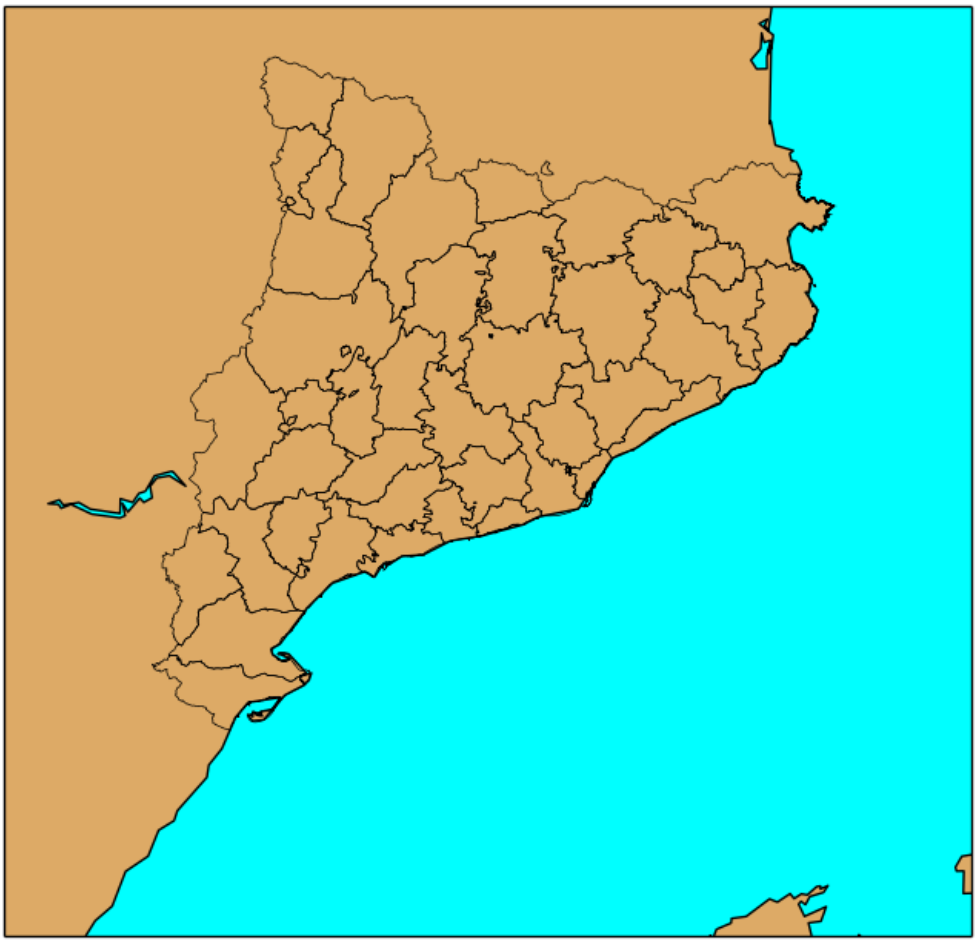
1.5 Working with shapefiles

The way used by Basemap to handle vectorial files is quite different from other libraries, and deserves some attention.

1.5.1 Basic usage

Let's start with the easiest way to plot a shapefile:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(llcrnrlon=-0.5,llcrnrlat=39.8,urcrnrlon=4.,urcrnrlat=43.,
              resolution='i', projection='tmerc', lat_0 = 39.5, lon_0 = 1)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='#ddaa66',lake_color='aqua')
map.drawcoastlines()

map.readshapefile('../sample_files/comarques', 'comarques')

plt.show()
```



- The first parameter shapefile name must go without the shp extension. The library assumes that all shp, sbf and shx files will exist with this given name
- The second parameter is a name to access later to the shapefile information from the Basemap instance, as we will show later

There are some restrictions:

- The file must be in EPSG:4326, or lat/lon coordinates. If your file isn't, you can use ogr2ogr to transform it
- The elements must have only 2 dimensions
- This example will show something only if the elements are polygons or polylines

As the image shows, the result will be only the boundary of the polygons (or the polylines). To fill them, look at the

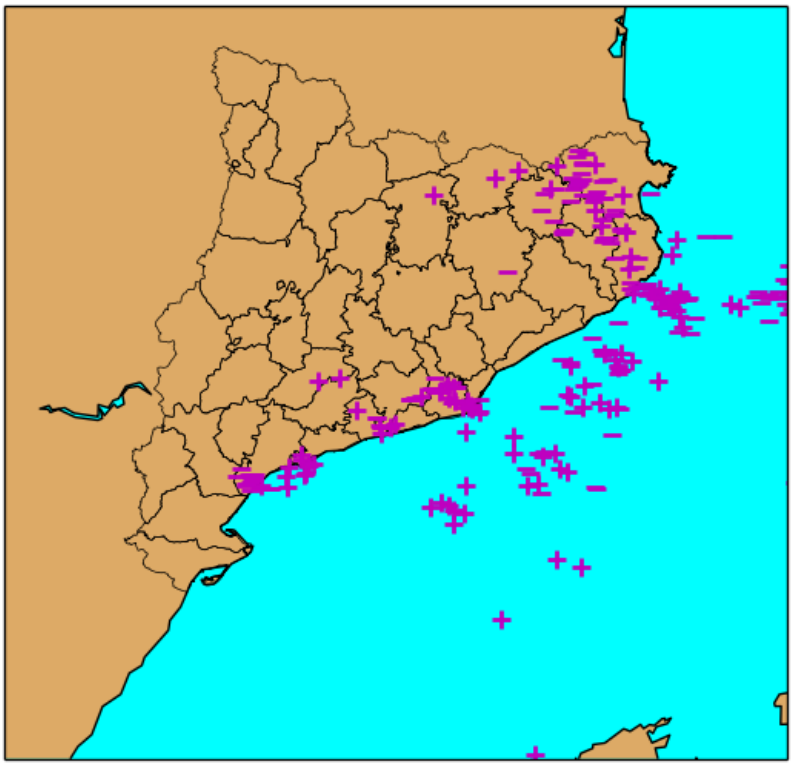
last section *Filling polygons*

1.5.2 Reading point data

Plotting points is a bit more complicated. First, the shapefile is read, and then the points can be plotted using scatter, plot or the matplotlib function that fits better the needs.

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(llcrnrlon=-0.5,llcrnrlat=39.8,urcrnrlon=4.,urcrnrlat=43.,
            resolution='i', projection='tmerc', lat_0 = 39.5, lon_0 = 1)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='#ddaa66',lake_color='aqua')
map.drawcoastlines()

map.readshapefile('../sample_files/comarques', 'comarques')

lightning_info = map.readshapefile('../sample_files/lightnings', 'lightnings')

print lightning_info

for info, lightning in zip(map.lightnings_info, map.lightnings):
    if float(info['amplitude']) < 0:
        marker = '_'
    else:
        marker = '+'
    map.plot(lightning[0], lightning[1], marker=marker, color='m', markersize=8,␣
↪markeredgewidth=2)

plt.show()
```

The example shows the lightnings fallen over Catalonia during a storm



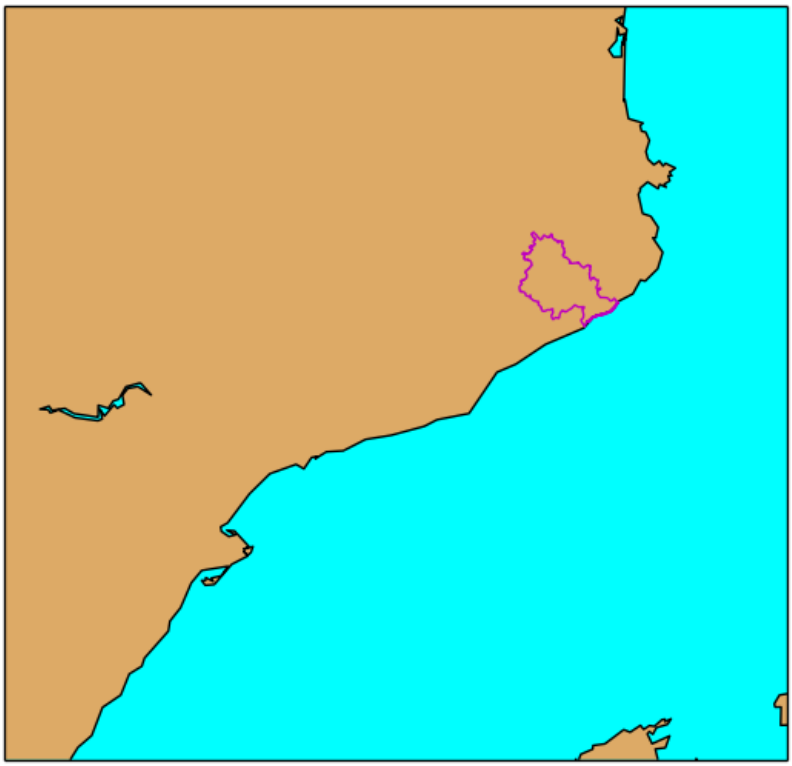
- The second parameter has been named lightnings, and the Basemap instance map, so the shapefile elements can be accessed using map.lightnings for geometries and map.lightnings_info for accessing the elements fields
- The shapefile method returns a sequence with the number of elements, the geometry type with the codes defined here and the bounding box
- **Line 17 shows a possible way to iterate all the elements**
  - zip will join each geometry with its field values
  - each element can be iterated with a for as when iterating a dict
- In the example, a field named amplitude can be used to guess if the lightning had positive or negative current and draw a different symbol for each case
- The points are plotted with the method plot, using the marker attribute to change the symbol

1.5.3 Polygon information

This example shows how to use the shapefile attributes to select only some geometries.

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(llcrnrlon=-0.5,llcrnrlat=39.8,urcrnrlon=4.,urcrnrlat=43.,
              resolution='i', projection='tmerc', lat_0 = 39.5, lon_0 = 1)
```

```
map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='#ddaa66',lake_color='aqua')
map.drawcoastlines()

map.readshapefile('../sample_files/comarques', 'comarques', drawbounds = False)

for info, shape in zip(map.comarques_info, map.comarques):
    if info['nombre'] == 'Selva':
        x, y = zip(*shape)
        map.plot(x, y, marker=None,color='m')
plt.show()
```

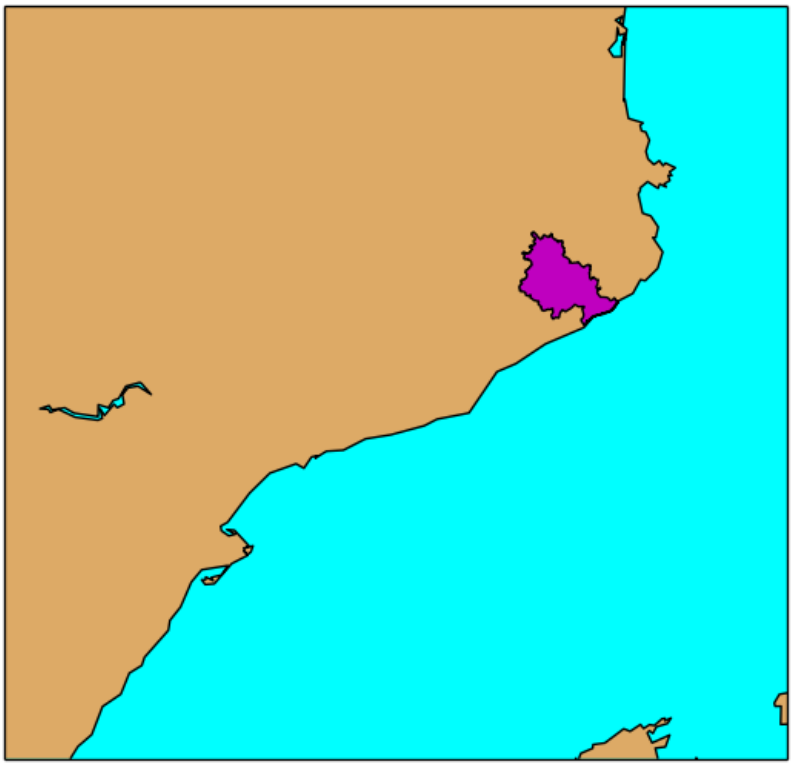


- To iterate all the elements, use zip as in the last example
- There is a field called 'nombre' that can be used to filter. In the example only the value 'Selva' is selected
- To plot a line, the x and y coordinates must be in separated arrays, but the geometry gives each point as a pair. There is an explanation on how to do it 'here <http://stackoverflow.com/questions/13635032/what-is-the-inverse-function-of-zip-in-python>'_
- The shape is plotted using the plot method, eliminating the markers to get only a line

1.5.4 Filling polygons

The basic way to plot a shapefile doesn't fill the polygons if this is the shape type. Here's how to do it:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
from matplotlib.collections import PatchCollection
from matplotlib.patches import PathPatch
import numpy as np

fig     = plt.figure()
ax      = fig.add_subplot(111)

map = Basemap(llcrnrlon=-0.5,llcrnrlat=39.8,urcrnrlon=4.,urcrnrlat=43.,
             resolution='i', projection='tmerc', lat_0 = 39.5, lon_0 = 1)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='#ddaa66',lake_color='aqua')
map.drawcoastlines()

map.readshapefile('../sample_files/comarques', 'comarques', drawbounds = False)

patches   = []

for info, shape in zip(map.comarques_info, map.comarques):
    if info['nombre'] == 'Selva':
        patches.append( Polygon(np.array(shape), True) )

ax.add_collection(PatchCollection(patches, facecolor= 'm', edgecolor='k',␣
↪linewidths=1., zorder=2))

plt.show()
```



- matplotlib uses a class called PatchCollection, which is a set shaped to add filled polygons, as explained at the official docs.
- The shapes, in this case, are of the type Polygon. To create it, the coordinates must be in a numpy array. The second argument sets the polygon to be closed

I learned it how to do it at StackOverflow

CHAPTER 2

All Basemap methods

2.1 Basemap

Any map created with the Basemap library must start with the creation of a Basemap instance

mpl_toolkits.basemap.Basemap(llcrnrlon=None, llcrnrlat=None, urcrnrlon=None, urcrnrlat=None, llcrnrx=None, llcrnry=None, urcrnrx=None, urcrnry=None, width=None, height=None, projection='cyl', resolution='c', area_thresh=None, rsphere=6370997.0, ellps=None, lat_ts=None, lat_1=None, lat_2=None, lat_0=None, lon_0=None, lon_1=None, lon_2=None, o_lon_p=None, o_lat_p=None, k_0=None, no_rot=False, suppress_ticks=True, satellite_height=35786000, boundinglat=None, fix_aspect=True, anchor='C', celestial=False, round=False, epsg=None, ax=None)

The class constructor has many possible arguments, and all of them are optional:

- **resolution: The resolution of the included coastlines, lakes, and so on. The options are c (crude, the default), l (low), i (inte**
  - None option is a good one if a Shapefile will be used instead of the included files, since no data must be loaded and the performance rises a lot.
- area_thresh: The threshold under what no coast line or lake will be drawn. Default 10000,1000,100,10,1 for resolution c, l, i, h, f.
- rsphere: Radius of the sphere to be used in the projections. Default is 6370997 meters. If a sequence is given, the first two elements are taken as the radius of the ellipsoid.
- ellps: An ellipsoid name, such as 'WGS84'. The allowed values are defined at pyproj.pj_ellps
- suppress_ticks: Suppress automatic drawing of axis ticks and labels in map projection coordinates
- fix_aspect: Fix aspect ratio of plot to match aspect ratio of map projection region (default True)
- anchor: The place in the plot where the map is anchored. Default is C, which means map is centered. Allowed values are C, SW, S, SE, E, NE, N, NW, and W
- celestial: Use the astronomical conventions for longitude (i.e. negative longitudes to the east of 0). Default False. Implies resolution=None

- ax: set default axes instance

2.1.1 Passing the bounding box

The following arguments are used to set the extent of the map.

To see some examples and explanations about setting the bounding box, take a look at the *Extension* section.

The first way to set the extent is by defining the map bounding box in geographical coordinates:

Argument	Description
llcrnrlon	The lower left corner geographical longitude
llcrnrlat	The lower left corner geographical latitude
urcrnrlon	The upper right corner geographical longitude
urcrnrlat	The upper right corner geographical latitude

An other option is setting the bounding box, but using the projected units:

Argument	Description
llcrnrx	The lower left corner x coordinate, in the projection units
llcrnry	The lower left corner y coordinate, in the projection units
urcrnrx	The upper right corner x coordinate, in the projection units
urcrnry	The upper right corner y coordinate, in the projection units

Finally, the last option is to set the bounding box giving the center point in geographical coordinates, and the width and height of the domain in the projection units

Argument	Description
width	The width of the map in the projection units
height	The height of the map in the projection units
lon_0	The longitude of the center of the map
lat_0	The latitude of the center of the map

2.1.2 Using the Basemap instance to convert units

The basemap instance can be used to calculate positions on the map and the inverse operation, converting positions on the map to geographical coordinates.

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='aeqd', lon_0 = 10, lat_0 = 50)

print map(10, 50)
print map(20015077.3712, 20015077.3712, inverse=True)
```

The output will be:

(20015077.3712, 20015077.3712) (10.000000000000002, 50.000000000000014)

When inverse is False, which is its default value, the input values are a longitude and a latitude, and the output, the position of this point in the map coordinates. When *inverse* is set to true, the behavior is the opposite, the arguments are an x and y positions in the map coordinates, and the output is the position of this point in geographical coordinates.

2.1.3 Basemap object fields

When a Basemap object is created, has some fields with data:

- **Some of them contain the polygons with the resolution set with the resolution parameter. They are all matplotlib Polygon**
  - landpolygons
  - lakepolygons
  - boundarylons
  - coastpolygons
  - coastpolygontypes
  - coastsegs
- **Other fields give information about the used projection**
  - proj4string A string that can be used with proj4 (or GDAL) to have the used projection definition
  - projection The code of the used projection, as indicated in the projection argument
  - projparams A dict with the projection parameters. The ones passed and the ones put by default by Basemap
  - rmajor The semi-major axis of the ellipsoid used with the projection
  - rminor The semi-minor axis of the ellipsoid used with the projection
  - xmax, ymax, xmin, ymin The bounding box in the projection units
  - anchor The point on the map where the axis coordinates start. By default is the center of the map, but can be changed to any corner or side
  - celestial indicates if the longitudes west of Greenwich are negative

2.2 Plotting data

2.2.1 annotate

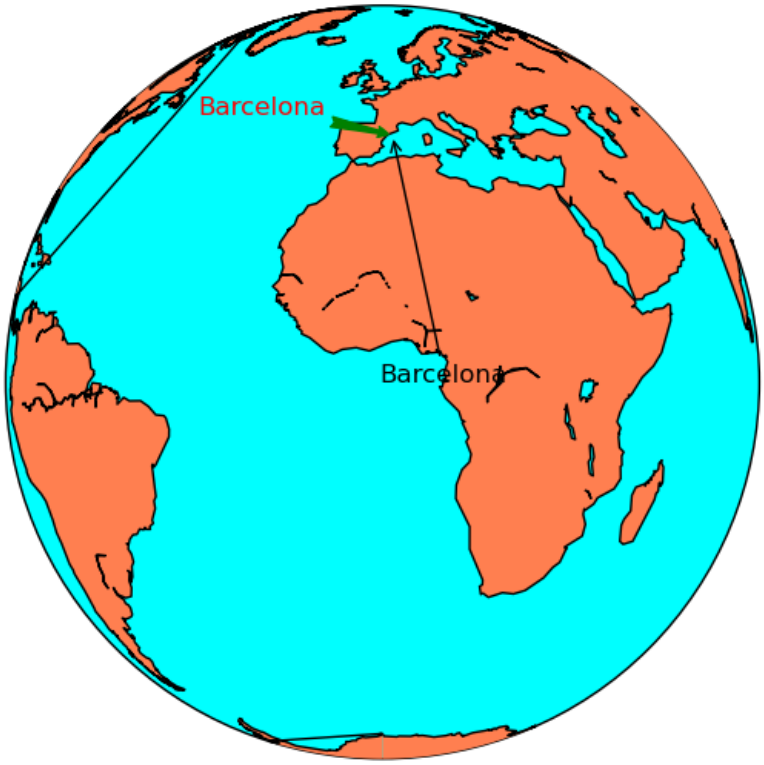
Creates text with an arrow indicating the point of interest. To create a *text* without an arrow, look at the *text* section.

annotate(*args, **kwargs)

- The text method does not belong to Basemap, but directly to matplotlib, so it must be called from the plot or axis instance
- The first argument is the text string
- xy is a list with the x and y coordinates of the point pointed by the arrow. This will be interprete depending on the xycoords argument
- **xycoords indicates the type of coordinates used in xy:**
  - data means the coordinates are the ones used by the data (the projection coordinates)
  - offset points means an offset in points
  - axes pixels indicates pixels from the lower left corner of the axes
  - The other options are at the annotation docs

- xytext a list with the x and y coordinates of the text, an the beginning of the arrow
- textcoords indicates the type of coordinates used in xytext, with the same options as in xycoords
- arrowprops this optional argument sets the arrow properties, as explained in the Line2D docs
- color the color of the text. This page explains all the color options

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt


map = Basemap(projection='ortho',
              lat_0=0, lon_0=0)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()


x, y = map(2, 41)
x2, y2 = (-90, 10)

plt.annotate('Barcelona', xy=(x, y),  xycoords='data',
                xytext=(x2, y2), textcoords='offset points',
                color='r',
                arrowprops=dict(arrowstyle="fancy", color='g')
                )

x2, y2 = map(0, 0)
plt.annotate('Barcelona', xy=(x, y),  xycoords='data',
                xytext=(x2, y2), textcoords='data',
                arrowprops=dict(arrowstyle="->")
                )
plt.show()
```



2.2.2 barbs

Plots wind barbs on the map

barbs(x, y, u, v, *args, **kwargs)

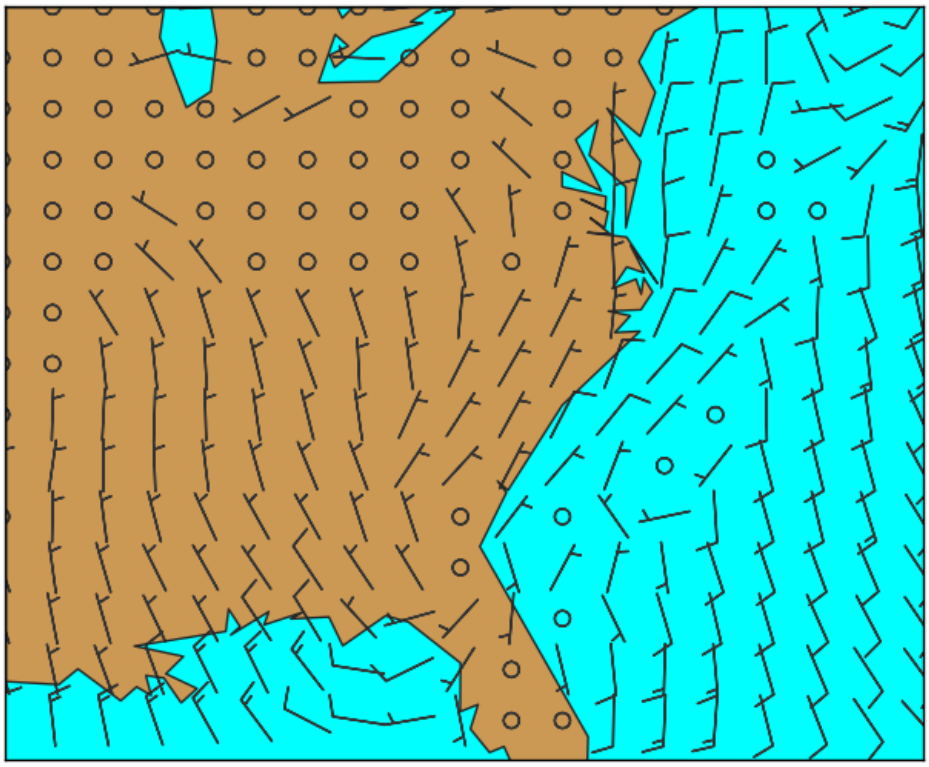
The barbs docs from the matplotlib documentation is much more detailed.

- x and y give the positions of the grid data if the latlon argument is true, the values are supposed to be in geographical coordinates. If not, in the map coordinates
- **u and v are the left-right and up-down magnitudes of the wind in knots (the values to create the barbs are 5, 10, 15 and so**
  - Note that they are NOT in north-south ans west-east. If the input projection has non-cylindrical projections (those other than cyl, merc, cyl, gall and mill), the u and v should be rotated, using *rotate_vector* or *transform_scalar* methods

There are many other optional arguments, documented at the matplotlib docs. Some interesting arguments are:

- pivot changes the point of rotation of the barbs. By default is 'tip', but can be changed to 'middle'
- barbcolor changes the color. If a sequence, the colors are alternated with the sequence values
- fill_empty fills the circles when the wind is lower than 5 kt
- barb_increments can be used to change the values where the barbs add ticks and flags

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from osgeo import gdal
import numpy as np


map = Basemap(llcrnrlon=-93.7, llcrnrlat=28., urcrnrlon=-66.1, urcrnrlat=39.5,
              projection='lcc', lat_1=30., lat_2=60., lat_0=34.83158, lon_0=-98.)

ds = gdal.Open("../sample_files/wrf.tiff")
lons = ds.GetRasterBand(4).ReadAsArray()
lats = ds.GetRasterBand(5).ReadAsArray()
u10 = ds.GetRasterBand(1).ReadAsArray()
v10 = ds.GetRasterBand(2).ReadAsArray()

x, y = map(lons, lats)

yy = np.arange(0, y.shape[0], 4)
xx = np.arange(0, x.shape[1], 4)

points = np.meshgrid(yy, xx)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='#cc9955', lake_color='aqua', zorder = 0)
map.drawcoastlines(color = '0.15')

map.barbs(x[points], y[points], u10[points], v10[points],
    pivot='middle', barbcolor='#333333')


plt.show()
```



- **The main problem when using barbs is that the point density may be too high, and the method can't skip points by itself.**
  - To do it, a matrix taking only 1/4 th of the elements is created
  - The matrix points has the element positions to take
  - The arguments passed to barbs are selected using the points matrix, so taking only one of every 4 elements
- u and v are supposed to be in the map coordinates, not north-south or west-east. If they were in geographical coordinates, *rotate_vector* could be used to rotate them properly
- The pivot argument has been used to rotate the barbs from the middle, since the effect rotating from the upper part (pivot = 'tip') creates strange effects when the wind rotates

2.2.3 contour

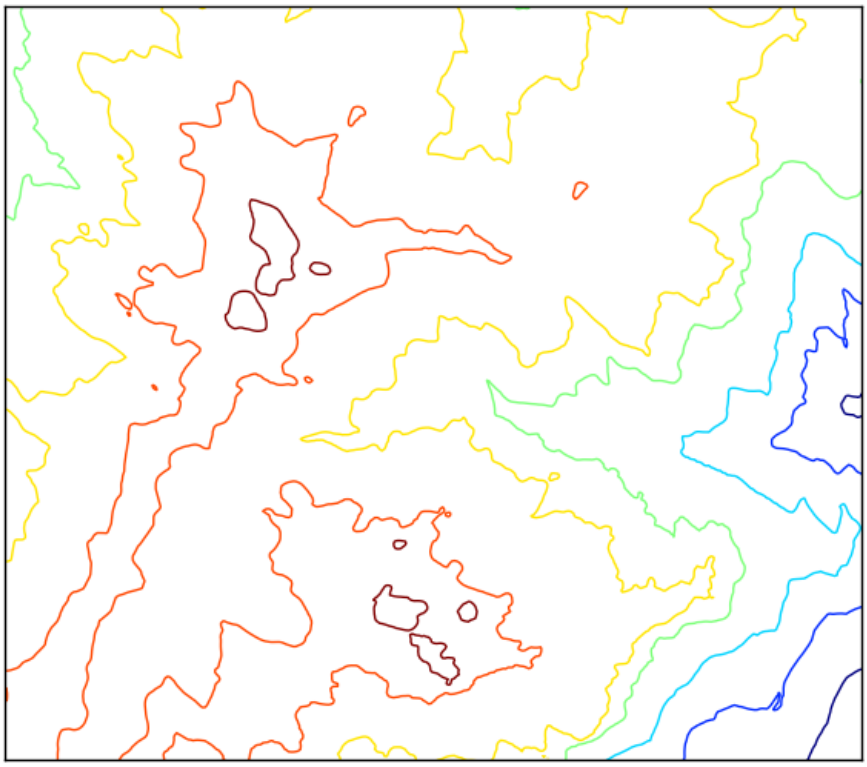
Creates a contour plot.

contour(x, y, data)

- x and y are matrices of the same size as data, containing the positions of the elements in the map coordinates
- data is the matrix containing the data values to plot
- A fourth argument can be passed, containing a list of the levels to use when creating the contour

- The default colormap is *jet*, but the argument *cmap* can be used to change the behavior
- The argument tri = True makes the grid to be assumed as unstructured. See this post to check the differences
- Other possible arguments are documented in the matplotlib function docs
- Labels can be added to the contour result, as in the *contour* example at basic functions section

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from osgeo import gdal
from numpy import linspace
from numpy import meshgrid

map = Basemap(projection='tmerc',
              lat_0=0, lon_0=3,
              llcrnrlon=1.819757266426611,
              llcrnrlat=41.583851612359275,
              urcrnrlon=1.841589961763497,
              urcrnrlat=41.598674173123)

ds = gdal.Open("../sample_files/dem.tiff")
data = ds.ReadAsArray()

x = linspace(0, map.urcrnrx, data.shape[1])
y = linspace(0, map.urcrnry, data.shape[0])

xx, yy = meshgrid(x, y)

map.contour(xx, yy, data)

plt.show()
```



2.2.4 contourf

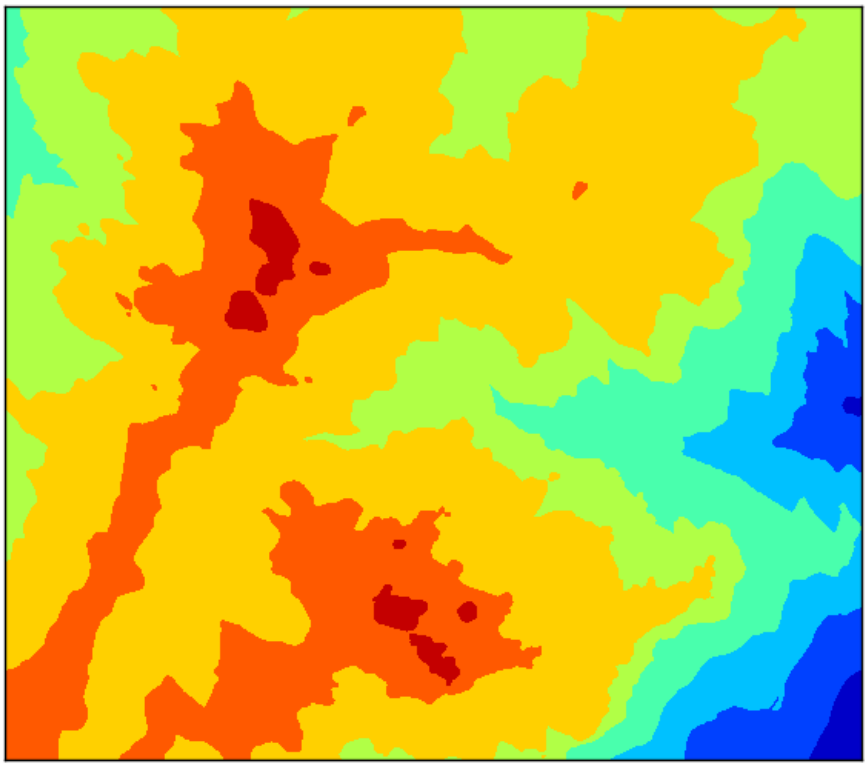
Creates a filled contour plot.

contourf(x, y, data)

- x and y are matrices of the same size as data, containing the positions of the elements in the map coordinates
- data is the matrix containing the data values to plot
- The default colormap is *jet*, but the argument *cmap* can be used to change the behavior
- A fourth argument can be passed, containing a list of the levels to use when creating the contourf
- The argument tri = True makes the grid to be assumed as unstructured. See this post to check the differences
- Other possible arguments are documented in the matplotlib function docs

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from osgeo import gdal
from numpy import linspace
from numpy import meshgrid

map = Basemap(projection='tmerc',
              lat_0=0, lon_0=3,
```

```
                llcrnrlon=1.819757266426611,
                llcrnrlat=41.583851612359275,
                urcrnrlon=1.841589961763497,
                urcrnrlat=41.598674173123)

ds = gdal.Open("../sample_files/dem.tiff")
data = ds.ReadAsArray()

x = linspace(0, map.urcrnrx, data.shape[1])
y = linspace(0, map.urcrnry, data.shape[0])

xx, yy = meshgrid(x, y)

map.contourf(xx, yy, data)

plt.show()
```



2.2.5 hexbin

Plots an hexagonal bin from a set of positions. Can plot the number of occurrences in each bin (hexagon) or give a weight to each occurrence

hexbin(x, y, **kwargs)

The information is much better at the matplotlib.hexbin docs

- x and y are numpy arrays with the coordinates of the points. Regular lists won't work, they have to be numpy arrays.
- gridsize sets the number of bins (hexagons) in the x direction. By default is 100
- C argument is also a numpy array with the values at each point. Those values are processed by default with numpy.mean function at each bin (hexagon)
- reduce_C_function is the function applied to the elements in each bin (hexagon). By default, is numpy.mean
- **bins argument can change the behavior of the counting function.**
  - By default is None, which means that the number of occurrences are plotted.
  - 'log' makes the logarithm of the number of occurrences is plotted
  - An integer value divides the number of occurrences by this number (useful for percentages if the number is the total of points)
  - A sequence makes the lower bound of the bin to be used
- mincnt is the minimum of occurrences in each bin (hexagon) to be plotted. By default is 0, so to avoid plotting hexagons without occurrences, set it to 1
- cmap sets the color map
- edgecolors is the color of the hexagons border. This linewidths argument must not be None to see a result
- linewidths is the line width of the edge of the hexagons. By default is None, so no borders are plot
- alpha sets the transparency of the layer

Note: Old versions of the library don't supoort hexbin

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from numpy import array
from numpy import max


map = Basemap(llcrnrlon=-0.5,llcrnrlat=39.8,urcrnrlon=4.,urcrnrlat=43.,
              resolution='i', projection='tmerc', lat_0 = 39.5, lon_0 = 1)


map.readshapefile('../sample_files/lightnings', 'lightnings')

x = []
y = []
c = []

for info, lightning in zip(map.lightnings_info, map.lightnings):
    x.append(lightning[0])
    y.append(lightning[1])

    if float(info['amplitude']) < 0:
        c.append(-1 * float(info['amplitude']))
    else:
```

```
25          c.append(float(info['amplitude']))
26
27  plt.figure(0)
28
29  map.drawcoastlines()
30  map.readshapefile('../sample_files/comarques', 'comarques')
31
32  map.hexbin(array(x), array(y))
33
34  map.colorbar(location='bottom')
35
36
37
38  plt.figure(1)
39
40  map.drawcoastlines()
41  map.readshapefile('../sample_files/comarques', 'comarques')
42
43  map.hexbin(array(x), array(y), gridsize=20, mincnt=1, cmap='summer', bins='log')
44
45  map.colorbar(location='bottom', format='%.1f', label='log(# lightnings)')
46
47
48
49  plt.figure(2)
50
51  map.drawcoastlines()
52  map.readshapefile('../sample_files/comarques', 'comarques')
53
54  map.hexbin(array(x), array(y), gridsize=20, mincnt=1, cmap='summer', norm=colors.
    ↪LogNorm())
55
56  cb = map.colorbar(location='bottom', format='%d', label='# lightnings')
57
58  cb.set_ticks([1, 5, 10, 15, 20, 25, 30])
59  cb.set_ticklabels([1, 5, 10, 15, 20, 25, 30])
60
61
62
63
64  plt.figure(3)
65
66  map.drawcoastlines()
67  map.readshapefile('../sample_files/comarques', 'comarques')
68
69  map.hexbin(array(x), array(y), C = array(c), reduce_C_function = max, gridsize=20,␣
    ↪mincnt=1, cmap='YlOrBr', linewidths=0.5, edgecolors='k')
70
71  map.colorbar(location='bottom', label='Mean amplitude (kA)')
72
73
74  plt.show()
```

- The example creates four examples to show different options
- Lines 14 to 25 read all the points, corresponding to lightning data, as explained at the section *Reading point data*.
- The first example shows the minimum use of hexbin. Note that the bins are quite small, that the zones with no

occurrences are plotted with dark blue while the zones outside the shapefile data bounding box stay without data

- The second example changes the bin (hexagon) size using the gridsize argument, sets a minimum of occurrences to plot the data with mincnt and makes the colors to behave in a logarithmic scale with bins. Note that the color bar shows the value of the logarithm
- To avoid the color bar to show the logarithm value, I found this solution at StackOverflow that substitutes the bins argument but draws the same while getting a better color bar
- **The last example shows how to use the C argument.**
  - **The amplitude module of the lightning has been stored at the c variable, and passed to the C argument**
    - The reduce_C_function argument is used to show the maximum value in each bin instead of the mean
    - linewidths and edgecolors make a border to be plotted in each hexagon

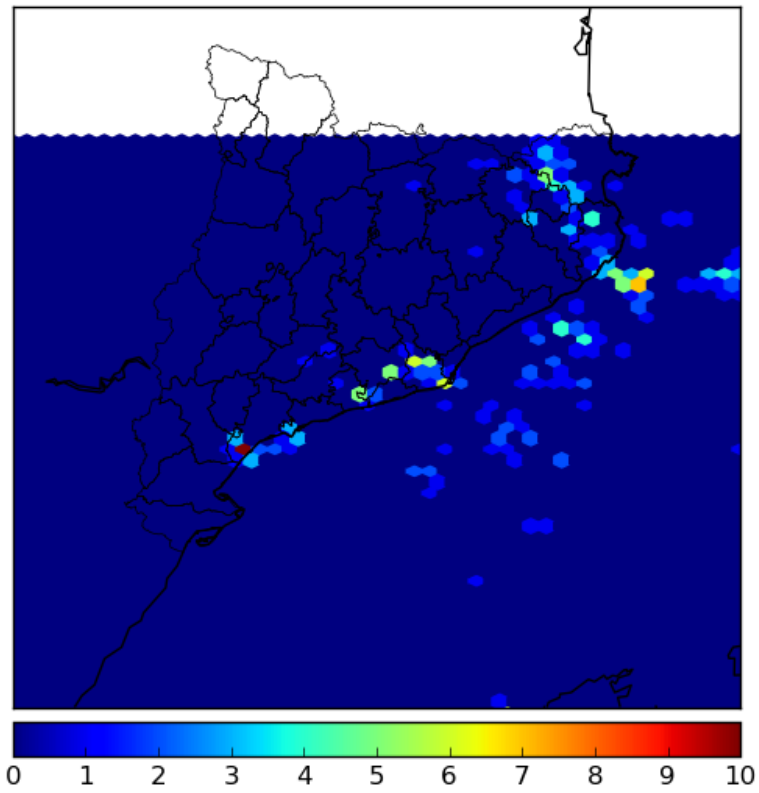


Fig. 2.1: Basic usage

2.2.6 imshow

Plots an image on the map. The image can be a regular rgb image, or a field colored with a cmap.

imshow(*args, **kwargs)

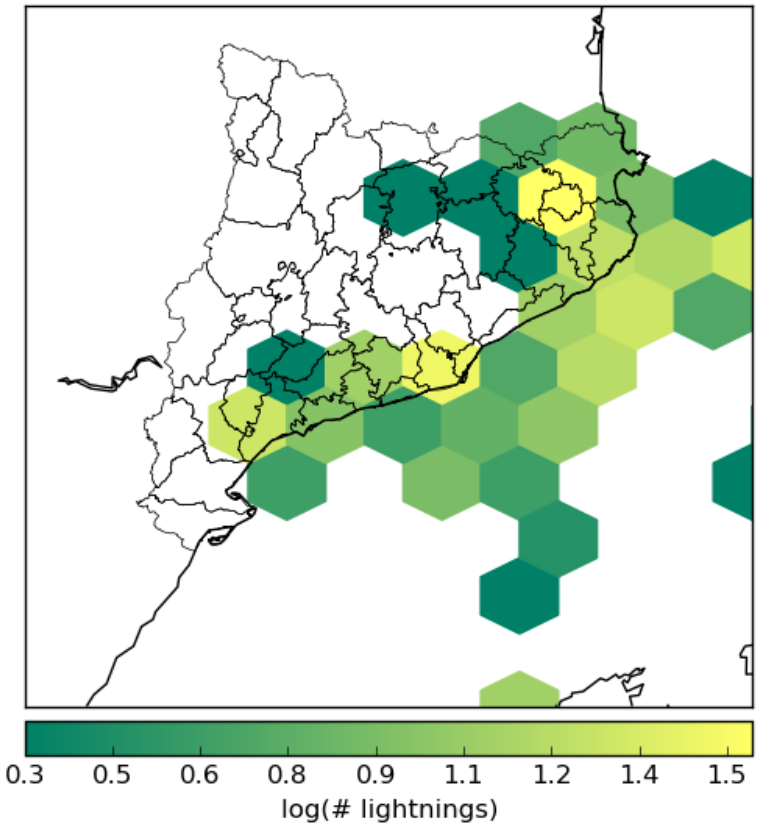


Fig. 2.2: Log scale, different hexagon size

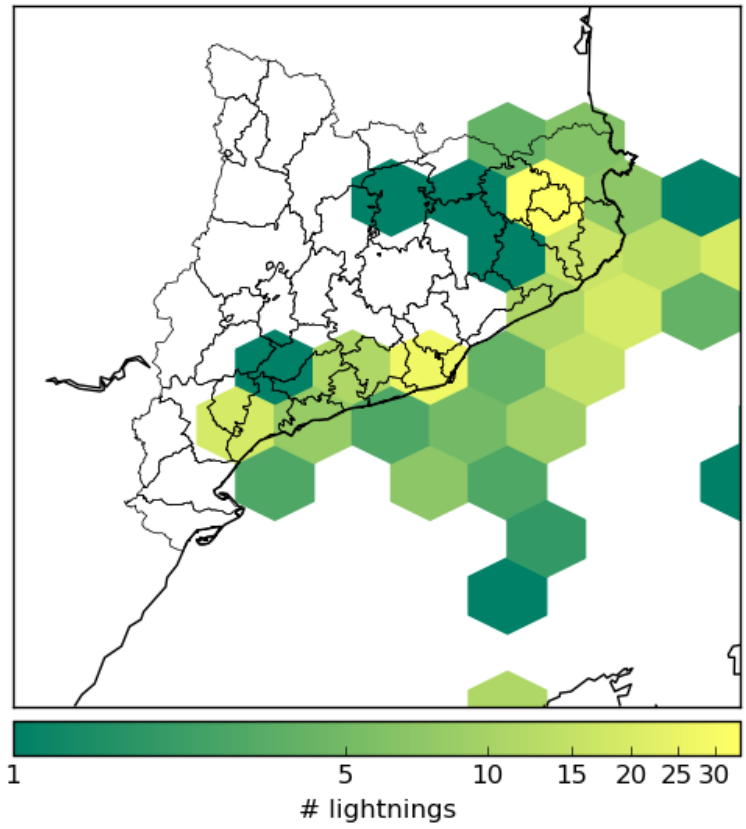


Fig. 2.3: Log scale with a more convenient color bar

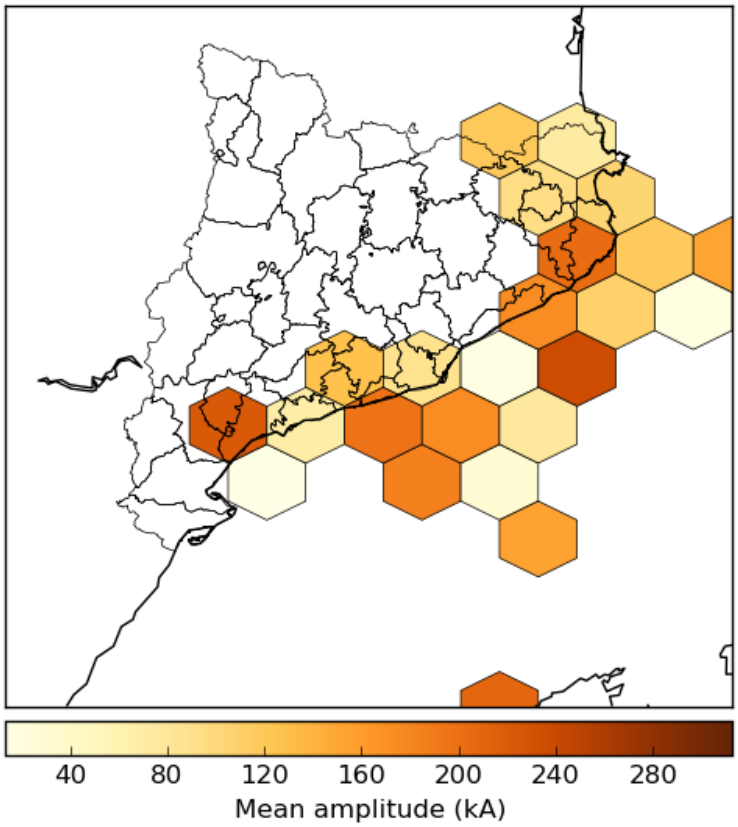


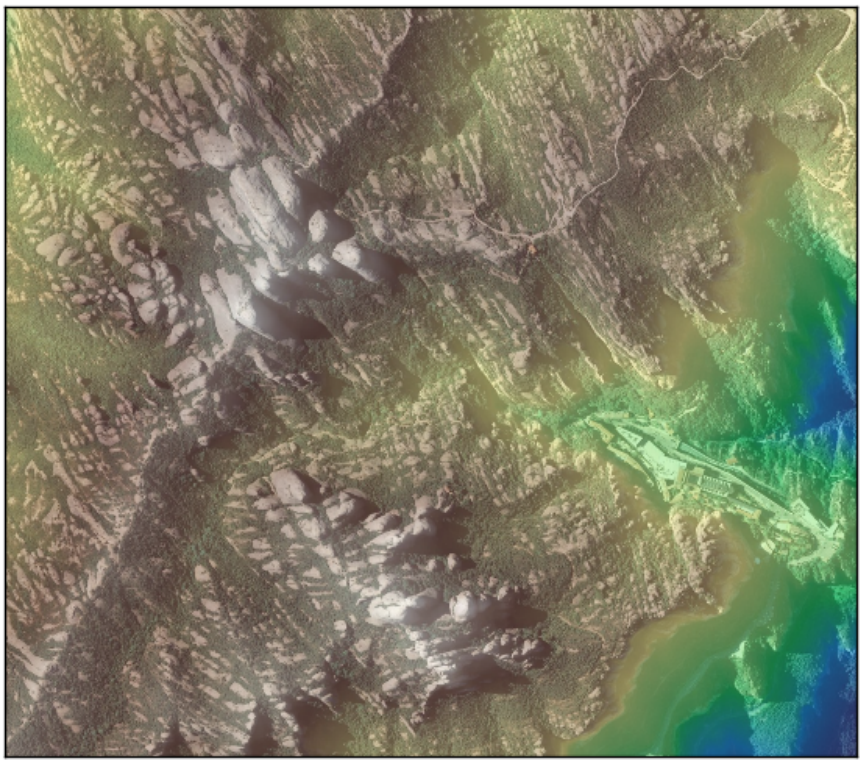
Fig. 2.4: Using the C argument and plotting the bin edges

As in other cases, the best docs are at the matplotlib documentation.

- The first argument is the image array. If it has three bands, RGB will be assumed, and the image will be plotted. If it has one band, a pseudocolor will be created according to the cmap argument (jet by default). Arrays with two or more than three bands will make the method tp raise an error
- extent sets the position of four corners of the image, in map coordinates. It has to be a sequence with the elements (x0, x1, y0, y1)
- origin changes the initial line position of the image. By default, the first line position is defined by image.origin, and this can be changed using the values 'upper' or 'lower'
- cmap is the desired colormap if the image has one band.
- alpha sets the transparency with values from 0 to 1
- interpolation changes the interpolation mode when resizing the image. Possible values are 'none', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos

The first example shows hot to use jpg image and a field as a pseudocolored image:

```
from mpl_toolkits.basemap import Basemap
from mpl_toolkits.basemap import shiftgrid
import matplotlib.pyplot as plt
from osgeo import gdal
import numpy as np

map = Basemap(projection='tmerc',
              lat_0=0, lon_0=3,
              llcrnrlon=1.819757266426611,
              llcrnrlat=41.583851612359275,
              urcrnrlon=1.841589961763497,
              urcrnrlat=41.598674173123)

ds = gdal.Open("../sample_files/dem.tiff")
elevation = ds.ReadAsArray()

map.imshow(plt.imread('../sample_files/orthophoto.jpg'))

map.imshow(elevation, cmap = plt.get_cmap('terrain'), alpha = 0.5)

plt.show()
```

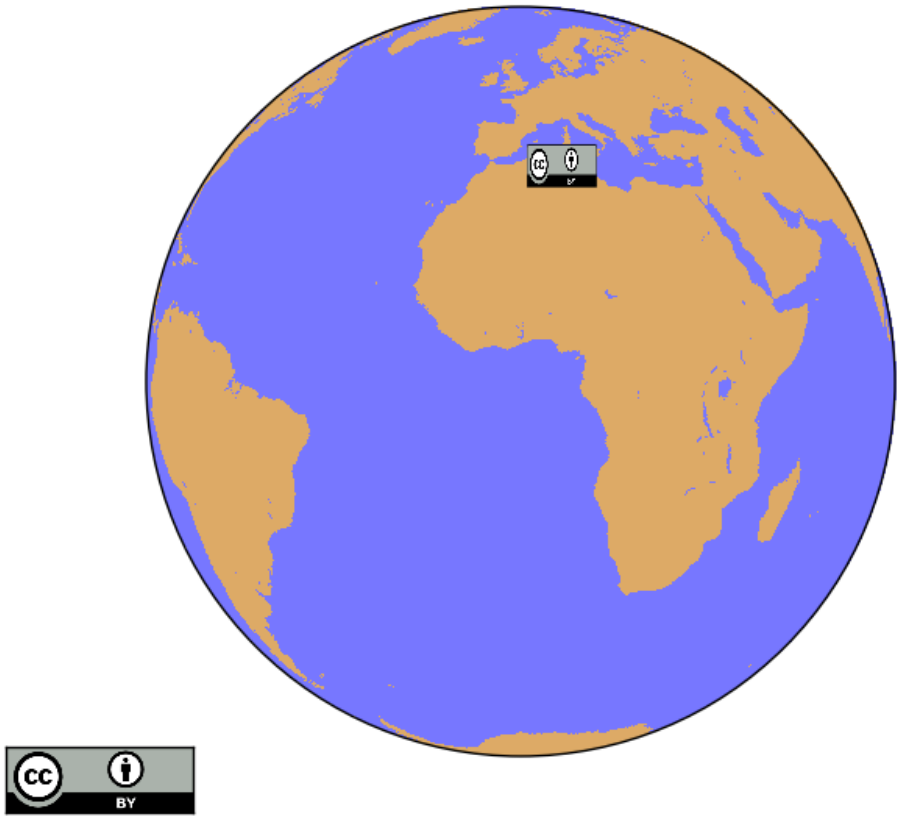


- plt.imread reads any common image file into a numpy array
- The elevation data is plotted with the terrain colormap and with an alpha value to allow viewing both layers at once
- Note that the extent hasn't been set, since the map extent is the same as the image extension

The second example shows how to add a logo directly on the map or on a new created axis:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
fig = plt.figure()

map = Basemap(projection='ortho',
              lat_0=0, lon_0=0)

map.drawlsmask(land_color = "#ddaa66",
               ocean_color="#7777ff",
               resolution = 'l')

x0, y0 = map(1., 31.)
x1, y1 = map(15., 39.)

plt.imshow(plt.imread('../sample_files/by.png'),  extent = (x0, x1, y0, y1))

axicon = fig.add_axes([0.1, 0., 0.15, 0.15])
axicon.imshow(plt.imread('../sample_files/by.png'), origin = 'upper')
```

```
axicon.set_xticks([])
axicon.set_yticks([])

plt.show()
```



- The first icon is drawn on the map using the extent argument. The coordinates have been converted to the map units
- **The second logo is outside the globe, so no map coordinates can be used to place it properly**
  - An axis named axicon is created. The numbers set the x and y position, and width and height of the axis. All them in fractions of the plot size.
  - The image is plotted, but with the argument origin to avoid to have upside down
  - xticks and ytixcks are set to null, so they are not drawn on the logo

I learned how to do it from these examples: 1 and 2

2.2.7 pcolor

The behavior of this function is almost the same as in *pcolormesh*. A good explanation here

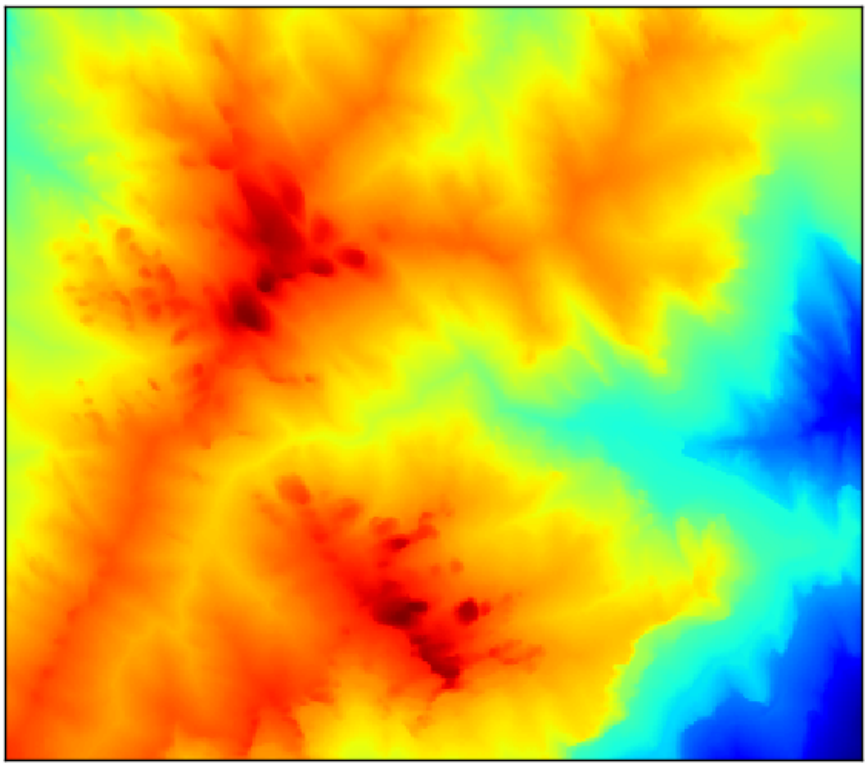
2.2.8 pcolormesh

Creates a pseudo-color plot

pcolormesh(x, y, data, *args, **kwargs)

- x and y are matrices of the same size as data, containing the positions of the elements in the map coordinates
- data is the matrix containing the data values to plot
- The default colormap is *jet*, but the argument *cmap* can be used to change the behavior
- Other possible arguments are documented in the matplotlib function docs

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from osgeo import gdal
from numpy import linspace
from numpy import meshgrid

map = Basemap(projection='tmerc',
              lat_0=0, lon_0=3,
              llcrnrlon=1.819757266426611,
              llcrnrlat=41.583851612359275,
              urcrnrlon=1.841589961763497,
              urcrnrlat=41.598674173123)

ds = gdal.Open("../sample_files/dem.tiff")
data = ds.ReadAsArray()

x = linspace(0, map.urcrnrx, data.shape[1])
y = linspace(0, map.urcrnry, data.shape[0])

xx, yy = meshgrid(x, y)

map.pcolormesh(xx, yy, data)

plt.show()
```



2.2.9 plot

Plots markers or lines on the map

The function has the following arguments:

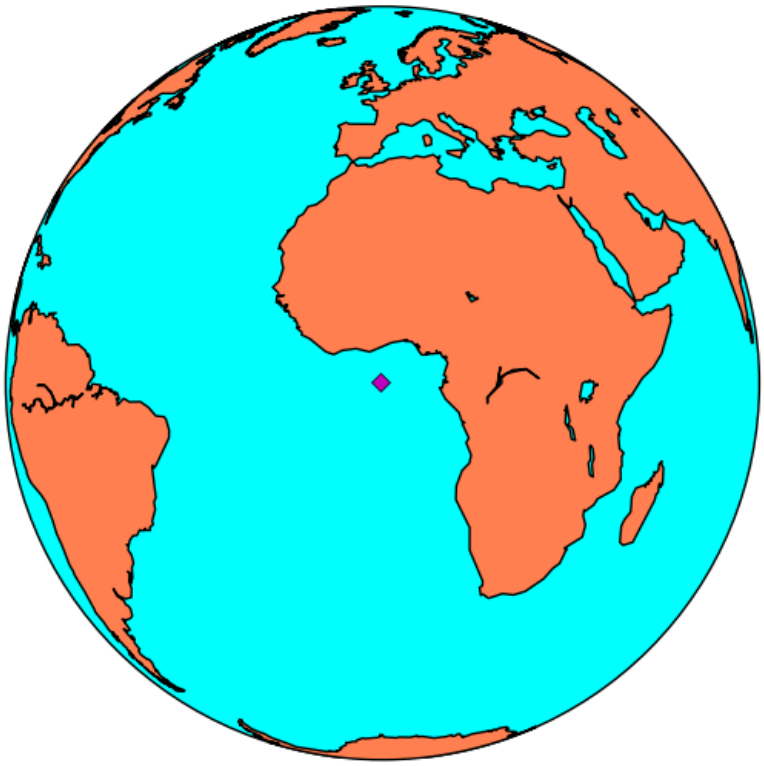
plot(x, y, *args, **kwargs)

- x and y can be either a float with the position of a marker in the projection units, or lists with the points form drawing a line
- If latlon keyword is set to True, x,y are interpreted as longitude and latitude in degrees. Won't work in old *basemap* versions
- By default, the marker is a point. This page explains all the options
- By default, the color is black (k). This page explains all the color options

The first example shows a single point:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='ortho',
              lat_0=0, lon_0=0)
```

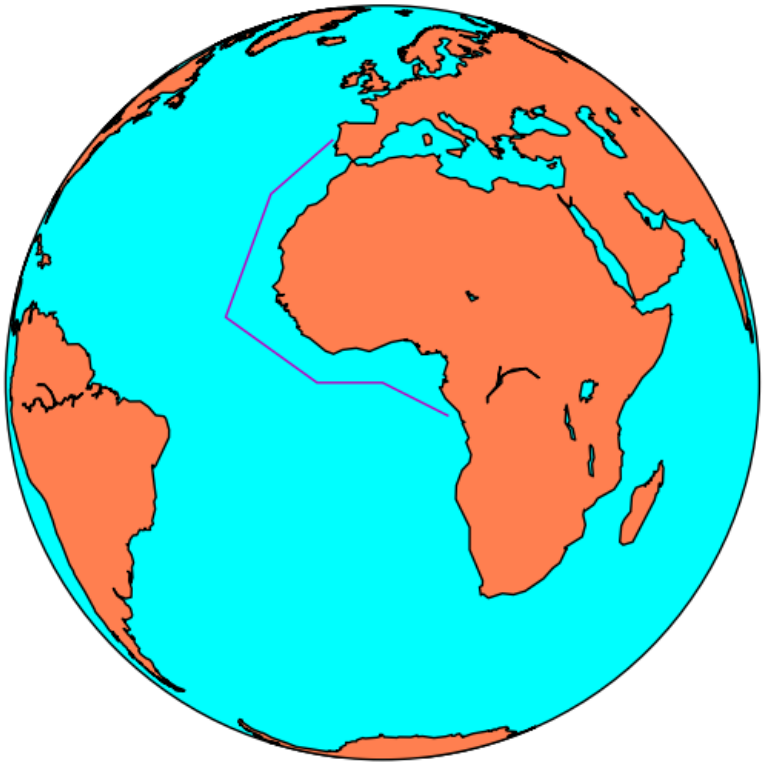
```
map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()

x, y = map(0, 0)

map.plot(x, y, marker='D',color='m')

plt.show()
```



If the arguments are arrays, the output is a line (without markers in this case):

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='ortho',
              lat_0=0, lon_0=0)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()

lons = [-10, -20, -25, -10, 0, 10]
lats = [40, 30, 10, 0, 0, -5]
```

```
x, y = map(lons, lats)

map.plot(x, y, marker=None,color='m')

plt.show()
```



2.2.10 quiver

Plots a vector field on the map. Many of the behavior is similar to the *barbs* method.

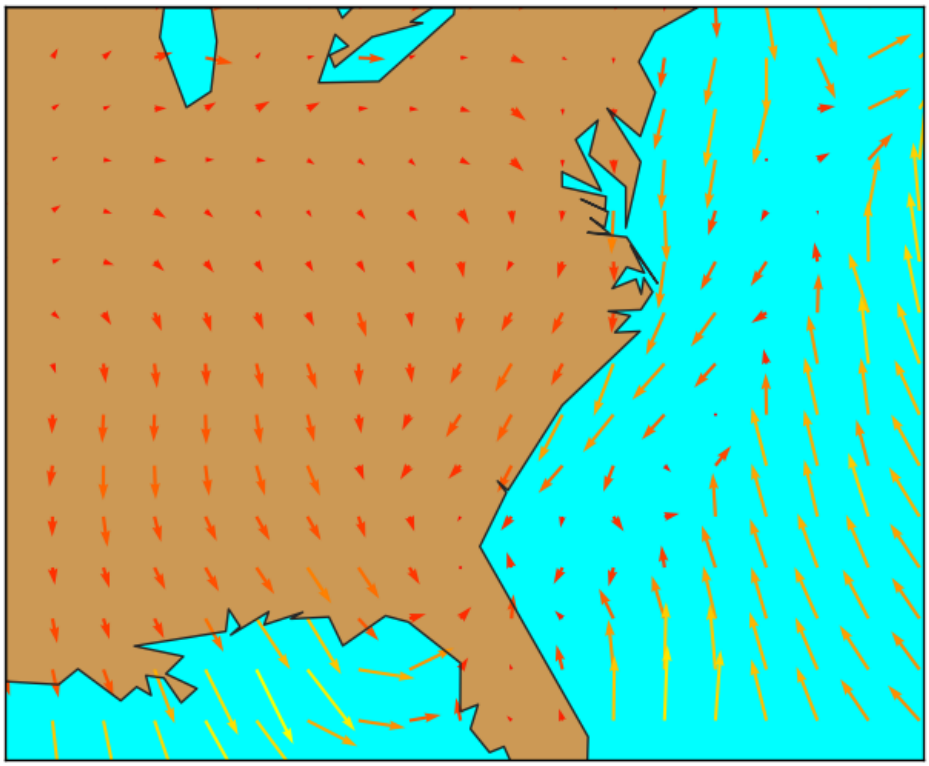
quiver(x, y, u, v, *args, **kwargs)

The documentation at matplotlib is much more detailed than at the basemap docs.

- x and y give the positions of the grid data if the latlon argument is true, the values are supposed to be in geographical coordinates. If not, in the map coordinates
- **u and v are the left-right and up-down magnitudes**
  - Note that they are NOT in north-south ans west-east. If the input projection has non-cylindrical projections (those other than cyl, merc, cyl, gall and mill), the u and v should be rotated, using *rotate_vector* or *transform_scalar* methods
- The fifth argument, which is optional, sets a value to assign the color to the arrow
- scale makes the arrows to be longer or shorter

- minshaft value below which arrow grows. The value should be greater than 1
- pivot changes the point of rotation of the arrows. By default is 'tip', but can be changed to 'middle'

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from osgeo import gdal
import numpy as np


map = Basemap(llcrnrlon=-93.7, llcrnrlat=28., urcrnrlon=-66.1, urcrnrlat=39.5,
              projection='lcc', lat_1=30., lat_2=60., lat_0=34.83158, lon_0=-98.)

ds = gdal.Open("../sample_files/wrf.tiff")
lons = ds.GetRasterBand(4).ReadAsArray()
lats = ds.GetRasterBand(5).ReadAsArray()
u10 = ds.GetRasterBand(1).ReadAsArray()
v10 = ds.GetRasterBand(2).ReadAsArray()
speed = np.sqrt(u10*u10 + v10*v10)

x, y = map(lons, lats)

yy = np.arange(0, y.shape[0], 4)
xx = np.arange(0, x.shape[1], 4)

points = np.meshgrid(yy, xx)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='#cc9955', lake_color='aqua', zorder = 0)
map.drawcoastlines(color = '0.15')

map.quiver(x[points], y[points],
    u10[points], v10[points], speed[points],
    cmap=plt.cm.autumn)

plt.show()
```



- **The main problem when using quiver is that the point density may be too high, and the method can't skip points by itself.**
  - To do it, a matrix taking only 1/4 th of the elements is created
  - The matrix points has the element positions to take
  - The arguments passed to barbs are selected using the points matrix, so taking only one of every 4 elements
- u and v are supposed to be in the map coordinates, not north-south or west-east. If they were in geographical coordinates, *rotate_vector* could be used to rotate them properly
- The wind speed is calculated to pass the color values to the quiver method. The length of the array must be the same as in x, y, u & v, of course

2.2.11 scatter

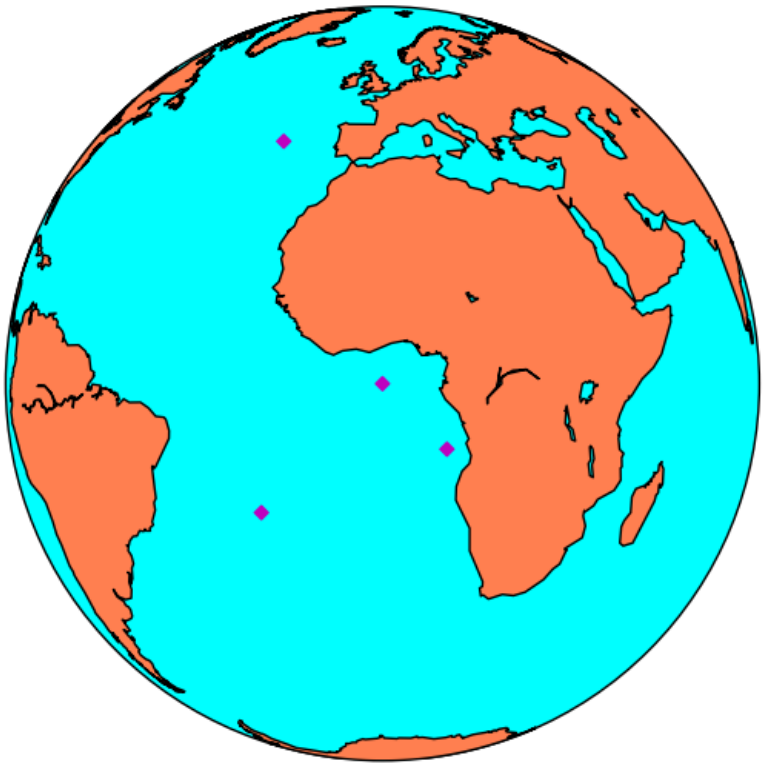
Plot multiple markers on the map

scatter(x, y, *args, **kwargs)

- x and y is a list of points to add to the map as markers
- If latlon keyword is set to True, x,y are interpreted as longitude and latitude in degrees. Won't work in old *basemap* versions

- By default, the marker is a point. This page explains all the options
- By default, the color is black (k). This page explains all the color options

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='ortho',
              lat_0=0, lon_0=0)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()

lons = [0, 10, -20, -20]
lats = [0, -10, 40, -20]

x, y = map(lons, lats)

map.scatter(x, y, marker='D',color='m')

plt.show()
```



2.2.12 streamplot

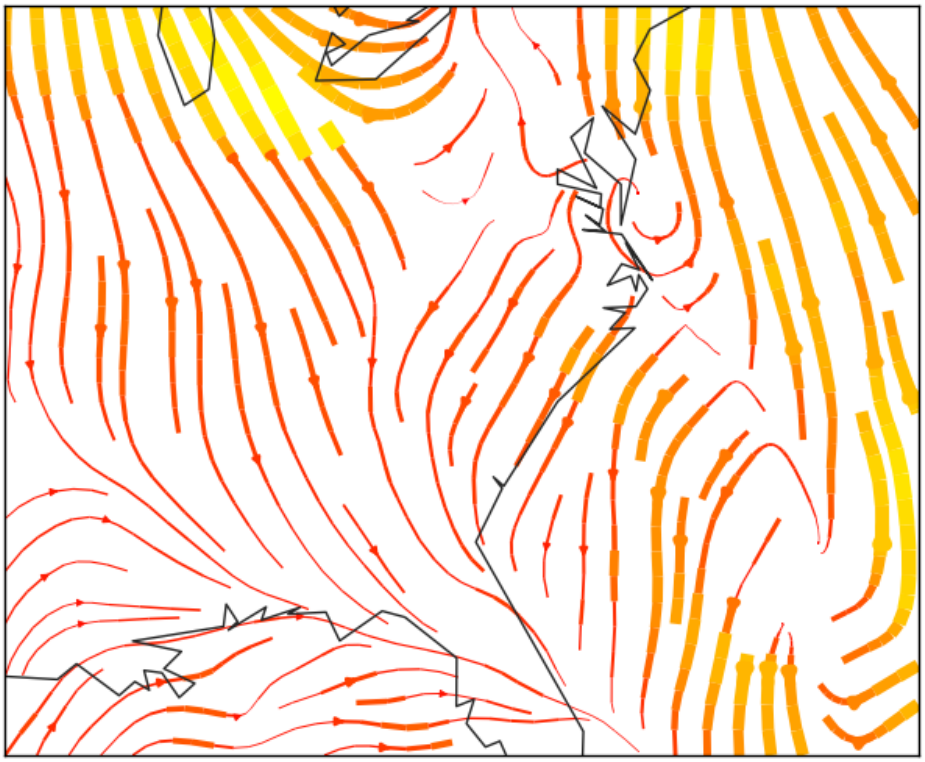
Plots streamlines from a vectorial field.

streamplot(x, y, u, v, *args, **kwargs)

- **x and y are matrices of the same size as *u* and *v* data, containing the positions of the elements in the map coordinates.**
  - As the docs explain, x and y must be evenly spaced. This mean that when the original values come from a different projection, the data matrix must be re-projected, and the x and y matrices re-calculated, as you can see in the example.
  - To calculate an evenly spaced grid, the method *makegrid* can be used. It's better to use the *returnxy=True* attribute to get the grid in the map projection units.
- **u and v are the left-right and up-down magnitudes**
  - Note that they are NOT in north-south ans west-east. If the input projection has non-cylindrical projections (those other than cyl, merc, cyl, gall and mill), the u and v should be rotated, using *rotate_vector* or *transform_scalar* methods
  - The dimensions must be the same as *x* and *y*
- **color can set the same color for the streamlines, or change depending of the data:**
  - If the value is a scalar, all the streamlines will have the indicated color, depending on the colormap
  - If the value is an array of the same size as data (the module of the wind in the example), the color will change according to it, using the color map
- cmap sets the color map
- **linewidth sets the width of the streamlines in a similar way as the color**
  - If is a scalar, all the streamlines have the indicated width
  - If is an array, the streamline width will change according to the values of the array
- density sets the closeness of streamlines to plot. a 1 value means the domains is divided in 30x30, with only one streamline crossing each sector. If a list with two elements is passed, the x and y densities are set to different values
- norm Normalizes the scale to set luminance data
- arrowsize scales the arrow size
- arrowstyle stes the arrow style. The docs are at FancyArrowPatch
- minlength sets the minimum length of the streamline in the map coordinates
- The best documentation for the function is not the one in Basemap, but in matplotlib

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from osgeo import gdal
import numpy


map = Basemap(llcrnrlon=-93.7, llcrnrlat=28., urcrnrlon=-66.1, urcrnrlat=39.5,
              projection='lcc', lat_1=30., lat_2=60., lat_0=34.83158, lon_0=-98.)

ds = gdal.Open("../sample_files/wrf.tiff")
lons = ds.GetRasterBand(4).ReadAsArray()
lats = ds.GetRasterBand(5).ReadAsArray()
```

```
u10 = ds.GetRasterBand(1).ReadAsArray()
v10 = ds.GetRasterBand(2).ReadAsArray()
speed = numpy.sqrt(u10*u10 + v10*v10)
xx, yy = map.makegrid(v10.shape[1], v10.shape[0], returnxy=True)[2:4]

map.streamplot(xx, yy, u10, v10, color=speed, cmap=plt.cm.autumn, linewidth=0.5*speed)

map.drawcoastlines(color = '0.15')

plt.show()
```



Note: The matplotlib and basemap versions must be quite new to use this function. From Ubuntu 14.04, for instance you need to use *sudo pip install –upgrade matplotlib*. Do then the same with Basemap

2.2.13 text

Plots a text on the map

text(x, y, s, fontdict=None, withdash=False, **kwargs)

- The text method does not belong to Basemap, but directly to matplotlib, so it must be called from the plot or

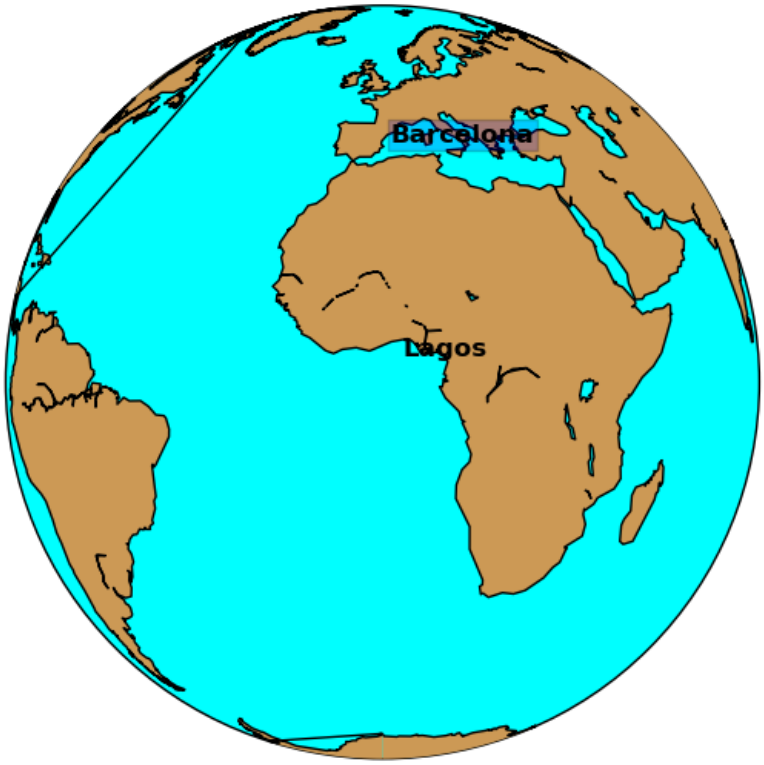
axis instance

- x and y are the coordinates in the map projection. Arrays of coordinates are not accepted, so to add multiple labels, call the method multiple times
- s is the text string
- withdash creates a text with a dash when set to true
- fontdict can be used to group the text properties

The text can have many many options such as:

- fontsize the font size
- fontweight font weight, such as bold
- ha horitzontal align, like center, left or right
- va vertical align, like center, top or bottom
- color the color. This page explains all the color options
- bbox creates a box around the text: bbox=dict(facecolor='red', alpha=0.5)

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='ortho',
              lat_0=0, lon_0=0)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='#cc9955',lake_color='aqua')
map.drawcoastlines()

lon = 3.4
lat = 3.4

x, y = map(lon, lat)

plt.text(x, y, 'Lagos',fontsize=12,fontweight='bold',
                    ha='left',va='bottom',color='k')

lon = 2.1
lat = 41.

x, y = map(lon, lat)

plt.text(x, y, 'Barcelona',fontsize=12,fontweight='bold',
                    ha='left',va='center',color='k',
                    bbox=dict(facecolor='b', alpha=0.2))
plt.show()
```



To draw a text label with an arrow, use *annotate*.

2.3 Background methods

I've called background methods those functions useful for drawing borders, lands, etc. to distinguish them from those aimed to draw user data.

2.3.1 arcgisimage

Downloads and plots an image using the arcgis REST API service

arcgisimage(server='http://server.arcgisonline.com/ArcGIS', service='ESRI_Imagery_World_2D', xpixels=400, ypixels=None, dpi=96, verbose=False, **kwargs)

- server can be used to connect to another server using the same REST API
- service is the layer to draw. To get the full list of available layers, check the API web page,
- xpixels actually sets the zoom of the image. A bigger number will ask a bigger image, so the image will have more detail. So when the zoom is bigger, the xsize must be bigger to maintain the resolution
- ypixels can be used to force the image to have a different number of pixels in the y direction that the ones defined with the aspect ratio. By default, the aspect ratio is maintained, which seems a good value

- dpi is the image resolution at the output device. Changing its value will change the number of pixels, but not the zoom level
- verbose prints the url used to get the remote image. It's interesting for debugging

An important point when using this method is that the projection must be set using the *epsg* argument, unless 4326, or *cyl* in *Basemap notation* is used. To see how to set a projection this way, see the section *Using epsg to set the projection*

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(llcrnrlon=3.75,llcrnrlat=39.75,urcrnrlon=4.35,urcrnrlat=40.15,␣
↪epsg=5520)
#http://server.arcgisonline.com/arcgis/rest/services

map.arcgisimage(service='ESRI_Imagery_World_2D', xpixels = 1500, verbose= True)
plt.show()
```

The example shows Menorca island using the UTM projection at the zone 31N, and can be used with many layers:

The default *ESRI_Imagery_World_2D* layer

The *World_Shaded_Relief* layer

2.3.2 bluemarble

Plots the bluemarble image on the map.

bluemarble(ax=None, scale=None, **kwargs)

- The scale is useful to downgrade the original image resolution to speed up the process. A value of 0.5 will divide the size of the image by 4
- The image is warped to the final projection, so all projections work properly with this method

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(llcrnrlon=-10.5,llcrnrlat=33,urcrnrlon=10.,urcrnrlat=46.,
              resolution='i', projection='cass', lat_0 = 39.5, lon_0 = 0.)

map.bluemarble()

map.drawcoastlines()

plt.show()
```

2.3.3 drawcoastlines

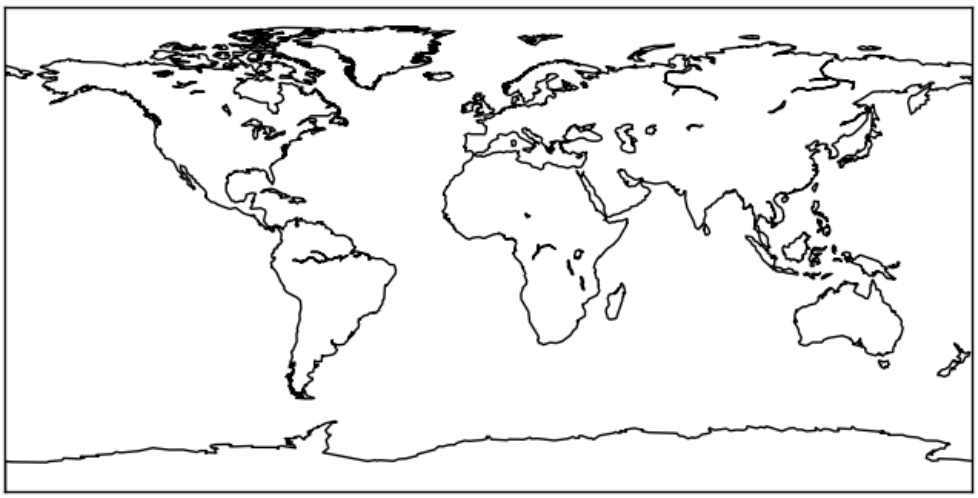
Draws the coastlines.

The function has the following arguments:

drawcoastlines(linewidth=1.0, linestyle='solid', color='k', antialiased=1, ax=None, zorder=None)

- linewidth sets, of course, the line width in pixels
- linestyle sets the line type. By default is solid, but can be dashed, or any matplotlib option
- color is k (black) by default. Follows also matplotlib conventions
- antialiased is true by default
- zorder sets the layer position. By default, the order is set by Basemap

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap()

map.drawcoastlines()

plt.show()
plt.savefig('test.png')
```



2.3.4 drawcounties

Draws the USA counties from the layer included with the library

drawcounties(linewidth=0.1, linestyle='solid', color='k', antialiased=1, facecolor='none', ax=None, zorder=None, drawbounds=False)

- linewidth sets, of course, the line width in pixels
- linestyle sets the line type. By default is solid, but can be dashed, or any matplotlib option
- color is k (black) by default. Follows also matplotlib conventions
- antialiased is true by default
- zorder sets the layer position. By default, the order is set by Basemap

---

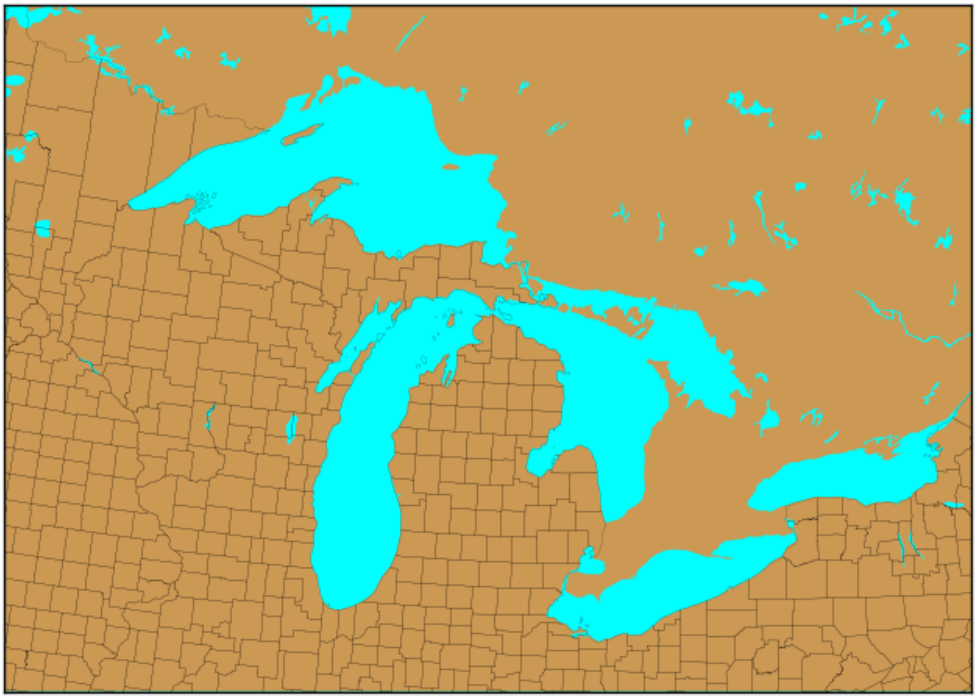
Note: facecolor argument, which is supposed to color the counties, doesn't seem to work at least in some Basemap versions.

Note that:

- The resolution is fix, and doesn't depend on the resolution parameter passed to the class constructor

- The coastline is in another function, and the country coasts are not considered coast, which makes necessary to combine the method with others to get a good map

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(llcrnrlon=-93.,llcrnrlat=40.,urcrnrlon=-75.,urcrnrlat=50.,
              resolution='i', projection='tmerc', lat_0 = 40., lon_0 = -80)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='#cc9955', lake_color='aqua')

map.drawcounties()

plt.show()
```



2.3.5 drawcountries

Draws the country borders from the layer included with the library.

The function has the following arguments:

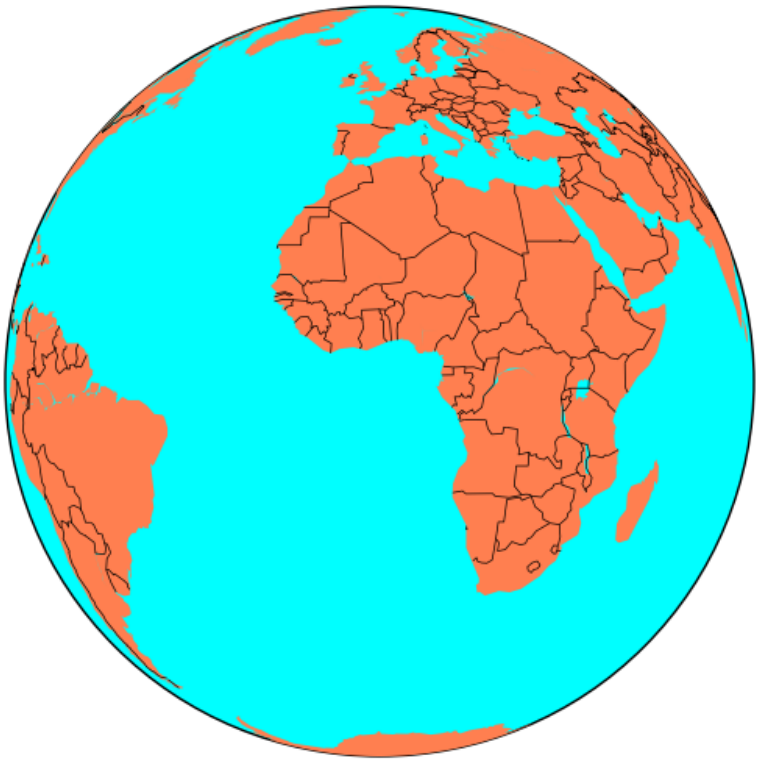
drawcountries(linewidth=1.0, linestyle='solid', color='k', antialiased=1, ax=None, zorder=None)

- linewidth sets, of course, the line width in pixels
- linestyle sets the line type. By default is solid, but can be dashed, or any matplotlib option
- color is k (black) by default. Follows also matplotlib conventions
- antialiased is true by default
- zorder sets the layer position. By default, the order is set by Basemap

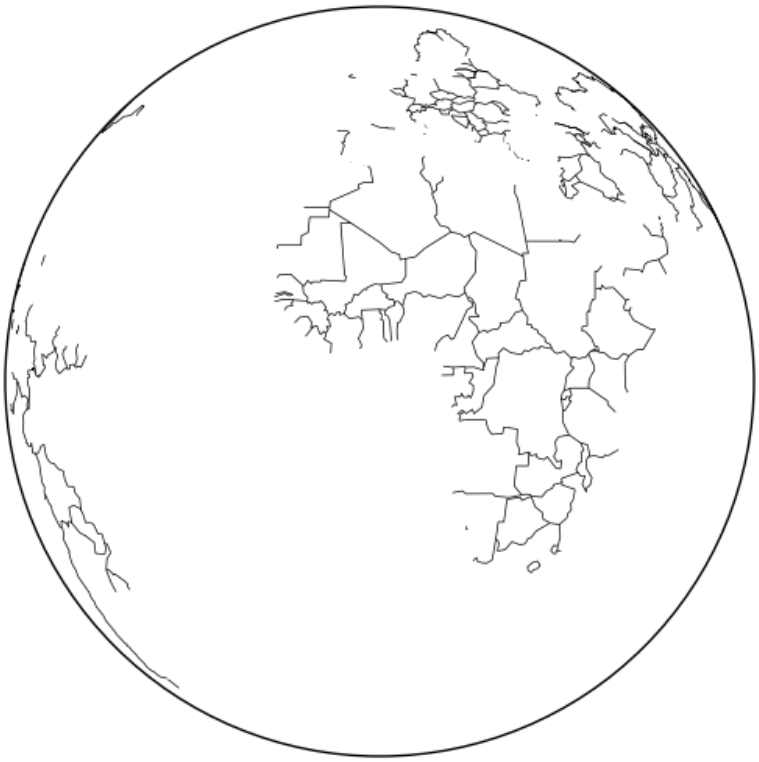
Note that:

- The resolution indicated when creating the *Basemap* instance makes the layer to have a better or coarser resolution
- The coastline is in another function, and the country coasts are not considered coast, which makes necessary to combine the method with others to get a good map

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='ortho',
              lat_0=0, lon_0=0)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')

map.drawcountries()

plt.show()
```



Without drawing the coastline, the result is a bit strange:



2.3.6 drawlsmask

A method that draws at once lakes, land and oceans. Avoids *fillcontinents* and *drawmapboundary*. Besides, it can change the data origin to a custom array of points.

A difference from the other methods is that the zorder can't be set in this method.

drawlsmask(land_color='0.8', ocean_color='w', lsmask=None, lsmask_lons=None, lsmask_lats=None, lakes=True, resolution='l', grid=5, **kwargs)

- land_color sets the color assigned to the land (a gray by default)
- ocean_color sets the colors of the oceans (white by default)
- lsmask An array with alternative data. If None, the default data from the library is taken. The array must contain 0's for oceans and 1's for land
- lsmask_lons the longitudes for the alternative land sea mask
- lsmask_lats the latitudes for the alternative land sea mask
- resolution can change the resolution of the data defined by the Basemap instance
- grid The mask array grid resolution in minutes of arc. Default is 5 minutes

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
```

```
map = Basemap(projection='ortho',
              lat_0=0, lon_0=0)

map.drawlsmask(land_color = "#ddaa66",
               ocean_color="#7777ff",
               resolution = 'l')

plt.show()
```

2.3.7 drawmapboundary

Draws the earth boundary on the map, with optional filling.

drawmapboundary(color='k', linewidth=1.0, fill_color=None, zorder=None, ax=None)

- linewidth sets, of course, the line width in pixels
- color sets the edge color and is k (black) by default. Follows also matplotlib conventions
- fill_color sets the color that fills the globe, and is None by default . Follows also matplotlib conventions
- zorder sets the layer position. By default, the order is set by Basemap

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

plt.figure(0)
map = Basemap(projection='ortho',lon_0=0,lat_0=0,resolution='c')
map.drawmapboundary()

plt.figure(1)
map = Basemap(projection='sinu',lon_0=0,resolution='c')
map.drawmapboundary(fill_color='aqua')

plt.show()
```

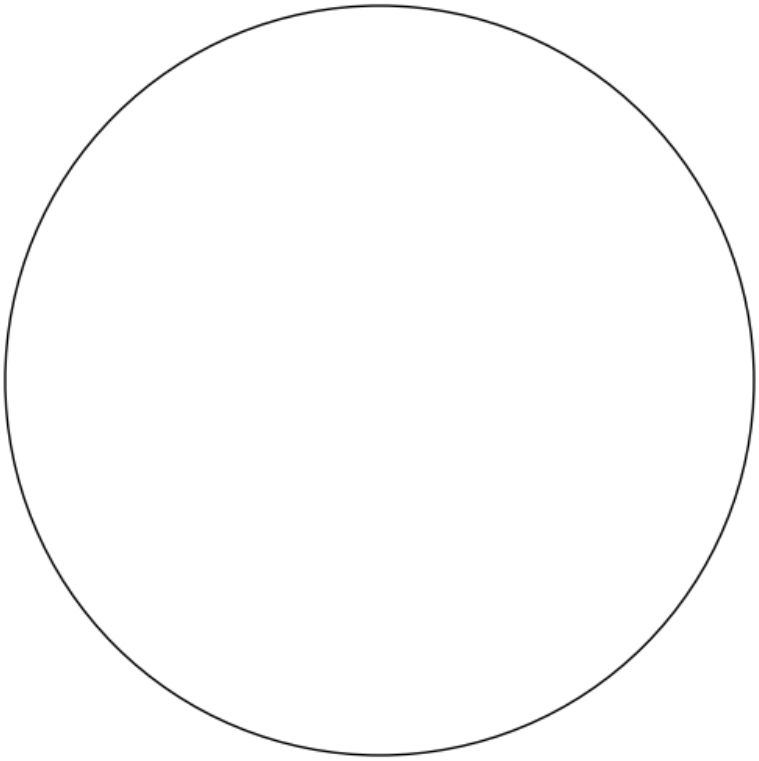


Fig. 2.5: Orthographic projection result

2.3.8 drawmeridians

Draws the meridians on the map

drawmeridians(meridians, color='k', linewidth=1.0, zorder=None, dashes=[1, 1], labels=[0, 0, 0, 0], labelstyle=None, fmt='%g', xoffset=None, yoffset=None, ax=None, latmax=None, **kwargs)

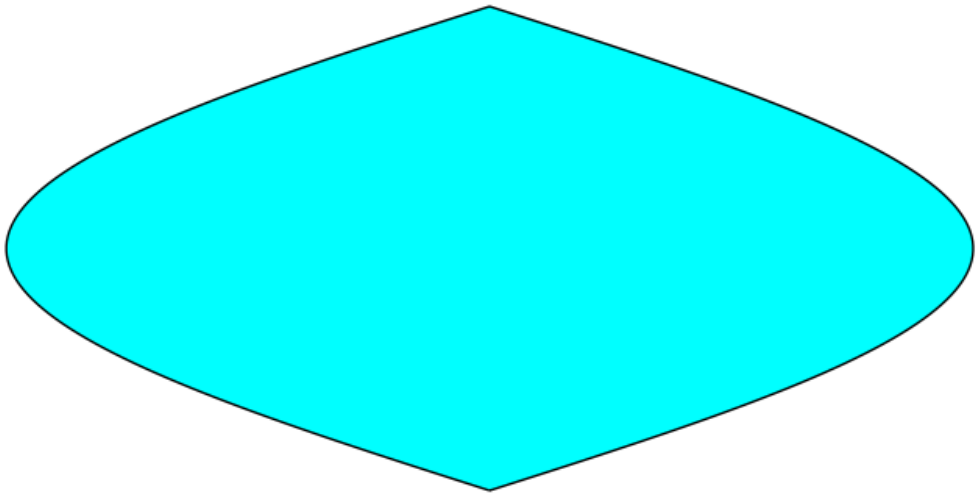
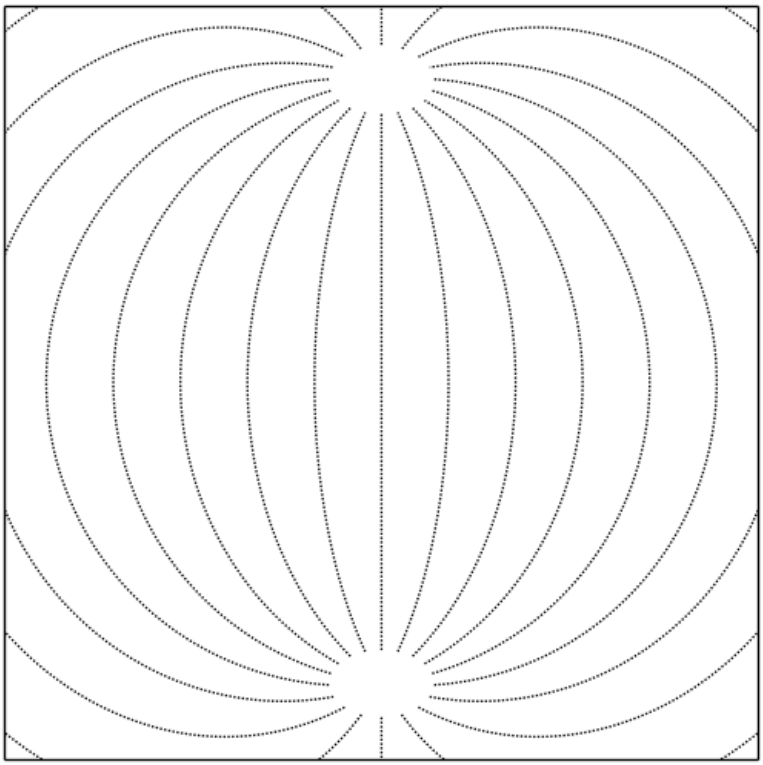


Fig. 2.6: Sinusoidal Projection result

- meridians is a list with the longitudes to plot. This can be created with *range()* if the values are integers. If you need floats, *np.arange()* is a good option
- color sets the color of the line. This page explains all the color options
- linewidth sets, of course, the line width in pixels
- zorder changes the position of the lines, to be able, for instance, to make the land to cover the meridians, or the opposite
- Sets the dashing style. The first element is the number of pixels to draw, and the second, the number of pixels to skip
- labels change the positions where the labels are drawn. Setting the value to 1 makes the labels to be drawn at the selected edge of the map. The four positions are [left, right, top, bottom]

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='aeqd',
              lon_0=0.0, lat_0=0,
              width=25000000, height=25000000)

map.drawmeridians(range(0, 360, 20))

plt.show()
```



This example shows the simplest way to use the function, using the Azimuthal equidistant projection. To see a more complex example, take a look at *drawparallels*

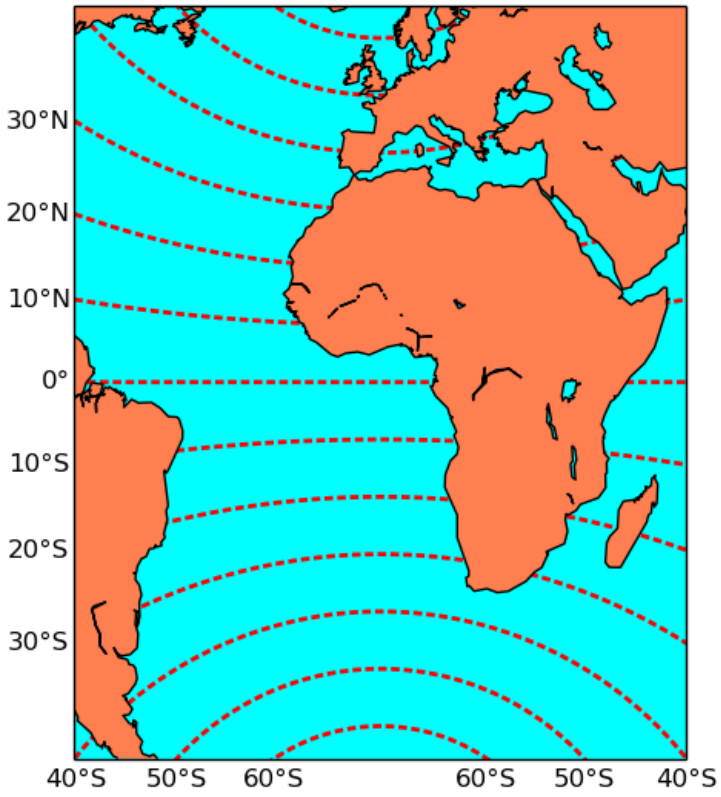
2.3.9 drawparallels

Draws the parallels on the map

drawparallels(circles, color='k', linewidth=1.0, zorder=None, dashes=[1, 1], labels=[0, 0, 0, 0], labelstyle=None, fmt='%g', xoffset=None, yoffset=None, ax=None, latmax=None, **kwargs)

- parallels is a list with the longitudes to plot. This can be created with *range()* if the values are integers. If you need floats, *np.arange()* is a good option
- color sets the color of the line. This page explains all the color options
- linewidth sets, of course, the line width in pixels
- zorder changes the position of the lines, to be able, for instance, to make the land to cover the parallels, or the opposite
- Sets the dashing style. The first element is the number of pixels to draw, and the second, the number of pixels to skip
- labels change the positions where the labels are drawn. Setting the value to 1 makes the labels to be drawn at the selected edge of the map. The four positions are [left, right, top, bottom]

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='poly',
              lon_0=0.0, lat_0=0,
              llcrnrlon=-80.,llcrnrlat=-40,urcrnrlon=80.,urcrnrlat=40.)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()

map.drawparallels(range(-90, 100, 10), linewidth=2, dashes=[4, 2], labels=[1,0,0,1],␣
↪color='r', zorder=0 )
plt.show()
```



The example shows some avance functions, such as labeling or zorder, using the polyconic projection. To see a simpler example, take a look ar *drawmeridians*

2.3.10 drawrivers

Draws the rivers from the layer included with the library.

drawrivers(linewidth=0.5, linestyle='solid', color='k', antialiased=1, ax=None, zorder=None)

- linewidth sets, of course, the line width in pixels
- linestyle sets the line type. By default is solid, but can be dashed, or any matplotlib option
- color is k (black) by default. Follows also matplotlib conventions
- antialiased is true by default
- zorder sets the layer position. By default, the order is set by Basemap

Note that:

- The resolution is fix, and doesn't depend on the resolution parameter passed to the class constructor

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
```

```
map = Basemap(llcrnrlon=-93.,llcrnrlat=40.,urcrnrlon=-75.,urcrnrlat=50.,
              resolution='i', projection='tmerc', lat_0 = 40., lon_0 = -80)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='#ddaa66', lake_color='#0000ff')

map.drawcountries()
map.drawrivers(color='#0000ff')

plt.show()
```

2.3.11 drawstates

Draws the American countries states borders from the layer included with the library. Draws also the Australian states.

drawstates(linewidth=0.5, linestyle='solid', color='k', antialiased=1, ax=None, zorder=None)

- linewidth sets, of course, the line width in pixels
- linestyle sets the line type. By default is solid, but can be dashed, or any matplotlib option
- color is k (black) by default. Follows also matplotlib conventions
- antialiased is true by default

- zorder sets the layer position. By default, the order is set by Basemap

Note that:

- The resolution is fix, and doesn't depend on the resolution parameter passed to the class constructor
- The country border is not drawn, creating a strange effect if the method is not combined with drawcountries

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt


map = Basemap(width=12000000,height=9000000,
              rsphere=(6378137.00,6356752.3142),\
              resolution='l',area_thresh=1000.,projection='lcc',\
              lat_1=45.,lat_2=55,lat_0=50,lon_0=-107.)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='#ddaa66', lake_color='aqua')

map.drawcountries()
map.drawstates(color='0.5')

plt.show()
```

2.3.12 etopo

Plots a relief image called *etopo* taken from the NOAA. The image has a 1" arch resolution, so when zooming in, the results are quite poor.

etopo(ax=None, scale=None, **kwargs)

- The scale is useful to downgrade the original image resolution to speed up the process. A value of 0.5 will divide the size of the image by 4
- The image is warped to the final projection, so all projectinos work properly with this method

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(llcrnrlon=-10.5,llcrnrlat=33,urcrnrlon=10.,urcrnrlat=46.,
              resolution='i', projection='cass', lat_0 = 39.5, lon_0 = 0.)

map.etopo()

map.drawcoastlines()

plt.show()
```

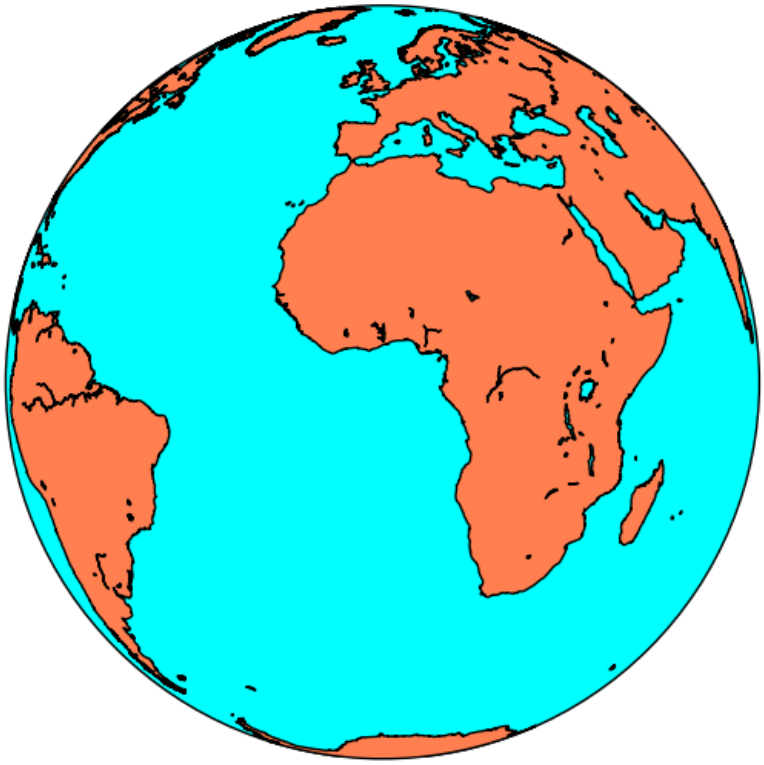
2.3.13 fillcontinents

Draws filled polygons with the continents

fillcontinents(color='0.8', lake_color=None, ax=None, zorder=None, alpha=None)

- color sets the continent color. By default is a gry color. This page explains all the color options
- lake color sets the color of the lakes. By default doesn't draw them, but you may set it to aqua to plot them blue
- alpha is a value from 0 to 1 to set the transparency
- zorder sets the position of the layer related to others. It can be used to hide (or show) a contourf layer, that should be only on the sea, for instance

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='ortho',
              lat_0=0, lon_0=0)

#Fill the globe with a blue color
map.drawmapboundary(fill_color='aqua')
#Fill the continents with the land color
map.fillcontinents(color='coral',lake_color='aqua')

map.drawcoastlines()

plt.show()
```



2.3.14 shadedrelief

Plots a shaded relief image. The origin is the www-shadedrelief.com web page. The original image size is 10800x5400

shadedrelief(ax=None, scale=None, **kwargs)

- The scale is useful to downgrade the original image resolution to speed up the process. A value of 0.5 will divide the size of the image by 4. The original size is quite big, 10800x5400 pixels
- The image is warped to the final projection, so all projections work properly with this method

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(llcrnrlon=-10.5,llcrnrlat=33,urcrnrlon=10.,urcrnrlat=46.,
              resolution='i', projection='cass', lat_0 = 39.5, lon_0 = 0.)

map.shadedrelief()

map.drawcoastlines()

plt.show()
```

2.3.15 warpimage

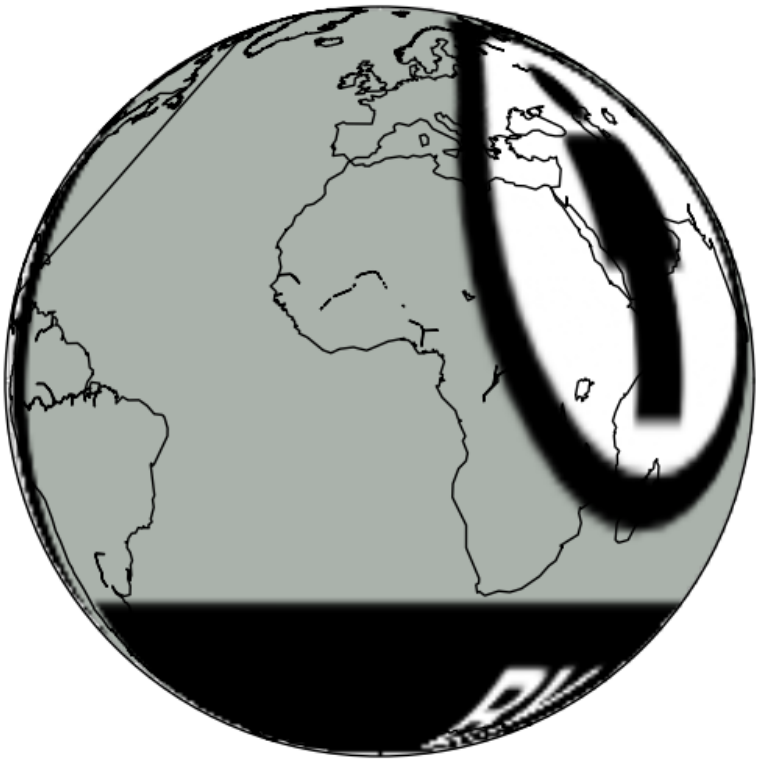
Displays an image as a background.

warpimage(image='bluemarble', scale=None, **kwargs)

- By default, displays the NASA Bluemarble image
- The image must be in latlon projection, so the x size must be double than the y size
- The image must cover the whole world, with the longitude starting at -180

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import Image

map = Basemap(projection='ortho',
              lat_0=0, lon_0=0)


tmpdir = '/tmp'

size = [600, 300]
im = Image.open("../sample_files/by.png")

im2 = im.resize(size, Image.ANTIALIAS)
```

```
im2.save(tmpdir+'/resized.png', "PNG")

map.warpimage(tmpdir+'/resized.png')

map.drawcoastlines()

plt.show()
```



- The image must be resized to fit the proportions. The script won't work if the computer hasn't got the /tmp directory.

2.3.16 wmsimage

Downloads and plots an image, using the WMS protocol

wmsimage(server, xpixels=400, ypixels=None, format='png', verbose=False, **kwargs)

Note: Many arguments aren't documented, making this method a little bit difficult to use

- server can be used to connect to another server using the same REST API
- xpixels actually sets the zoom of the image. A bigger number will ask a bigger image, so the image will have more detail. So when the zoom is bigger, the xsize must be bigger to maintain the resolution

- ypixels can be used to force the image to have a different number of pixels in the y direction that the ones defined with the aspect ratio. By default, the aspect ratio is maintained, which seems a good value
- format sets the image format to ask at the WMS server. Usually, the possibilities are png/gif/jpeg.
- verbose prints the url used to get the remote image. It's interesting for debugging, since prints all the available layers, projections in EPSG codes, and other information

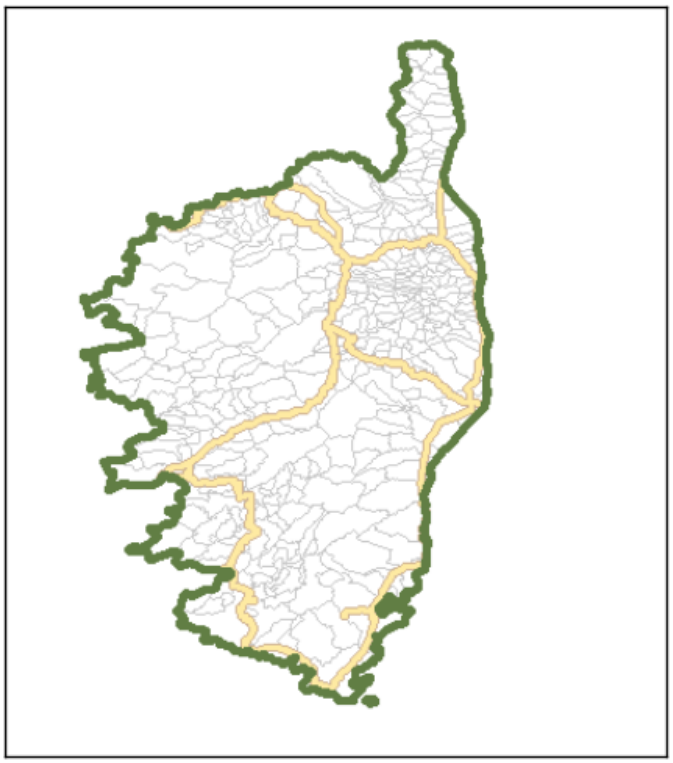
The problem is that using only these parameters won't add the layer properly. There are more mandatory arguments:

- **layers is a list of the WMS layers to use. To get all the possible layers, take a look at the WMS GetCapabilities or, easier, u**
  - When the layer name has a space, the method won't work or, at least, I couldn't make it work. Unfortunately, many services offer layers with spaces in its name
- styles is a list with the styles to ask to the WMS service for the layers. Usually will work without this parameter, since the server has usually default styles
- Other parameters, such as date, elevation or colorscale have the same names and do the same things as in the WMS standard
- An other important point when using this method is that the projection must be set using the *epsg* argument, unless 4326, or *cyl* in *Basemap notation* is used. To see how to set a projection this way, see the section *Using epsg to set the projection*

Note: The method requires OWSLib. To install it, just type *sudo pip install OWSLib*

The Basemap test files shows how to use the method quite well.

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(llcrnrlon=8.35,llcrnrlat=41.225,urcrnrlon=10.01,urcrnrlat=43.108,
              projection='cyl', epsg=4326)

wms_server = "http://www.ga.gov.au/gis/services/topography/Australian_Topography/
↪MapServer/WMSServer"
wms_server = "http://wms.geosignal.fr/metropole?"

map.wmsimage(wms_server, layers=["Communes", "Nationales", "Regions"], verbose=True)
plt.show()
```



The source of the map data is http://www.geosignal.org, which has many layers for France.

2.4 Basemap utility functions

2.4.1 addcyclic

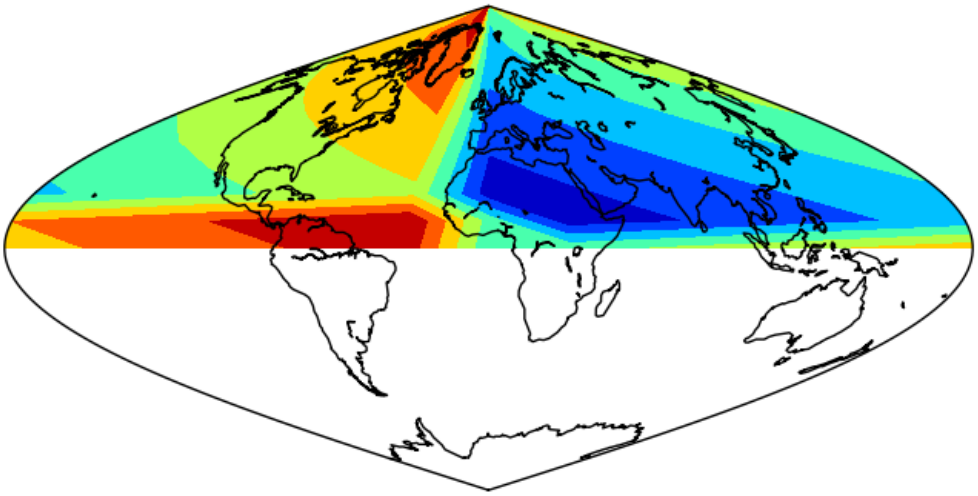
Adds a longitude value, and a columns of values to the data array. Useful to fill the missing data when the data covers the whole longitudes.

mpl_toolkits.basemap.addcyclic(arrin, lonsin)

- Note that it's not a basemap method, but a separate function
- arrin is the input data. A column will be added to fill the gap between the -180 and 180 longitudes
- lonsin is a single dimension array containing the longitudes

```
from mpl_toolkits.basemap import Basemap
from mpl_toolkits.basemap import addcyclic
import matplotlib.pyplot as plt
import numpy as np

map = Basemap(projection='sinu',
              lat_0=0, lon_0=0)
```

```
lons = np.arange(30, 390, 30)
lats = np.arange(0, 100, 10)

data = np.indices((lats.shape[0], lons.shape[0]))
data = data[0] + data[1]

data , lons = addcyclic(data, lons)

lons, data = map.shiftdata(lons, datain = data, lon_0=0)

llons, llats = np.meshgrid(lons, lats)

x, y = map(llons, llats)

map.contourf(x, y, data)

map.drawcoastlines()
plt.show()
```



- The example is the same as in *shiftdata*. The difference is that the white band disappears.
- Note the order of the output variables: longitudes and data
- Note that the longitudes are a one dimension array

2.4.2 colorbars

Draws the color legend at one of the edges of the map. The use is almost the same as in matplotlib colorbar

colorbar(mappable=None, location='right', size='5%', pad='2%', fig=None, ax=None, **kwargs)

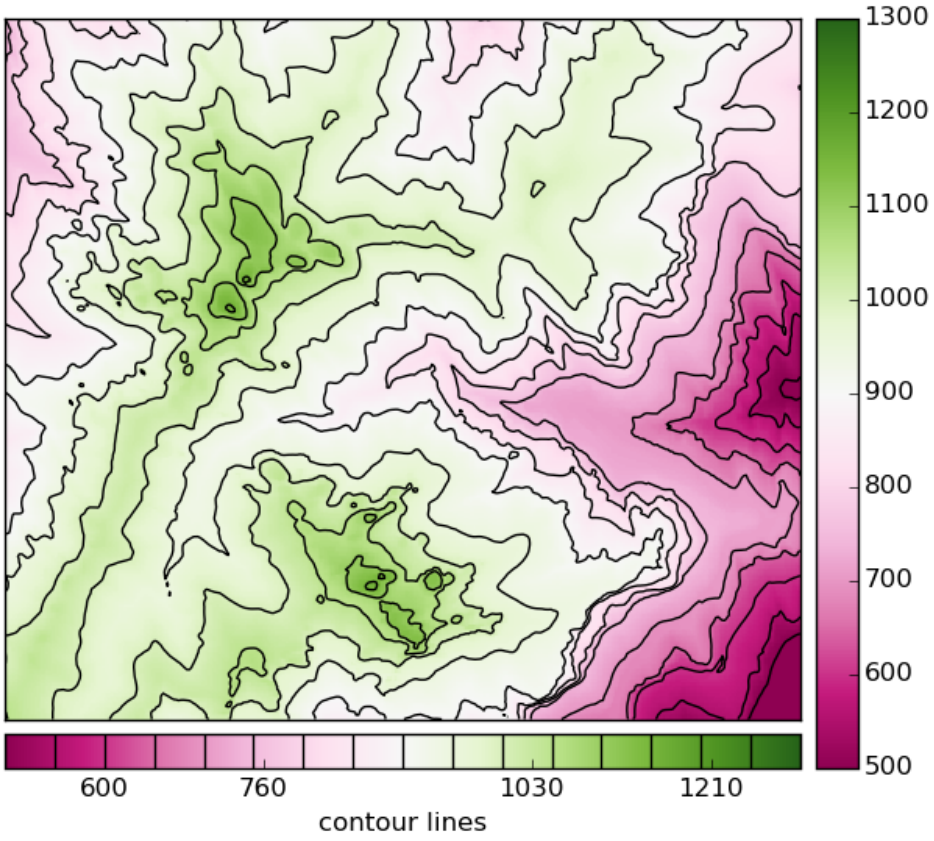
- mappable is the most important argument. Is the field on the map to be explained with the colorscale. It can be a contourf, a pcolormesh, contour, etc. If the value is None, the last drawn field is represented
- location sets the border of the map where the color scale is drawn. Can be top, right, left or bottom
- size sets the width of the color bar, in % of the parent axis
- pad sets the separation between the axes and the color bar, also in % of the parent axis
- fig is the figure the colorbar is associated with

Most of the matplotlib.colorbar arguments will work too, such as label

The colorbar method returns an object, which has some interesting methods too:

- add_lines adds to the color bar, the lines from an other field (look at the example to see how does it work)
- set_ticks changes the positions of the ticks and labels on the color bar

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from osgeo import gdal
from numpy import linspace
from numpy import meshgrid

map = Basemap(projection='tmerc',
              lat_0=0, lon_0=3,
              llcrnrlon=1.819757266426611,
              llcrnrlat=41.583851612359275,
              urcrnrlon=1.841589961763497,
              urcrnrlat=41.598674173123)

ds = gdal.Open("../sample_files/dem.tiff")
data = ds.ReadAsArray()

x = linspace(0, map.urcrnrx, data.shape[1])
y = linspace(0, map.urcrnry, data.shape[0])

xx, yy = meshgrid(x, y)

cmap = plt.get_cmap('PiYG')

colormesh = map.pcolormesh(xx, yy, data, vmin = 500, vmax = 1300, cmap=cmap)
contour = map.contour(xx, yy, data, range(500, 1350, 50), colors = 'k', linestyles =
↪'solid')

map.colorbar(colormesh)
cb = map.colorbar(colormesh, location='bottom', label="contour lines")

cb.add_lines(contour)
cb.set_ticks([600, 760, 1030, 1210])

plt.show()
```



- A colormesh and a contour fields are plotted, to be able to use some advanced colorbar attributes
- The first colorbar (line 27), shows the default use of the colorbar. The pcolormesh is passed as the argument, to force the method to draw this one instead of the contour field
- **The second colorbar uses some more arguments**
  - The position is changed to bottom
  - A label is set
  - The method add_lines is used with the contour field, so the colorbar shows the pcolormesh and contour field legends at once
  - The ticks are set at random positions, to show how to change them

To see an example with logarithmic scales, take a look at the *hexbin* example

2.4.3 drawmapscale

Draws the scale of the map at the indicated position

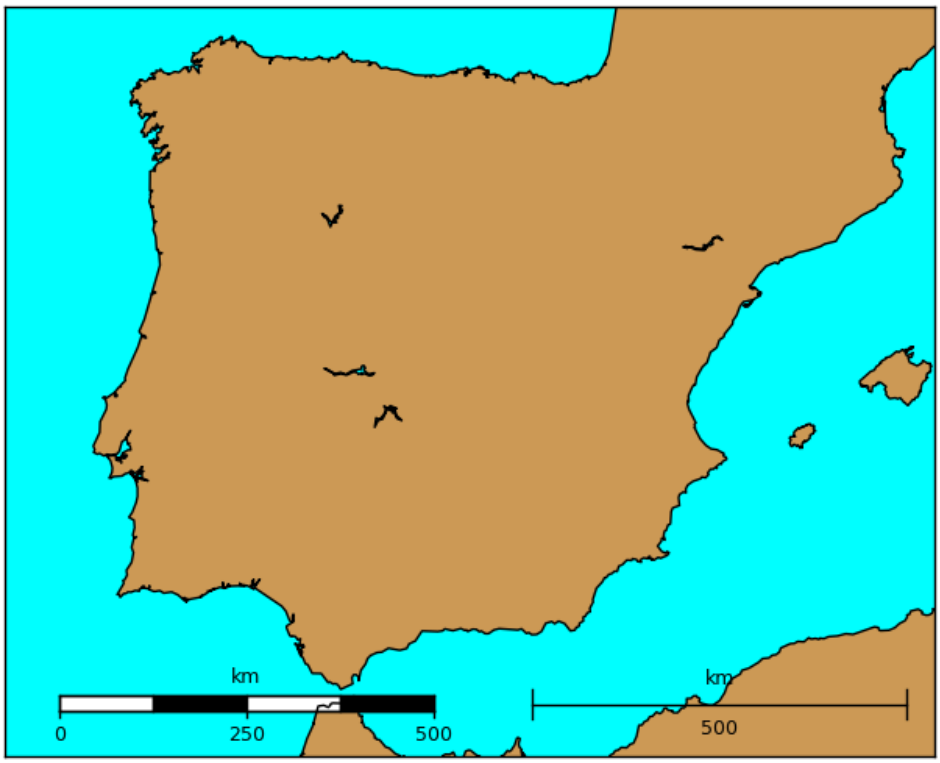
drawmapscale(lon, lat, lon0, lat0, length, barstyle='simple', units='km', fontsize=9, yoffset=None, labelstyle='simple', fontcolor='k', fillcolor1='w', fillcolor2='k', ax=None, format='%d', zorder=None)

- lon and lat indicate the position of the scale on the map. The fact that the geographical coordinates must be used is a problem, since there's no possibility to put the scale outside the map

- lon0 lat0 indicate the location where the scale is calculated
- length is the number of kilometers to represent in the scale
- barstyle can be 'simple' or 'fancy', and changes the scalebar style. Both styles are represented in the example
- units indicates the units to represent in the scale, kilometers by default
- fontsize changes the font size of the units drawn on the scale bar
- fontcolor sets the font color of the units drawn on the scale bar
- yoffset controls the height of the scalebar. By default is 0.02 times the height of the map. It doesn't seem to work properly in all the versions
- fillcolor1 and fillcolor2 set the colors of the bar at the scale bar when the style is 'fancy'
- format sets the number format on the scale bar

Note: The projection *cyl* (lat/lon), which is the default, can't use this method. More informatino here.

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(llcrnrlon=-10.5,llcrnrlat=35,urcrnrlon=4.,urcrnrlat=44.,
              resolution='i', projection='tmerc', lat_0 = 39.5, lon_0 = -3.25)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='#cc9955',lake_color='aqua')
map.drawcoastlines()

map.drawmapscale(-7., 35.8, -3.25, 39.5, 500, barstyle='fancy')

map.drawmapscale(-0., 35.8, -3.25, 39.5, 500, fontsize = 14)

plt.show()
```



2.4.4 gcpoints

Calculates n points along a great circle given two coordinates

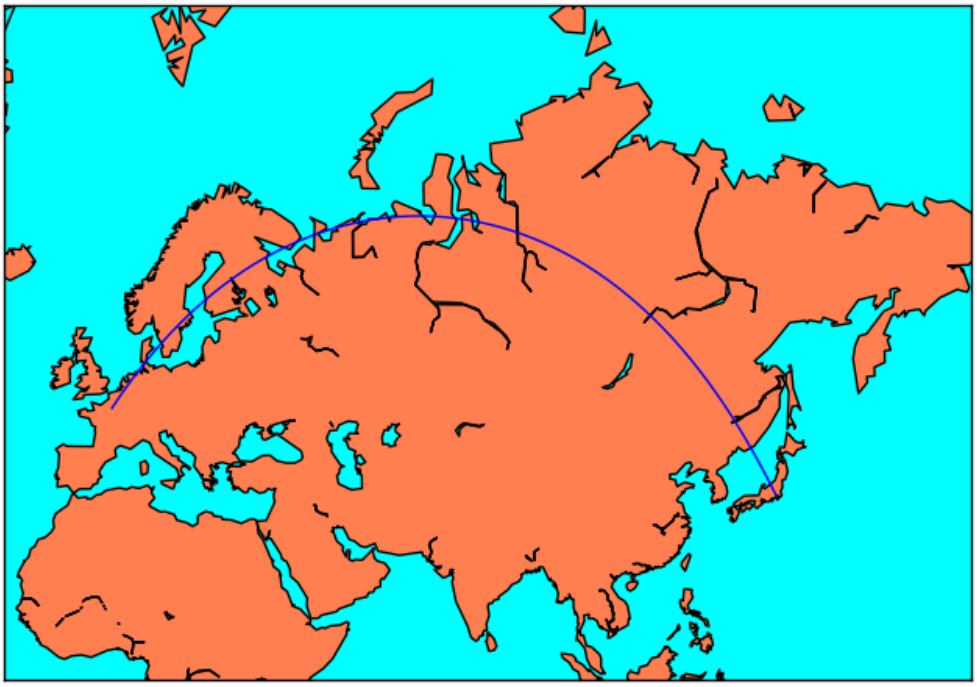
gcpoints(lon1, lat1, lon2, lat2, npoints)

- lon1 and lat1 are the geographical coordinates of the initial point
- lon2 and lat2 are the geographical coordinates of the final point
- npoints is the number of points to calculate

Note: To draw a great circle, the *greatcircle* function it may be easier to use. This function is useful to get the points values, or draw cases when greatcircle fails because of edges problems

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='merc',
              lat_0=0, lon_0=0,
              llcrnrlon=-20.,llcrnrlat=0.,urcrnrlon=180.,urcrnrlat=80.)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
```

```
map.drawcoastlines()

x, y = map.gcpoints(2.3, 48.9, 139.7, 35.6, 300)

print x, y

map.plot(x, y)
plt.show()
```



2.4.5 greatcircle

A great circle is the maximum circle that can be drawn that passes through two points in a sphere (excepting when the points are the antipodes)

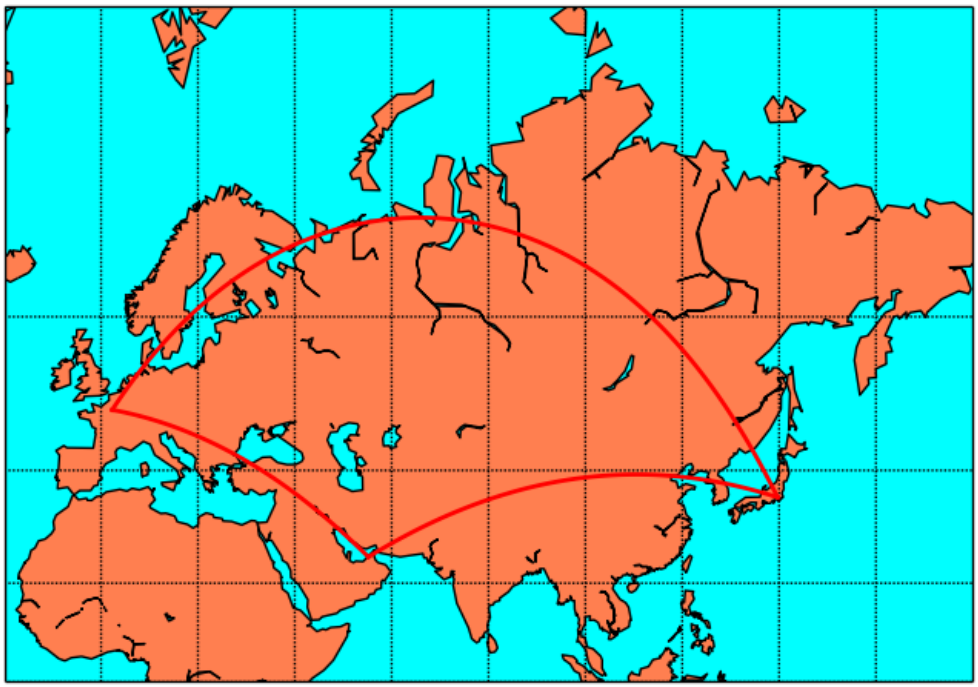
drawgreatcircle(lon1, lat1, lon2, lat2, del_s=100.0, **kwargs)

- lon1 and lat1 are the longitude and latitude of the starting point
- lon2 and lat2 are the longitude and latitude of the ending point
- del_s is the number of kilometers that separate each point of the great circle. Defaults to 100
- linewidth argument sets the width of the line
- color sets the color of the line. This page explains all the color options

Note: If the circle gets cut by the edges of the map, i.e. starts at longitude -179 and ends at 179, the method can't handle it properly

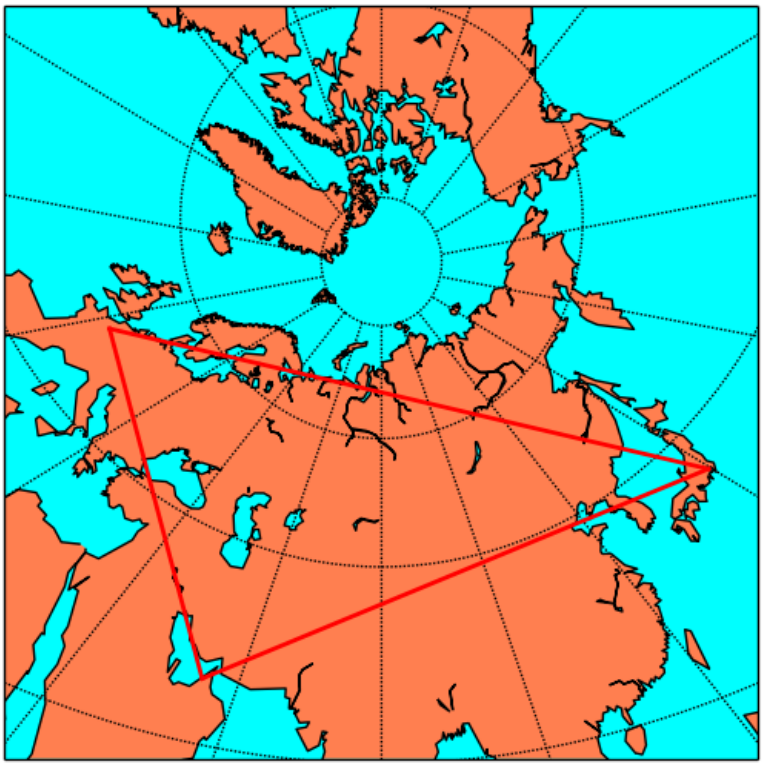
```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='merc',
              lat_0=0, lon_0=0,
              llcrnrlon=-20.,llcrnrlat=0.,urcrnrlon=180.,urcrnrlat=80.)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()

map.drawparallels(range(0, 90, 20))
map.drawmeridians(range(0, 180, 20))

#Paris-Tokyo
map.drawgreatcircle(2.3, 48.9, 139.7, 35.6,linewidth=2,color='r')
#Tokyo-Dubai
map.drawgreatcircle(139.7, 35.6, 55.2, 25.,linewidth=2,color='r')
#Dubai-Paris
map.drawgreatcircle(55.2, 25., 2.3, 48.9,linewidth=2,color='r')

plt.show()
```

When using Mercator projection, the meridians and parallels are straight lines, but the great circles usually are not



```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='gnom',
                                width=15.e6,height=15.e6,
              lat_0=70, lon_0=80)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()

map.drawparallels(range(0, 90, 20))
map.drawmeridians(range(0, 360, 20))

#Paris-Tokyo
map.drawgreatcircle(2.3, 48.9, 139.7, 35.6,linewidth=2,color='r')
#Tokyo-Dubai
map.drawgreatcircle(139.7, 35.6, 55.2, 25.,linewidth=2,color='r')
#Dubai-Paris
map.drawgreatcircle(55.2, 25., 2.3, 48.9,linewidth=2,color='r')

plt.show()
```

The gnomonic projection makes the great circles to be a straight line in any direction:



2.4.6 interp

Interpolates a grid to another grid, with different sizes.

Useful to create smoother plots or to have more elements when using barbs or quiver. Also useful when using *maskoceans*.

interp(datain, xin, yin, xout, yout, checkbounds=False, masked=False, order=1)

- This function is not a method of the basemap instance, but a separate one in the basemap module
- datain is the data array to interpolate. It has to be a 2d numpy array
- **xin and yin are the coordinates of the input data array, in one dimension each.**
  - This point is important, since implies that the input grid has to be regular (so no lon-lat grid in a different projection)
  - y has to be in increasing order
- xout and yout are the coordinates of the output data array. They have to be 2d numpy arrays.
- checkbounds if set to True, the xin and yin values are checked to be in the bounds of xout and yout. If False, and there are values outside the bounds, the output data array values will be clipped to the boundary values
- masked makes the points outside the new grid to be masked if is set to True, or an arbitrary value if given
- **order sets the interpolation method:**

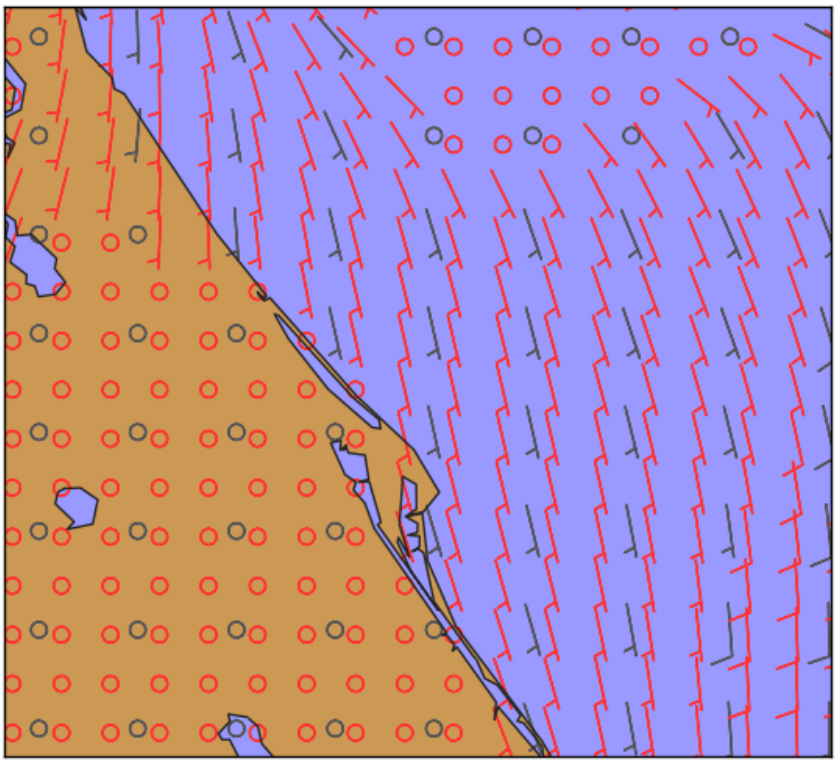
  - 0 uses the nearest neighbor method
  - 1 uses a bilinear interpolation
  - 3 uses a cubic spline, and requires scipy.ndimage to be installed

```
from mpl_toolkits.basemap import Basemap
from mpl_toolkits.basemap import interp
import matplotlib.pyplot as plt
from osgeo import gdal
import numpy as np


map = Basemap(llcrnrlon=-82., llcrnrlat=28., urcrnrlon=-79., urcrnrlat=29.5,
              projection='lcc', lat_1=30., lat_2=60., lat_0=34.83158, lon_0=-98.,
              resolution='i')

ds = gdal.Open("../sample_files/wrf.tiff")
lons = ds.GetRasterBand(4).ReadAsArray()
lats = ds.GetRasterBand(5).ReadAsArray()
u10 = ds.GetRasterBand(1).ReadAsArray()
v10 = ds.GetRasterBand(2).ReadAsArray()

x, y = map(lons, lats)

x2 = np.linspace(x[0][0],x[0][-1],x.shape[1]*2)
y2 = np.linspace(y[0][0],y[-1][0],y.shape[0]*2)

x2, y2 = np.meshgrid(x2, y2)

u10_2 = interp(u10,  x[0], np.flipud(y[:, 0]), x2, np.flipud(y2),order=1)
v10_2 = interp(v10,  x[0], np.flipud(y[:, 0]), x2, np.flipud(y2),order=1)

map.drawmapboundary(fill_color='#9999FF')
map.fillcontinents(color='#cc9955', lake_color='#9999FF', zorder = 0)
map.drawcoastlines(color = '0.15')

map.barbs(x, y, u10, v10,
    pivot='middle', barbcolor='#555555')

map.barbs(x2, y2, u10_2, v10_2,
    pivot='middle', barbcolor='#FF3333')

plt.show()
```

- The example is taken from the *barbs* method example, but zoomin in a lot so the number of barbs is too low, and gives a reason to interpolate
- The positions of the wind barbs (x, y) are calculated using the basemap instance
- **The new grid is created, using linspace**
  - Since the grid is in the map projection, the x and y 2d array can be converted to a 1d array, since the values are the same for all the lines or columns
  - The new grid is created multiplying the number of elements by 2, with the same bounding values, using linspace
  - The new grid has to be in 2d arrays, which is done with the meshgrid method
- **interp can be now called**

  - Since x and y are two dimension arrays, only a single column is passed for x, and a single column for y
  - y axis is not in increasing order (coordinates gofrom north to south), so they have to be reversed using the numpy.flipud method. The output has to be reversed again to have the data properly ordered

- Once the new grid is created, the barbs can be drawn in the new resolution



In gray, the original points, and in red, the interpolated ones

2.4.7 is_land

Returns *True* if the indicated point is on the land, and *False* if on an ocean or lake

is_land(xpt, ypt)

- **xpt and ypt are the point coordinates where the land/water has to be calculated.**
  - The coordinates must be in the map coordinates
  - The resolution indicated in the Basemap constructor must not be None
  - The indicated resolution polygon will be used to calculated if the point is on a land zone, so the results change depending on that
- No arrays of coordinates are allowed, so the query must be done point by point

- There is an alternative way to calculate this for many points using the landpolygons attribute and the matplotlib PATH.contains_points method

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='aeqd', lon_0 = 10, lat_0 = 50, resolution='h')

x, y = map(0, 0)

print map.is_land(x, y)
```

The alternative way, which accepts multiple points and, in fact could be used with any polygon get from a shapefile (See *Filling polygons*)

```
#Idea taken from this post at StackOverflow: http://stackoverflow.com/questions/
↪13796315/plot-only-on-continent-in-matplotlib/13811775#13811775
#I've re-written it to work with modern matplotlib versions

from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from matplotlib.path import Path
import numpy as np

map = Basemap(projection='aeqd', lon_0 = 10, lat_0 = 50, resolution='h')

lons = [0., 0., 16., 76.]
lats = [0., 41., 19., 51.]

x, y = map(lons, lats)
locations = np.c_[x, y]

polygons = [Path(p.boundary) for p in map.landpolygons]

result = np.zeros(len(locations), dtype=bool)

for polygon in polygons:
    result += np.array(polygon.contains_points(locations))

print result
```

- locations is a numpy array containing numpy arrays with the projected points
- The PATH objects are calculated for each of the polygons
- For each PATH, all the points are evaluated using contains_points. The result is casted into a numpy array so can be added with the previous evaluations. If one of the polygons contains the point, the result element will be true

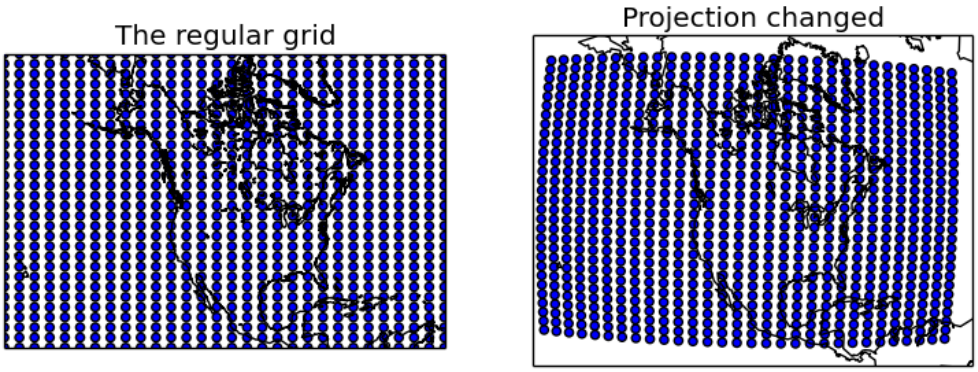
2.4.8 makegrid

makegrid method creates an arbitrary grid of equally spaced points in the map projection. Used to get the longitudes and latitudes that would form an an equally spaced grid using the map projection.

makegrid(nx, ny, returnxy=False)

- nx and n define the size of the output grid
- If returnxy is set to True, the positions of the grid in the map projection are also returned
- returns the longitudes and latitudes 2D numpy arrays, and the x and y in the map coordinates arrays if returnxy is set to True

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np

fig=plt.figure(figsize=(9, 3))

map = Basemap(width=12000000,height=8000000,
            resolution='l',projection='stere',
            lat_ts=50,lat_0=50,lon_0=-107.)

lons, lats, x, y = map.makegrid(30, 30, returnxy=True)

ax = fig.add_subplot(121)
ax.set_title('The regular grid')
map.scatter(x, y, marker='o')
map.drawcoastlines()


ax = fig.add_subplot(122)
ax.set_title('Projection changed')

map = Basemap(width=12000000,height=9000000,projection='aeqd',
            lat_0=50.,lon_0=-105.)
x, y = map(lons, lats)
map.scatter(x, y, marker='o')
map.drawcoastlines()

plt.show()
```



- Two maps are created, one using the same projection as in the regular grid, and the other with another projection to show how to use the created longitudes and latitudes matrices
- **makegrid is used with the returnxy set to True**
  - The first map uses the x and y matrices directly, since is in the same projection. As expected, the created points form a regular grid of 30x30
  - The second map uses the lons and lats matrices, re-projecting them into the new projection using the

basemap instance. The scatter shows that in the new projection, the grid is not regular

2.4.9 maskoceans

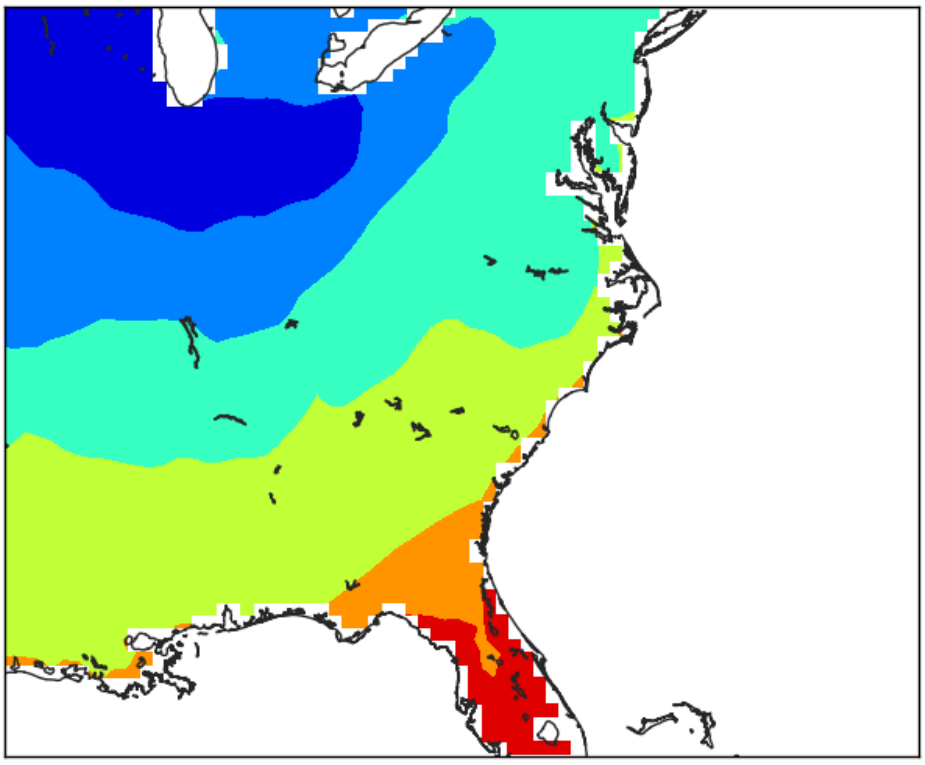
Takes a data array and masks the values at the points in the oceans and lakes

mpl_toolkits.basemap.maskoceans(lonsin, latsin, datain, inlands=True, resolution='l', grid=5)

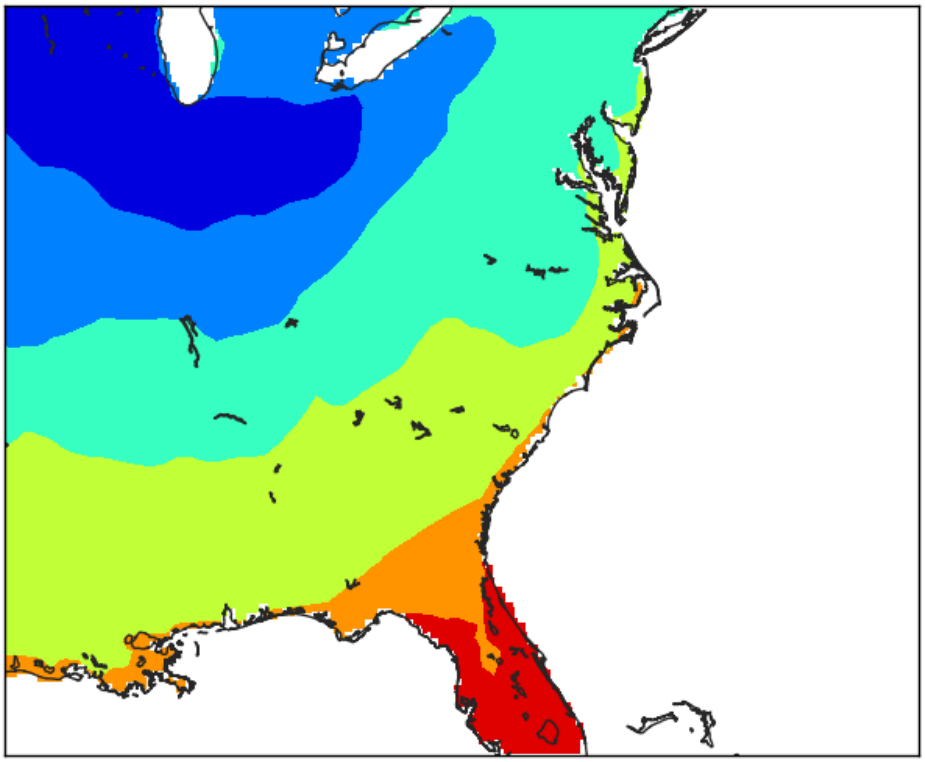
- This function is not a method of the basemap instance, but a separate one in the basemap module
- lonsin and latsin are the location of the points in 2 dimensions arrays. Only latlon projections can be used with this method
- datain is the array containing the values to mask with the oceans
- inland sets if the lakes have to be masked too (true by default)
- resolution selects the resolution of the land-sea boundaries to use then masking the arrays, 'l' by default. The one defined by the basemap instance is not used
- grid sets, in arch minutes, the resolution of the mask grid. 10, 5, 2.5 and 1.25 values are available
- The output array has the same dimensions as the input data array
- See the examples to understand the difference between the resolution and grid arguments

```
from mpl_toolkits.basemap import Basemap
from mpl_toolkits.basemap import maskoceans
from mpl_toolkits.basemap import interp
import matplotlib.pyplot as plt
from osgeo import gdal
import numpy as np


map = Basemap(llcrnrlon=-93.7, llcrnrlat=28., urcrnrlon=-66.1, urcrnrlat=39.5,
              projection='lcc', lat_1=30., lat_2=60., lat_0=34.83158, lon_0=-98.,
              resolution="h")

ds = gdal.Open("../sample_files/wrf.tiff")
lons = ds.GetRasterBand(4).ReadAsArray()
lats = ds.GetRasterBand(5).ReadAsArray()
data = ds.GetRasterBand(3).ReadAsArray()

x, y = map(lons, lats)

plt.figure(0)

mdata = maskoceans(lons, lats, data)

map.drawcoastlines(color = '0.15')
map.contourf(x, y, mdata)

plt.figure(1)

x2 = np.linspace(x[0][0],x[0][-1],x.shape[1]*5)
y2 = np.linspace(y[0][0],y[-1][0],y.shape[0]*5)

x2, y2 = np.meshgrid(x2, y2)

data2 = interp(data,  x[0], np.flipud(y[:, 0]), x2, np.flipud(y2),order=1)
```

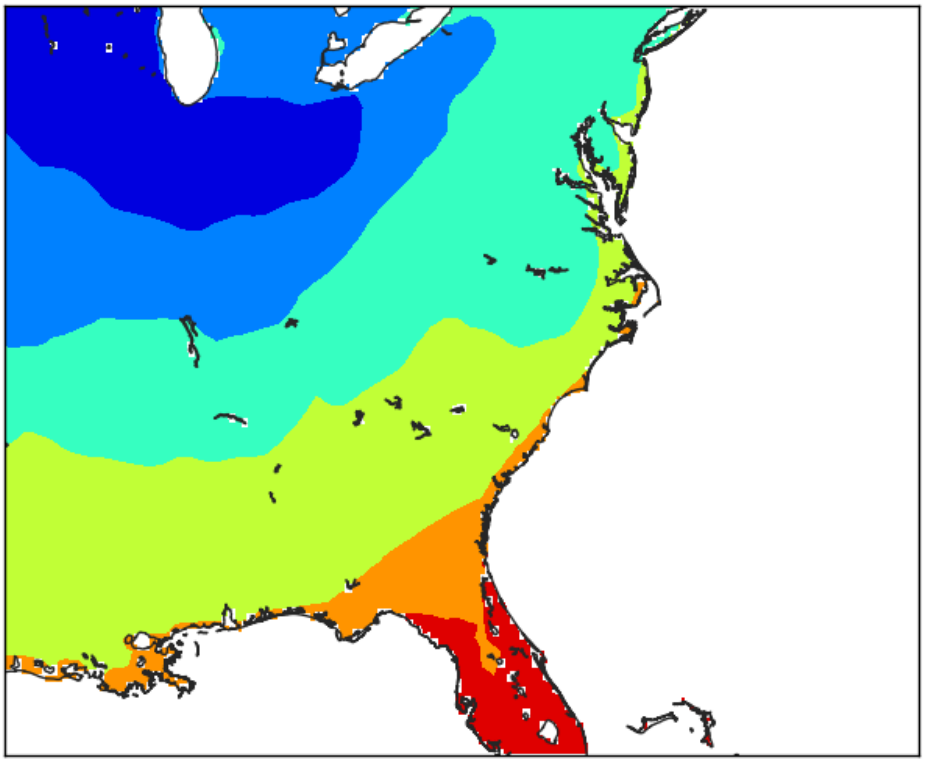
```
35
36 lons2, lats2 = map(x2, y2, inverse=True)
37 mdata = maskoceans(lons2, lats2, data2, resolution = 'c', grid = 10, inlands=True)
38
39 map.drawcoastlines(color = '0.15')
40 map.contourf(x2, y2, mdata)
41
42 plt.figure(2)
43
44 mdata = maskoceans(lons2, lats2, data2, resolution = 'h', grid = 10, inlands=True)
45
46 map.drawcoastlines(color = '0.15')
47 map.contourf(x2, y2, mdata)
48
49 plt.figure(3)
50
51 mdata = maskoceans(lons2, lats2, data2, resolution = 'h', grid = 1.25, inlands=True)
52
53 map.drawcoastlines(color = '0.15')
54 map.contourf(x2, y2, mdata)
55
56 plt.show()
```



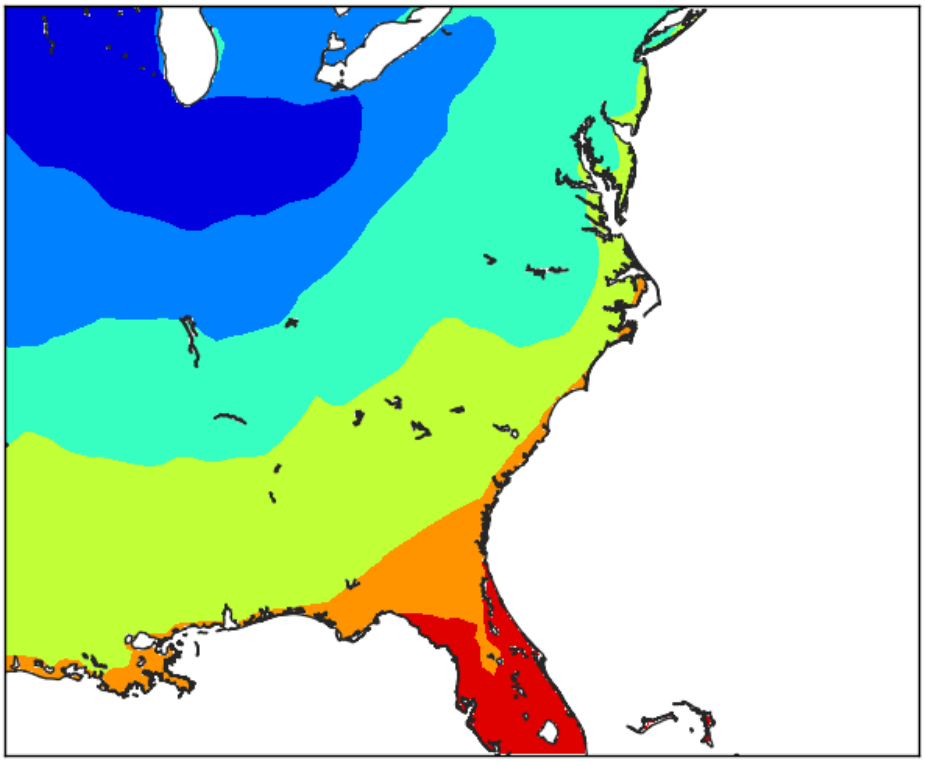
The first example (line 22), creates the mask directly. The result is coarse, due to the low resolution of the input data, not because of the maskoceans arguments



- The second example creates a finer grid (lines 29 to 36) to avoid the effect of the big pixels due to the data. Look at the *interp* for details
- The maskoceans function, however, is called with the lowest resolution in both grid and resolution arguments.
- Note that the lakes at Florida are not masked, because the resolution is low, and that the Florida coast still shows the pixels, because of the big grid value.



In this case, the resolution has been set to 'h', the maximum. The lakes are now masked, but the Florida coast still shows the pixel size used to create the mask.



Finally, when the grid is set at the finer resolution, the Florida coast smoothly matches the coast line.

2.4.10 nightshade

Draws the regions of the map which are dark at the specified date

nightshade(date, color='k', delta=0.25, alpha=0.5, ax=None, zorder=2)

- date is a datetime.datetime object
- color is the color of the drawn shadow
- delta is the resolution to which the shadow zone is calculated. By default is 0.25, and small values fail easily
- alpha is the transparency value
- zorder can change the layer vertical position

The example shows the current dark zones in the van der Grinten Projection:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from datetime import datetime

map = Basemap(projection='vandg',lon_0=0,resolution='c')

map.drawmapboundary(fill_color="#7777ff")
```

```
map.fillcontinents(color="#ddaa66",lake_color="#7777ff")

map.drawcoastlines()

map.nightshade(datetime.now(), delta=0.2)

plt.show()
```

2.4.11 rotate_vector

Given two matrices of the east-west and north-south components of a vectorial field, and the longitude and latitude of the points, rotates the vectors so they represent the direction properly on the map projection

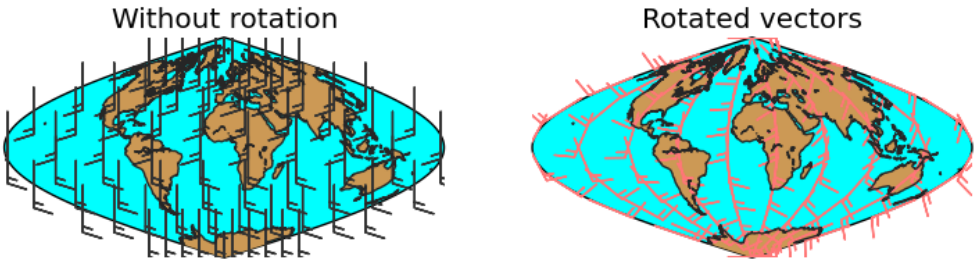
Some functions, such as barbs, quiver or streamplot, that use vectorial data, asks the vector components to be in the map coordinates i.e. u is from left to right, v from up do down. If the available data is in geographical coordinates i.e. west-east and north-south, these coordinates have to be rotated or the vector direction won't be plot properly. This is the aim of the rotate_vector method.

The method *transform_scalar* does the same function, but changing the grid size at the same time (interpolating the points)

rotate_vector(uin, vin, lons, lats, returnxy=False)

- uin and vin are the input data matrices. The directions are the geographical, so the u component is west-east and the v component, north-south
- lons, lats are 2D numpy arrays with the positions of the uin an vin matrices, in geographical coordinates
- returnxy makes the method to return the lons and lats matrices reprojected to the map coordinates. Just as calling the basemap instance

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np


fig=plt.figure(figsize=(9, 3))


map = Basemap(projection='sinu',
              lat_0=0, lon_0=0)

lons = np.linspace(-180, 180, 10)
lats = np.linspace(-90, 90, 10)

lons, lats = np.meshgrid(lons, lats)

v10 = np.ones((lons.shape)) * 15
u10 = np.zeros((lons.shape))

u10_rot, v10_rot, x, y = map.rotate_vector(u10, v10, lons, lats, returnxy=True)

ax = fig.add_subplot(121)
ax.set_title('Without rotation')

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='#cc9955', lake_color='aqua', zorder = 0)
map.drawcoastlines(color = '0.15')


map.barbs(x, y, u10, v10,
    pivot='middle', barbcolor='#333333')

ax = fig.add_subplot(122)
ax.set_title('Rotated vectors')

map.drawmapboundary(fill_color='9999FF')
map.fillcontinents(color='#ddaa66', lake_color='9999FF', zorder = 0)
map.drawcoastlines(color = '0.15')

map.barbs(x, y, u10_rot, v10_rot,
    pivot='middle', barbcolor='#ff7777')

plt.show()
```



- lons and lats are created in an equal spaced grid covering all the globe, using linspace
- v10 and u10 are created so they represent a south to north wind (v10 = 10, u10 = 0)
- The rotation is done, using the created matrices, and calculating the positions of the locatinos in the map projection (returnxy = True)
- **Two maps are drawn, creating the subplots with add_subplot**
  - The first draws the barbs without rotating them. Even though they are supposed to be south-north, the basemap instance takes them as if they were in the map projection, so the barbs go from the bottom to the top, which is not south to north
  - The second shows the result of the rotation

2.4.12 set_axes_limits

This method is usually called internally, and changes the matplotlib axes to the shape of the projection.

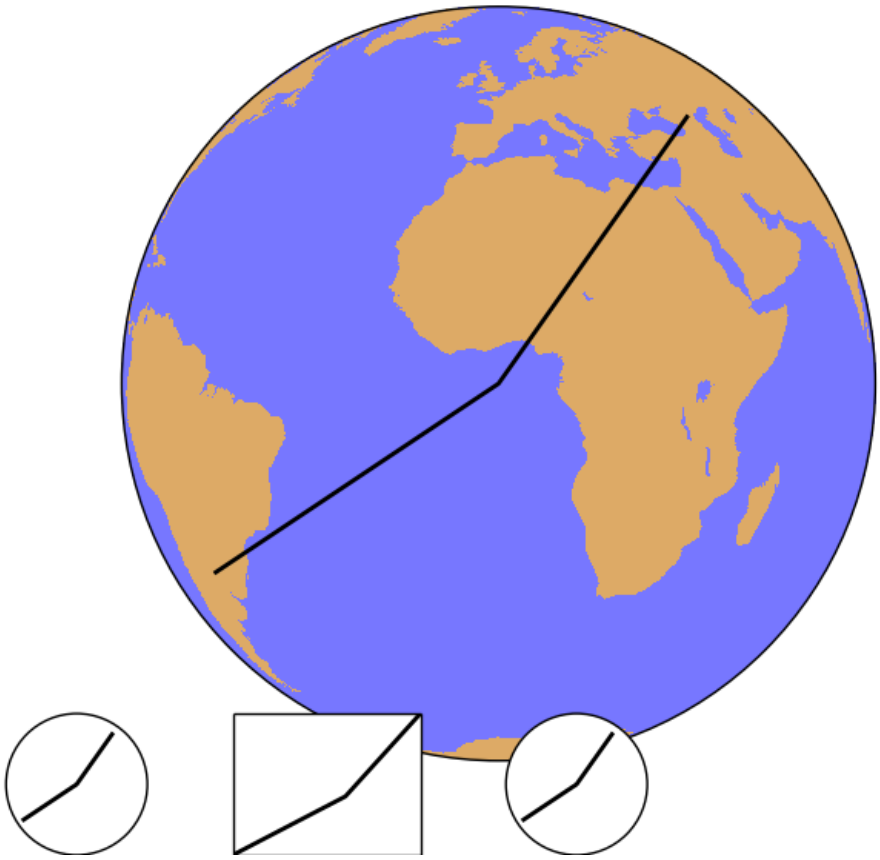
set_axes_limits(ax=None)

- ax is the axis instance that needs its limits to be set

Most of the methods, such as drawcountries, drawrivers, readshapefile... call this method at the end, so it's difficult to put an example to show how does it work:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
fig = plt.figure()

map = Basemap(projection='ortho',
              lat_0=0, lon_0=0)

map.drawlsmask(land_color = "#ddaa66",
               ocean_color="#7777ff",
               resolution = 'c')


x1, y1 = map(-60, -30)
x2, y2 = map(0, 0)
x3, y3 = map(45, 45)

plt.plot([x1, x2, x3], [y1, y2, y3], color='k', linestyle='-', linewidth=2)

ax1 = fig.add_axes([0.1, 0., 0.15, 0.15])
```

```
ax1.set_xticks([])
ax1.set_yticks([])

ax1.plot([x1, x2, x3], [y1, y2, y3], color='k', linestyle='-', linewidth=2)

map.set_axes_limits(ax=ax1)

ax2 = fig.add_axes([0.3, 0., 0.15, 0.15])
ax2.set_xticks([])
ax2.set_yticks([])

ax2.plot([x1, x2, x3], [y1, y2, y3], color='k', linestyle='-', linewidth=2)

ax3 = fig.add_axes([0.5, 0., 0.15, 0.15])
ax3.set_xticks([])
ax3.set_yticks([])

map.plot([x1, x2, x3], [y1, y2, y3], color='k', linestyle='-', linewidth=2, ax=ax3)

plt.show()
```



- After creating the map, the points 1, 2 and 3 are calculated in the map coordinates using the Basemap instance
- The line connecting the three points is plot on the main map
- **Then, three new axes are created using fig.add_axes(x, y, width, height), in fractions of the plot size. In all the cases, the th**

  - The first axis uses the set_axes_limits, so the final shape is a circle, correct for the orthographic projection. Note that plot is called as a method from the axis, not from the Basemap instance
  - The second axis example doesn't call the method, so the shape is not correct. The plot method is also called from the axis instance
  - In the third case, the plot method is called from the Basemap instance, so the method set_axes_limits is called internally from the basemap.plot method, and the final shape is correct

2.4.13 shiftdata

Adds values to the longitudes so they can fit the correct map origin. Changes also the data array so it can fit the new origin. Sometimes, the longitude data is given in an interval different from -180, 180. From 0 to 360, for instance. To draw the data properly, this values must be shifted.

shiftdata(lonsin, datain=None, lon_0=None)

- lonsin the original longitudes. They have to be either a one dimension or two dimension arrays, but always in a regular lat/lon grid
- datain is the data array. If the longitudes have to be cut (-180 becomes 360, for instance), the order of the data array has to be changed. If the data is given, this operation is done
- lon_0 is the map origin. The longitudes will be fit in the interval [lon_0-180,lon_0+180]

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np

map = Basemap(projection='sinu',
              lat_0=0, lon_0=0)

lons = np.arange(30, 390, 30)
lats = np.arange(0, 100, 10)

data = np.indices((lats.shape[0], lons.shape[0]))
data = data[0] + data[1]

print data

print lons

lons, data = map.shiftdata(lons, datain = data, lon_0=0)

print lons

llons, llats = np.meshgrid(lons, lats)

x, y = map(llons, llats)

map.contourf(x, y, data)

map.drawcoastlines()
plt.show()
```

- **The coordinates and data arrays are fake data. Coordinates are created with a simple range, and the data array is the sum**

  - Note that longitudes start at 30

- The longitudes and data is shifted using the shiftdata method
- The new coordinated are passed to 2d using meshgrid, and re-projected to the map projection using the basemap instance
- The filled contour can be now created
- The final result has a white band, which could be avoided using the addcyclic method: *addcyclic*

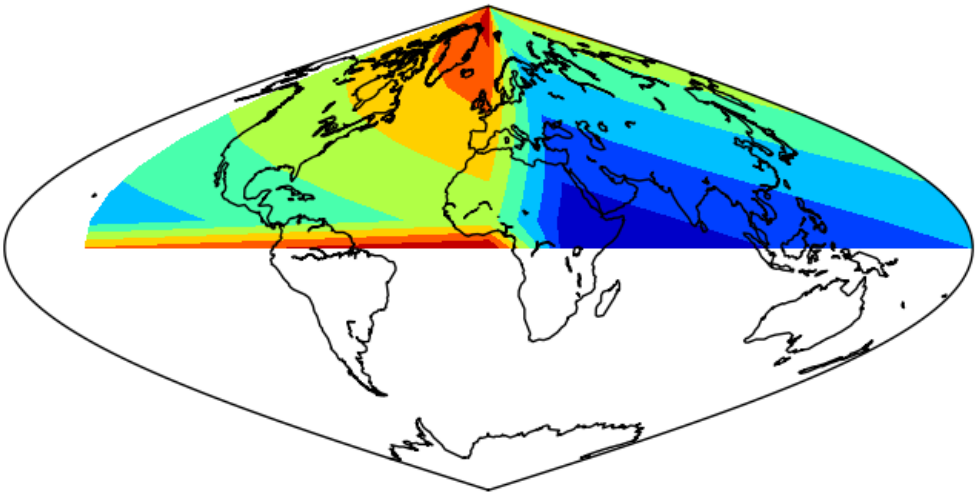


Fig. 2.7: The result applying the method to the data array

2.4.14 shiftgrid

This function, similar to *shiftdata*, moves all the longitudes and data east or west.

basemap.shiftgrid(lon0, datain, lonsin, start=True, cyclic=360.0)

- Note that it's not a basemap method, but a separate function
- lon0 The starting or ending longitude for the final grid. The value must be in the interval of the input longitudes, so sometimes must be set as the starting, and others as the ending with the start argument
- datain the input data array, that will be re-ordered according to the longitudes
- lonsin the input longitudes to be shifted. It's a 1D numpy array, but doesn't have to be in a regular interval

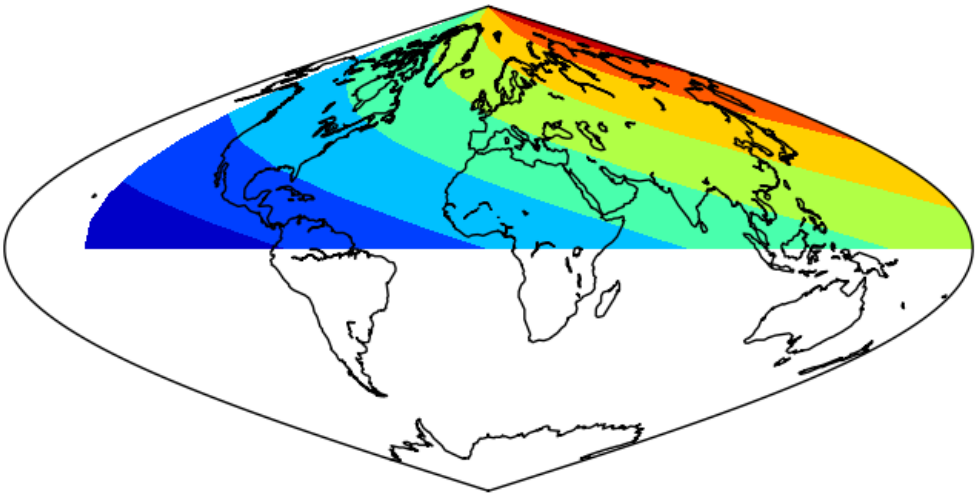


Fig. 2.8: The result without applying the method to the data array. Not that the blue (value 0) is not at longitude 0

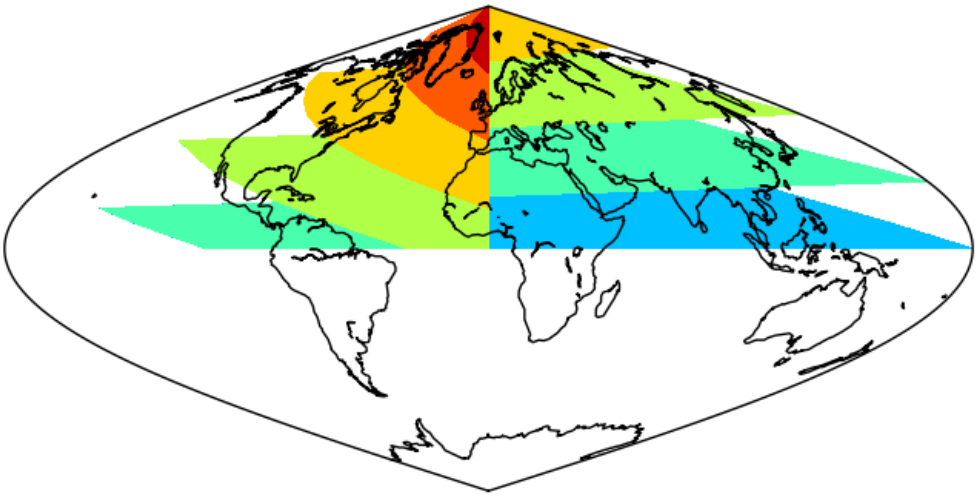
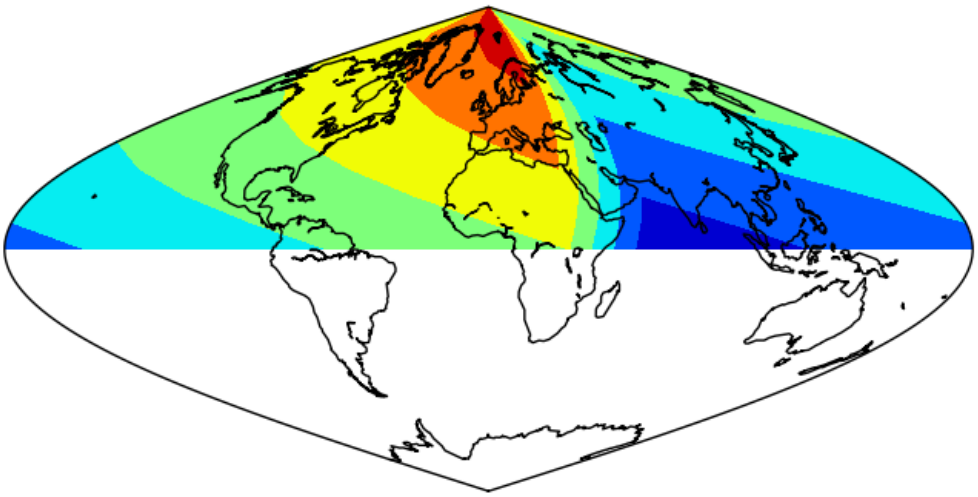


Fig. 2.9: The result without applying shiftdata. Note that the result is not as expected, since the longitudes are outside the right interval

- start argument, which is True by default, sets if the lon0 is the longitude of the initial longitude or the final one at the output array
- cyclic sets the longitude value where the longitudes start again to lon0
- The function returns the re-ordered data and the shifted longitudes

..note: The main difference from *shiftdata*, is that the projection doesn't need to be cylindrical, since it's not a method from the basemap instance, and that the longitudes don't need to have a uniform incerement

```
from mpl_toolkits.basemap import Basemap
from mpl_toolkits.basemap import shiftgrid
import matplotlib.pyplot as plt
import numpy as np

map = Basemap(projection='sinu',
              lat_0=0, lon_0=0)

lons = np.arange(30, 410, 30)
lons[1] = 70
lats = np.arange(0, 100, 10)

data = np.indices((lats.shape[0], lons.shape[0]))
data = data[0] + data[1]

print data
print lons

data, lons = shiftgrid(180., data, lons, start=False)

print data
print lons

llons, llats = np.meshgrid(lons, lats)

x, y = map(llons, llats)

map.contourf(x, y, data)
map.drawcoastlines()
plt.show()
```



- **The coordinates and data arrays are fake data. Coordinates are created with a simple range, and the data array is the sun**
  - Note that longitudes start at 30
  - The second element should be 60, but is changed to 70 to show that the interval doesn't have to be regular, not as in *shiftdata*
- **The longitudes and data is shifted using the shiftgrid method**
  - The lon0 is set to 180, and the start to False, so the final longitude will be 180, and the ones passing this value, will be passed to -180
  - It's not possible to set lon0 to -180, since is outside the input longitudes interval, so the avobe trick must be used
- The new coordinated are passed to 2d using meshgrid, and re-projected to the map projection using the basemap instance
- The filled contour can be now created

2.4.15 tissot

Tissot's indicatrix, or Tissot's ellipse of distortion is the representation of a circle in a map, showing how the projection distorts it. Usually, many of them are represented to show how the distortion varies with the position.

The function has the following arguments:

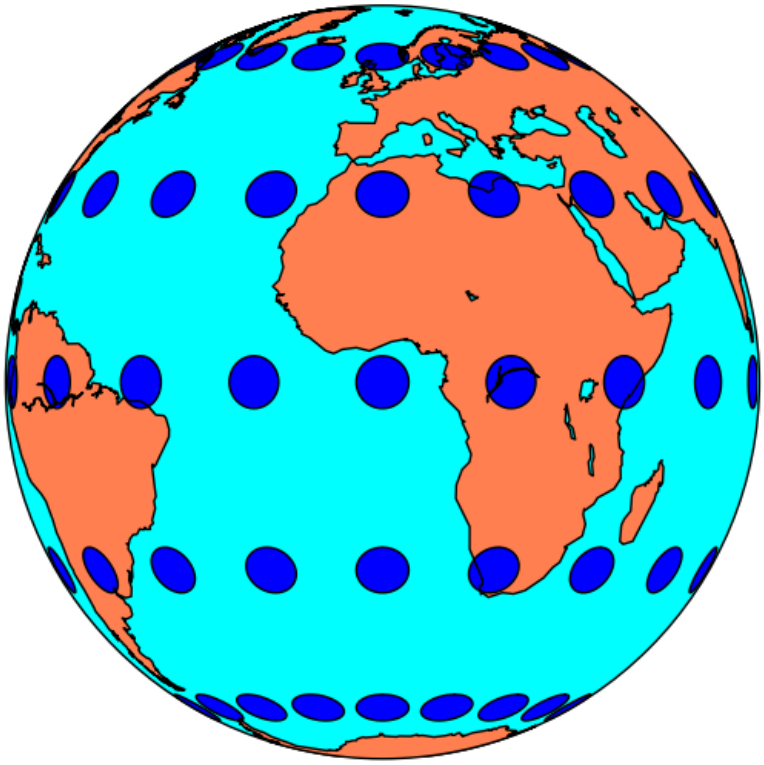
tissot(lon_0, lat_0, radius_deg, npts, ax=None, **kwargs)

- lon_0 and lat_0 indicate the position of the Tissot's ellipse
- radius_deg indicates the radius of the polygon
- npts Is the number of vertices of the polygon that will be used to approximate the ellipse. A higher npts will make better approached ellipses

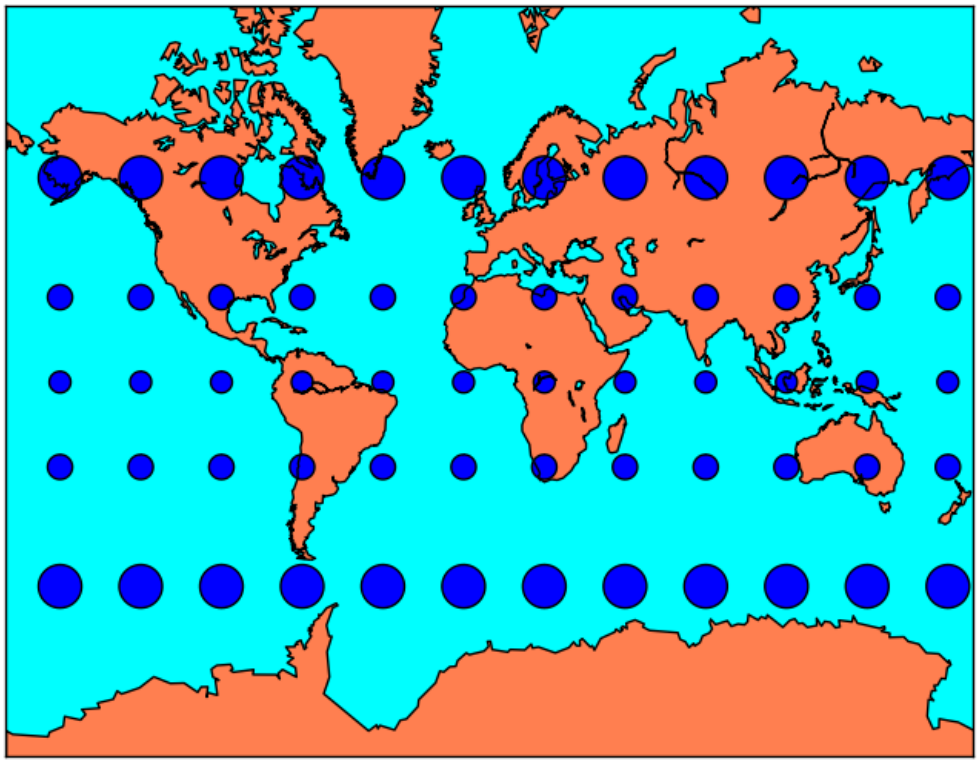
Note: If the circle gets cut by the edges of the map, i.e. starts at longitude -179 and ends at 179, the method can't handle it properly

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap(projection='ortho',
              lat_0=0, lon_0=0)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()

for lon in range(0, 360, 20):
    for lat in range(-60, 90, 30):
        map.tissot(lon, lat, 4, 50)

plt.show()
```

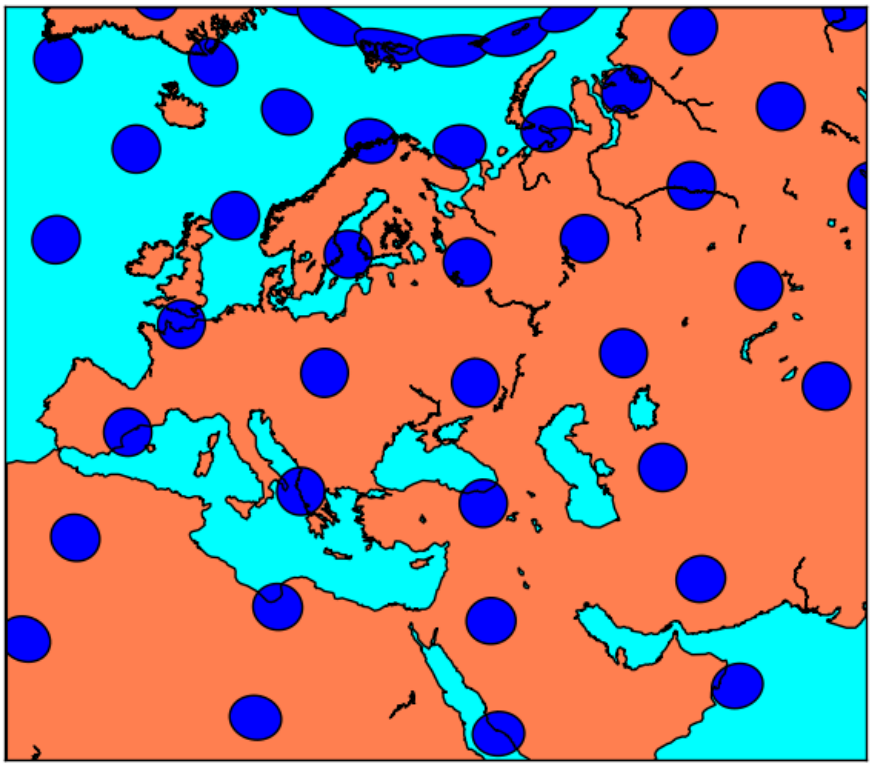
Tissot's indicatrices for the Orthographic projection:



Tissot's indicatrices for the Mercator projection:



Tissot's indicatrices for the Albers Equal Area projection:



2.4.16 transform_scalar

Given a matrix with scalar values in a cylindrical projection, and the longitude and latitude of the points, interpolates the points to a new matrix..

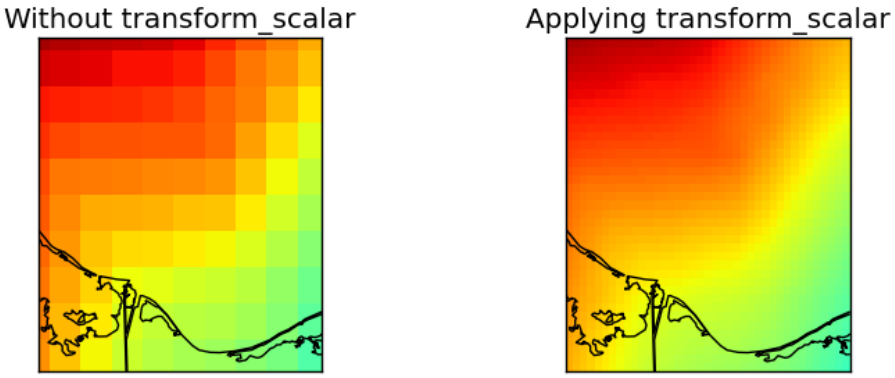
transform_scalar(datin, lons, lats, nx, ny, returnxy=False, checkbounds=False, order=1, masked=False)

- datain is a 2d numpy array with the scalar values
- lons, lats are 1D numpy arrays with the positions of the uin an vin matrices, in geographical coordinates. The input lon-lat grid has to be regular (projections cyl, merc, mill, cea and gall)
- nx and ny are the x any y dimensions of the output grid. The output grid covers all the map, not the original points outside its domain. So the final number of points visible on the map will be nx x ny
- returnxy makes the method to return the lons and lats matrices reprojected to the map coordinates. Just as calling the basemap instance
- checkbounds if set to True, the xin and yin values are checked to be in the bounds of xout and yout. If False, and there are values outside the bounds, the output data array values will be clipped to the boundary values
- masked makes the points outside the new grid to be masked if is set to True, or an arbitrary value if given
- **order sets the interpolation method:**
  - 0 uses the nearest neighbor method

- 1 uses a bilinear interpolation
- 3 uses a cubic spline, and requires scipy.ndimage to be installed

Note: When the input matrix is not regular in longitude-latitude (i.e. is not a cylindric projection), this method can't be used properly, since the longitude-latitude grid won't be regular. See the *interp* example for a solution.

```
from mpl_toolkits.basemap import Basemap
from mpl_toolkits.basemap import shiftgrid
import matplotlib.pyplot as plt
from osgeo import gdal
import numpy as np

fig=plt.figure(figsize=(9, 3))

map = Basemap(projection='merc',
              lat_0=0,
              resolution='h',
              llcrnrlon=32,
              llcrnrlat=31,
              urcrnrlon=33,
              urcrnrlat=32)

ds = gdal.Open("../sample_files/dem.tiff")
data = ds.ReadAsArray()

lons = np.linspace(0, 40, data.shape[1])
lats = np.linspace(0, 35, data.shape[0])

ax = fig.add_subplot(121)
ax.set_title('Without transform_scalar')

llons, llats = np.meshgrid(lons, lats)
x, y = map(llons, llats)

map.pcolormesh(x, y, data, vmin=900, vmax=950)

map.drawcoastlines()

ax = fig.add_subplot(122)
ax.set_title('Applying transform_scalar')

data_interp, x, y = map.transform_scalar(data, lons, lats, 40, 40, returnxy=True)

map.pcolormesh(x, y, data_interp, vmin=900, vmax=950)

map.drawcoastlines()

plt.show()
```

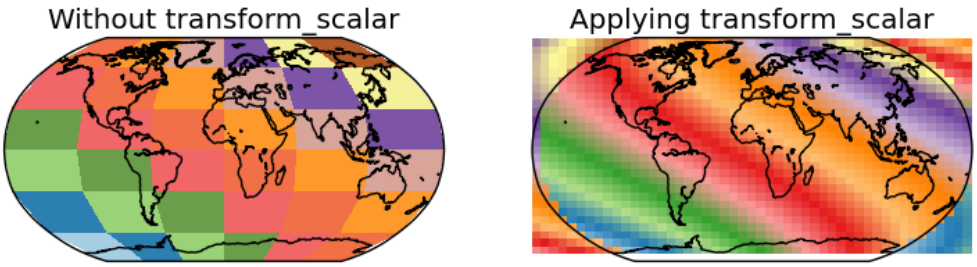


- The data taken for this example is a DEM data for another region and projection, but we'll use fake longitudes and latitudes so it can be used.
- numpy linspace is used to generate an equal spaced longitudes and latitudes arrays. They have to be 1D to use transform_scalar, so the projection has to be cylindrical (projections cyl, merc, mill, cea and gall)
- **The original field is drawn on the first map**
  - lons and lats are converted first to 2D arrays using meshgrid
  - The longitudes and latitudes are converted to the mercator projection using the basemap instance
  - *pcolormesh* is used to draw the result, taking care to set the maximum and minimum values, so the two maps behave the same way
- **transform_scalar is applied**
  - The returnxy argument set to true, so the new grid positions can be get easily
  - The size of the new grid will be 40x40, so the pixels are still visible, but small. A bigger number would make the pixels to be much smaller
- The same ref:*pcolormesh* is used to plot the data. The maximum and minimum values are the same as in the later case. If not, the function would assume only the values in the map region, so the colors would be different

Note: Masked doesn't seem to work

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np


fig=plt.figure(figsize=(9, 3))

map = Basemap(projection='robin',
              lat_0=0, lon_0=0)

lons = np.arange(-180, 190, 60)
lats = np.arange(-90, 100, 30)

data = np.indices((lats.shape[0], lons.shape[0]))
data = data[0] + data[1]

llons, llats = np.meshgrid(lons, lats)
```

```
ax = fig.add_subplot(121)
ax.set_title('Without transform_scalar')

x, y = map(llons, llats)

map.pcolormesh(x, y, data, cmap='Paired')

map.drawcoastlines()

ax = fig.add_subplot(122)
ax.set_title('Applying transform_scalar')

data_interp, x, y = map.transform_scalar(data, lons, lats, 50, 30, returnxy=True,␣
↪masked=True)

map.pcolormesh(x, y, data_interp, cmap='Paired')

map.drawcoastlines()

plt.show()
```



- In this case, the data is the same as the used in the *shiftdata* example
- **Since the map covers the whole world, some of the points in the output grid are outside the world**
  - Using masked = True, those points should have no-data values, but this doesn't seem to work, and the points are drawn anyway, creating a very strange effect

2.4.17 transform_vector

Given two matrices of the east-west and north-south components of a vectorial field in a cylindrical projection, and the longitude and latitude of the points, rotates the vectors so they represent the direction properly on the map projection, while interpolates the points to a new matrix..

Some functions, such as barbs, quiver or streamplot, that use vectorial data, asks the vector components to be in the map coordinates i.e. u is from left to right, v from up do down. If the available data is in geographical coordinates i.e. west-east and north-south, these coordinates have to be rotated or the vector direction won't be plot properly. This is the aim of the rotate_vector method.

When drawing barbs, quiver or stream lines, the number of available points may be too low, so interpolating them to a new matrix with more elements can be used to get a plot with a nice number of elements.

The method *rotate_vector* does the same function, but without interpolating the points

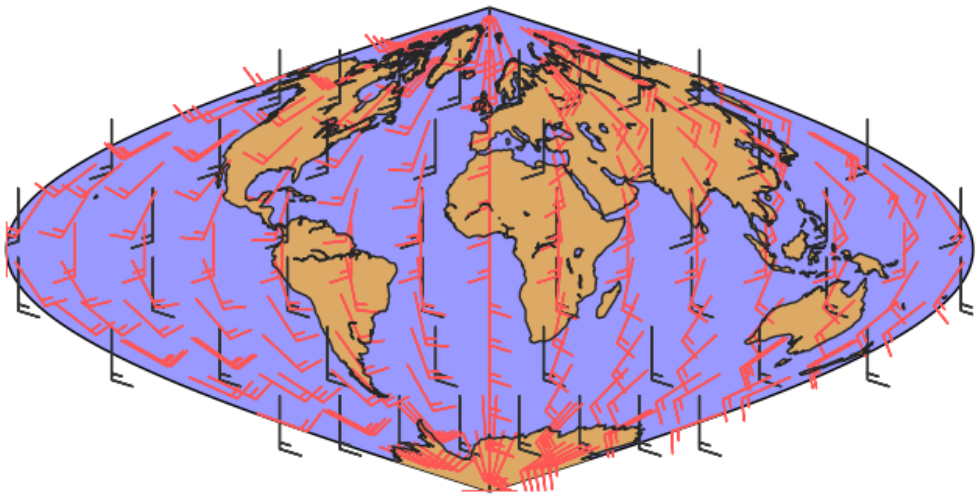
transform_vector(uin, vin, lons, lats, nx, ny, returnxy=False, checkbounds=False, order=1, masked=False)

- uin and vin are the input data matrices. The directions are the geographical, so the u component is west-east and the v component, north-south
- lons, lats are 1D numpy arrays with the positions of the uin an vin matrices, in geographical coordinates. The input lon-lat grid has to be regular (projections cyl, merc, mill, cea and gall)
- nx and ny are the x any y dimensions of the output grid. The output grid covers all the map, not the original points outside its domain. So the final number of points visible on the map will be nx x ny
- returnxy makes the method to return the lons and lats matrices reprojected to the map coordinates. Just as calling the basemap instance
- checkbounds if set to True, the xin and yin values are checked to be in the bounds of xout and yout. If False, and there are values outside the bounds, the output data array values will be clipped to the boundary values
- masked makes the points outside the new grid to be masked if is set to True, or an arbitrary value if given
- **order sets the interpolation method:**
  - 0 uses the nearest neighbor method
  - 1 uses a bilinear interpolation
  - 3 uses a cubic spline, and requires scipy.ndimage to be installed

Note: When the input matrix is not regular in longitude-latitude (i.e. is not a cylindric projection), this method can't be used properly, since the longitude-latitude grid won't be resular. See the *interp* example for a solution.

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from osgeo import gdal
import numpy as np

map = Basemap(projection='sinu',
              lat_0=0, lon_0=0)

lons = np.linspace(-180, 180, 8)
lats = np.linspace(-90, 90, 8)

v10 = np.ones((lons.shape)) * 15
u10 = np.zeros((lons.shape))

u10, v10 = np.meshgrid(u10, v10)

u10_rot, v10_rot, x_rot, y_rot = map.transform_vector(u10, v10,
                                                     lons, lats,
                                                     15, 15,
                                                     returnxy=True)

map.drawmapboundary(fill_color='#9999FF')
map.fillcontinents(color='#ddaa66', lake_color='#9999FF', zorder = 0)
map.drawcoastlines(color = '0.15')

#Drawing the original points
lons, lats = np.meshgrid(lons, lats)
x, y = map(lons, lats)
map.barbs(x, y, u10, v10,
    pivot='middle', barbcolor='#333333')
```

```
#Drawing the rotated & interpolated points
map.barbs(x_rot, y_rot, u10_rot, v10_rot,
    pivot='middle', barbcolor='#ff5555')

plt.show()
```



- lons and lats are created in an equal spaced grid covering all the globe, using linspace
- v10 and u10 are created so they represent a south to north wind (v10 = 10, u10 = 0). They are transformed to a 2D array using numpy.meshgrid
- **Once the data is created, the rotated and interpolated vectors and the new grid are created using transform_vector**
  - Setting nx and ny to 15, the new grid will be 15x15 in the map projection, so this is the final number of points visible in the final map
- The original and rotated-interpolated fields are drawn

CHAPTER 3

Cookbook

3.1 Custom colormaps

Matplotlib color maps are really powerful, much more than the usual possibilities in other softwares. But they are quite difficult to understand, and most of the times, a simple list with intervals and colors is easier to work with:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from osgeo import gdal
from numpy import linspace
from numpy import meshgrid
from matplotlib.colors import LinearSegmentedColormap

cmap1 = LinearSegmentedColormap.from_list("my_colormap", ((0, 0, 0), (1, 1, 1)), N=6,␣
↪gamma=1.0)

map = Basemap(projection='tmerc',
              lat_0=0, lon_0=3,
              llcrnrlon=1.819757266426611,
              llcrnrlat=41.583851612359275,
              urcrnrlon=1.841589961763497,
              urcrnrlat=41.598674173123)

ds = gdal.Open("../sample_files/dem.tiff")
data = ds.ReadAsArray()

x = linspace(0, map.urcrnrx, data.shape[1])
y = linspace(0, map.urcrnry, data.shape[0])

xx, yy = meshgrid(x, y)

map.contourf(xx, yy, data, (400, 600, 800, 1000, 1200),cmap=cmap1)

plt.show()
```

- The way to create the contourf plot is taken from the *contourf* example
- **The color map is created using the LinearSegmentedColormap.from_list static method. This method has the following arg**

    1. The name given to the color map
    2. The color list. This is a list or sequence, each element containing three floats in the range 0 to 1, which are the red, green and blue values of the color
    3. N is the number of color levels to create. If the number is lower than the list of colors length, the list will be truncated If it's longer, some colors will be repeated

- In our example, the values will go from black to white, in six levels
- The contourf is created forcing a level in each of the elements of the sequence

GDAL includes an utility named gdaldem, that classifies a raster using the values defined in a file. The file format is originally used by the GRASS r.colors function. I like the format, since is really easy to understand and can be used from different software. Here's how to read it and use it with basemap:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from osgeo import gdal
from numpy import linspace
from numpy import meshgrid
from os.path import exists
from matplotlib.colors import LinearSegmentedColormap
```

```
def read_color_table(color_file):
    '''
    The method for reading the color file.
    '''
    colors = []
    levels = []
    if exists(color_file) is False:
        raise Exception("Color file " + color_file + " does not exist")
    fp = open(color_file, "r")
    for line in fp:
        if line.find('#') == -1 and line.find('/') == -1:
            entry = line.split()
            levels.append(eval(entry[0]))
            colors.append((int(entry[1])/255.,int(entry[2])/255.,int(entry[3])/255.))

    fp.close()

    cmap = LinearSegmentedColormap.from_list("my_colormap",colors, N=len(levels),␣
↪gamma=1.0)

    return levels, cmap

levels, cmap = read_color_table("../sample_files/colorfile.clr")

map = Basemap(projection='tmerc',
              lat_0=0, lon_0=3,
              llcrnrlon=1.819757266426611,
              llcrnrlat=41.583851612359275,
              urcrnrlon=1.841589961763497,
              urcrnrlat=41.598674173123)

ds = gdal.Open("../sample_files/dem.tiff")
data = ds.ReadAsArray()

x = linspace(0, map.urcrnrx, data.shape[1])
y = linspace(0, map.urcrnry, data.shape[0])
xx, yy = meshgrid(x, y)

map.contourf(xx, yy, data, levels, cmap=cmap)

plt.show()
```

- **The function read_color_table opens and reads the color file, and returns the levels defined in the file, and a color map tha**
  - Checks if some line is commented
  - The original values are in the range 0-255, and have to be converted to 0-1
  - The color map is created using the LinearSegmentedColormap.from_list static method
  - The function returns the levels and the color map so they can be used together or separately
- contourf is used with the values defined in the file

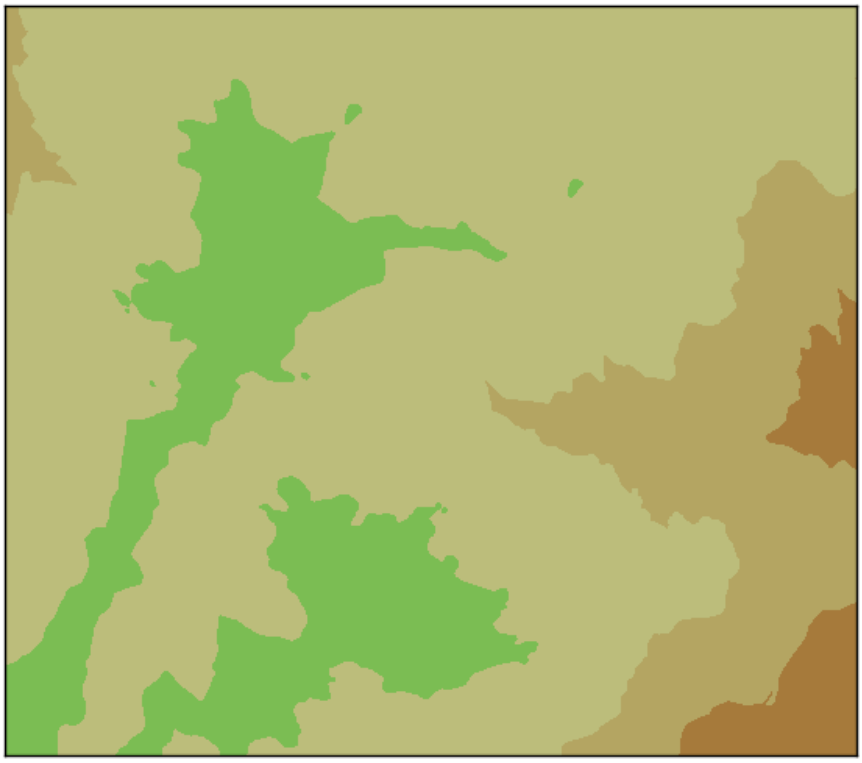


Fig. 3.1: The result if the levels are passed as an argument. Not all the colors are used, since some of them are defined out of the data range

Fig. 3.2: The result if the levels are not passed as an argument. The contourf method creates as many levels as number of colors of the cmap within the data values range

3.2 Multiple maps using subplots

Drawing multiple maps in the same figure is possible using matplotlib's *subplots*. There are several ways to use them, and depending on the complexity of the desired figure, one or other is better:

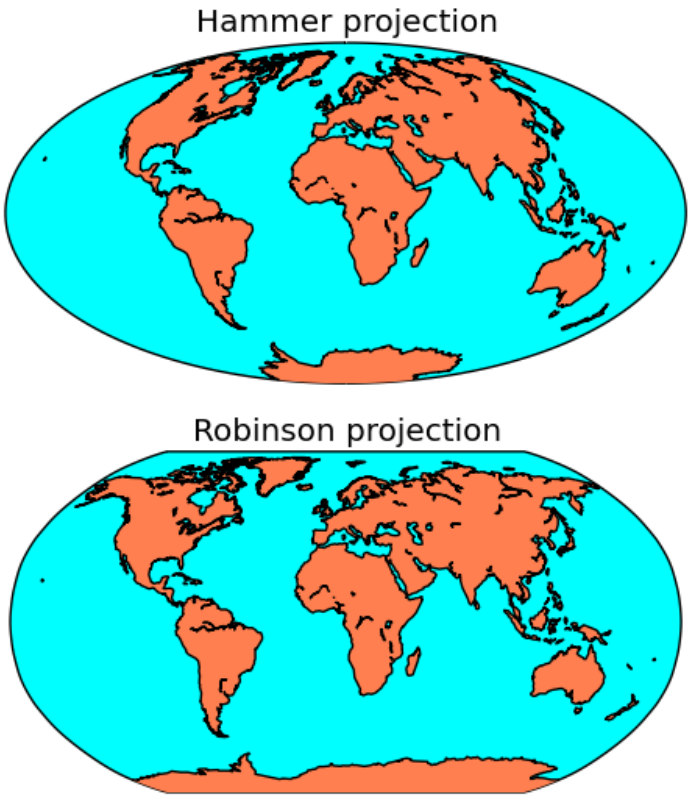
- Creating the axis using subplot directly with add_subplot
- Creating the subplots with pylab.subplots
- Using subplot2grid
- Creating *Inset locators*

3.2.1 Using add_subplot

This is the prefered way to add subplots in most of the examples:

```
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap

fig = plt.figure()

ax = fig.add_subplot(211)
ax.set_title("Hammer projection")
map = Basemap(projection='hammer', lon_0 = 10, lat_0 = 50)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()

ax = fig.add_subplot(212)
ax.set_title("Robinson projection")
map = Basemap(projection='robin', lon_0 = 10, lat_0 = 50)

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()

plt.show()
```

- **Before calling the basemap constructor, the fig.add_subplot method is called. The three numbers are:** # The number of rows in the final figure # The number of columns in the final figure # Which axis (subplot) to use, counting from the one top-left axis, as explained at this StackOverflow question
- Once the axis is created, the map created later will use it automatically (although the ax argument can be passed to use the selected axis)
- A title can be added to each subplot using set_title()



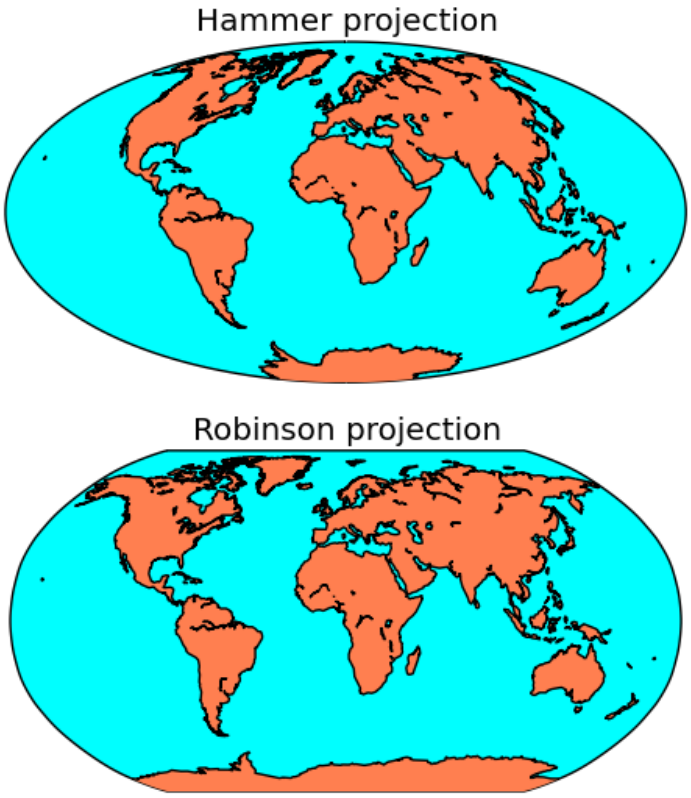
3.2.2 Generating the subplots at the beginning with plt.subplots

Using the add_subplot is a bit confusing in my opinion. If the basemap instance is created without the *ax* argument, the possibility of coding bugs is very high. So, to create the plots at the beginning and using them later, pyplot.subplots can be used:

```
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap

fig, axes = plt.subplots(2, 1)

axes[0].set_title("Hammer projection")
map = Basemap(projection='hammer', lon_0 = 10, lat_0 = 50, ax=axes[0])

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()

axes[1].set_title("Robinson projection")
map = Basemap(projection='robin', lon_0 = 10, lat_0 = 50, ax=axes[1])

map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
map.drawcoastlines()
```

```
plt.show()
```

- The arguments passed to the subplots method, are the number of rows and columns to be created
- The subplots method returns figure object, and a list of the created axes (subplots), where the first element is the one at the top-left position
- When creating the basemap instance, the ax argument must be passed, using the created axes



The result is the same as in the previous example

3.2.3 Using subplot2grid

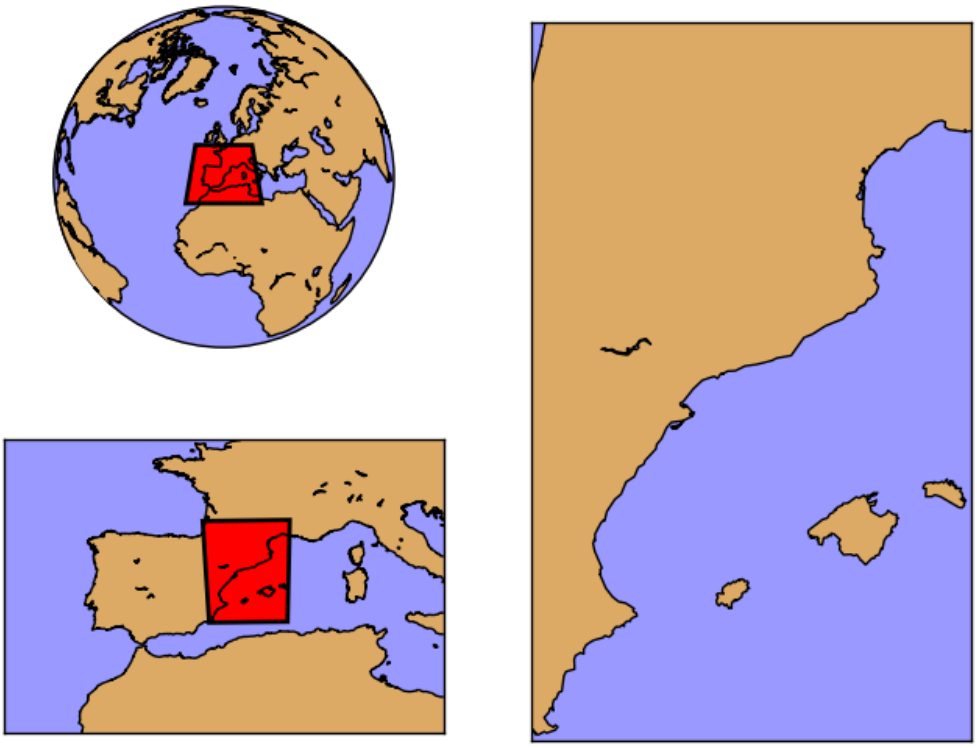
When the number of subplots is bigger, or the subplots must have different sizes, *subplot2grid* or *gridspec* can be used. Here is an example with subplor2grid:

```
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
from matplotlib.path import Path
import matplotlib.patches as patches

fig = plt.figure()

ax1 = plt.subplot2grid((2,2), (0,0))
```

```
ax2 = plt.subplot2grid((2,2), (1,0))
ax3 = plt.subplot2grid((2,2), (0,1), rowspan=2)

map1 = Basemap(projection='ortho', lon_0 = 0, lat_0 = 40, ax=ax1)

map1.drawmapboundary(fill_color='#9999FF')
map1.fillcontinents(color='#ddaa66',lake_color='#9999FF')
map1.drawcoastlines()

map2 = Basemap(projection='cyl', llcrnrlon=-15,llcrnrlat=30,urcrnrlon=15.,
↪urcrnrlat=50., resolution='i', ax=ax2)

map2.drawmapboundary(fill_color='#9999FF')
map2.fillcontinents(color='#ddaa66',lake_color='#9999FF')
map2.drawcoastlines()


map3 = Basemap(llcrnrlon= -1., llcrnrlat=37.5, urcrnrlon=4.5, urcrnrlat=44.5,
             resolution='i', projection='tmerc', lat_0 = 39.5, lon_0 = 3, ax=ax3)

map3.drawmapboundary(fill_color='#9999FF')
map3.fillcontinents(color='#ddaa66',lake_color='#9999FF')
map3.drawcoastlines()


#Drawing the zoom rectangles:

lbx1, lby1 = map1(*map2(map2.xmin, map2.ymin, inverse= True))
ltx1, lty1 = map1(*map2(map2.xmin, map2.ymax, inverse= True))
rtx1, rty1 = map1(*map2(map2.xmax, map2.ymax, inverse= True))
rbx1, rby1 = map1(*map2(map2.xmax, map2.ymin, inverse= True))

verts1 = [
    (lbx1, lby1), # left, bottom
    (ltx1, lty1), # left, top
    (rtx1, rty1), # right, top
    (rbx1, rby1), # right, bottom
    (lbx1, lby1), # ignored
    ]

codes2 = [Path.MOVETO,
          Path.LINETO,
          Path.LINETO,
          Path.LINETO,
          Path.CLOSEPOLY,
          ]

path = Path(verts1, codes2)
patch = patches.PathPatch(path, facecolor='r', lw=2)
ax1.add_patch(patch)

lbx2, lby2 = map2(*map3(map3.xmin, map3.ymin, inverse= True))
ltx2, lty2 = map2(*map3(map3.xmin, map3.ymax, inverse= True))
rtx2, rty2 = map2(*map3(map3.xmax, map3.ymax, inverse= True))
rbx2, rby2 = map2(*map3(map3.xmax, map3.ymin, inverse= True))

verts2 = [
    (lbx2, lby2), # left, bottom
```

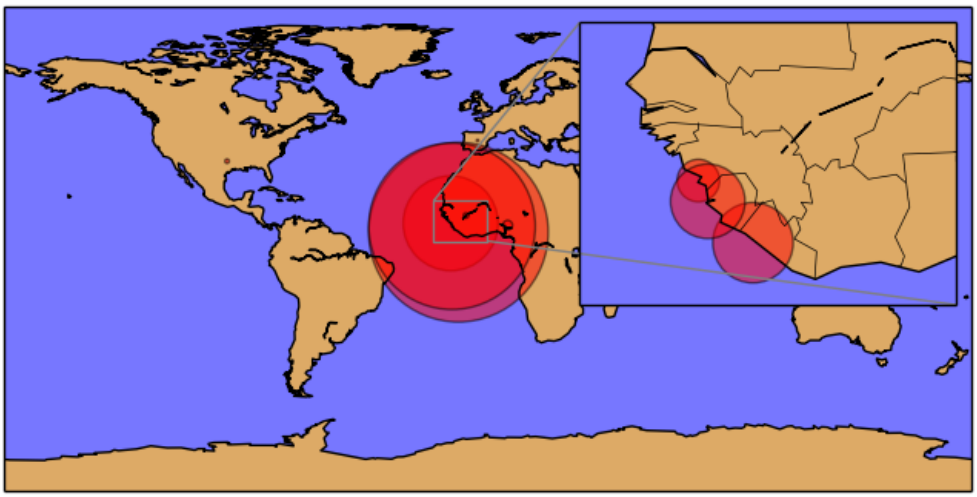
```
    (ltx2, lty2), # left, top
    (rtx2, rty2), # right, top
    (rbx2, rby2), # right, bottom
    (lbx2, lby2), # ignored
    ]

codes2 = [Path.MOVETO,
         Path.LINETO,
         Path.LINETO,
         Path.LINETO,
         Path.CLOSEPOLY,
         ]

path = Path(verts2, codes2)
patch = patches.PathPatch(path, facecolor='r', lw=2)
ax2.add_patch(patch)

plt.show()
```

- **Each subplot is created with the method subplot2grid. The three possible arguments are:** # The output matrix shape, in a sequence with two elements, the y size and x size # The position of the created subplot in the output matrix # The rowspan or colspan. As in the html tables, the cells can occuy more than one position in the output matrix. The rowspan and colspan arguments can set it, as in the example, where the second column has only one cell occupying two rows
- **In the example, the layout is used to show different zoom levels. Each level is indicated with a polygon on the previous ma**
  - **The position of each corner of the bounding box is calculated**
    - The corners can be retrieved using the xmin, xmax, ymin, ymax fields of the basemap instance (the next map, since this will indicate the zoomed area)
    - Since the fields are in the projected units, the basemap instance with inverse argument is used. See the *Using the Basemap instance to convert units* section to see how does it work
    - The calculated longitudes and latitudes of each corner is passed to the current map instance to get the coordinates in the current map projection. Since we have the values in a sequence, an asterisk is used to unpack them
  - Once the points are known, the polygon is created using the Path class



3.2.4 Inset locators

A small map inside the main map can be created using *Inset locators*, explained in an other section. The result is better than just creating a small subplot inside the main plot:



3.3 Basemap in 3D

Even though many people don't like them, maps with 3d elements can be created using basemap and the matplotlib mplot3d toolkit.

3.3.1 Creating a basic map

The most important thing to know when starting with 3d matplotlib plots is that the *Axes3D* class has to be used. To add geographical data to the map, the method *add_collection3d* will be used:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.basemap import Basemap

map = Basemap()

fig = plt.figure()
ax = Axes3D(fig)

'''
ax.azim = 270
ax.elev = 90
```

```
ax.dist = 5
'''

ax.add_collection3d(map.drawcoastlines(linewidth=0.25))
ax.add_collection3d(map.drawcountries(linewidth=0.35))

plt.show()
```

- The ax variable is in this example, an Axes3D instance. All the methods will be used from this instance, so they need to support 3D operations, which doesn't occur in many cases on the basemap methods
- The commented block shows how to rotate the resulting map so the view is better
- To draw lines, just use the add_collection3d method with the output of any of the basemap methods that return an matplotlib.patches.LineCollection object, such as drawcountries

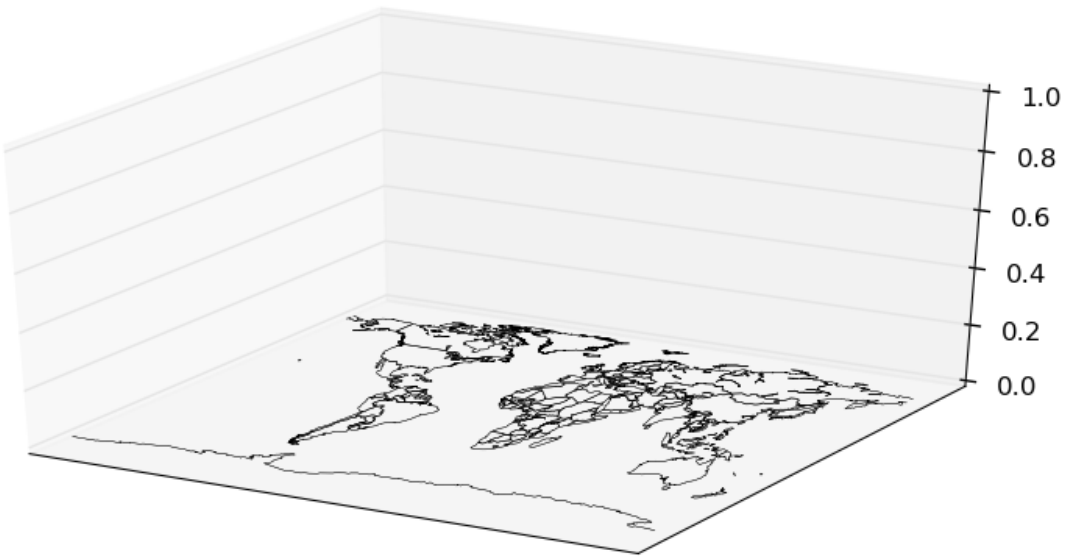


Fig. 3.3: Basic usage, the axis rotation is the one by default

3.3.2 Filling the polygons

Unfortunately, the basemap *fillcontinents* method doesn't return an object supported by add_collection3d (PolyCollection, LineColleciton, PatchCollection), but a list of matplotlib.patches.Polygon objects.

The solution, of course, is to create a list of PolyCollection:

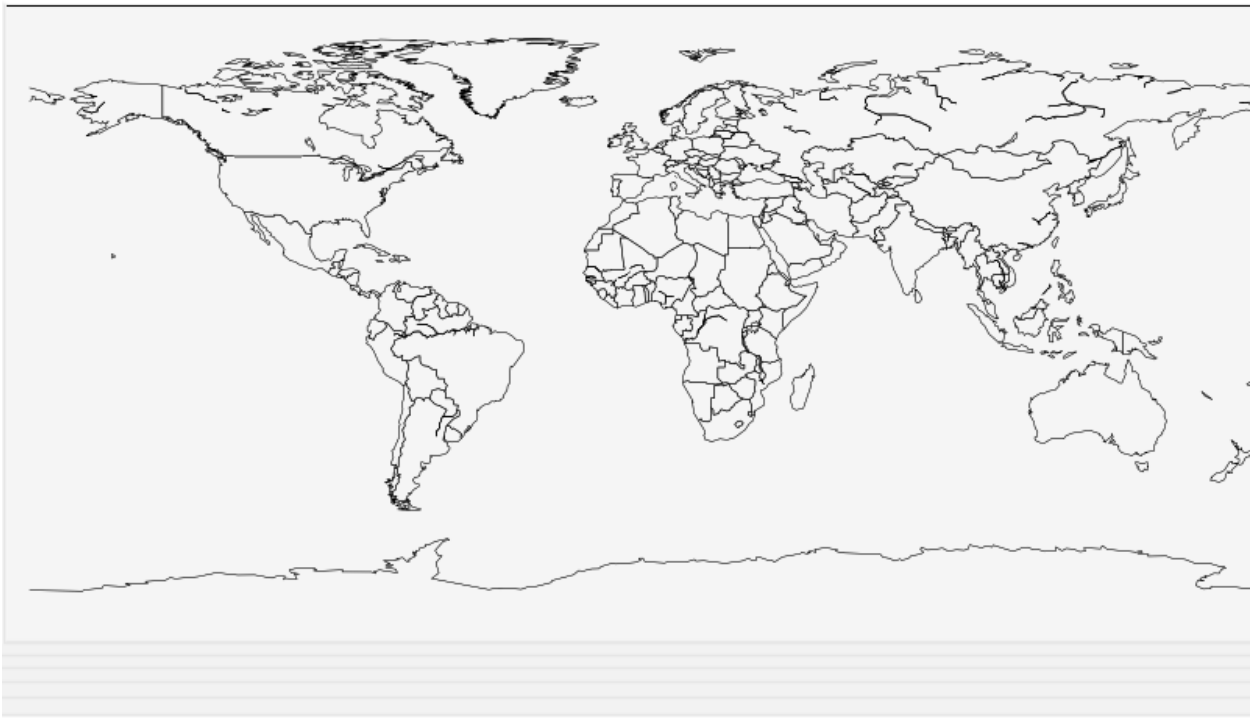
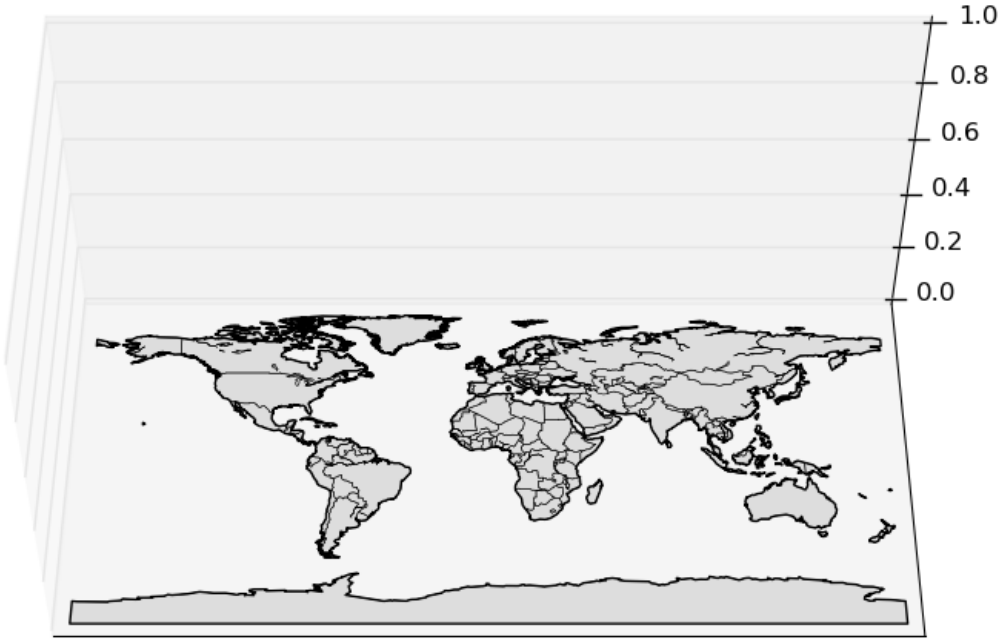


Fig. 3.4: The axis rotation is set so the map is watched from the z axis, like when drawing it in 2D

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.basemap import Basemap
from matplotlib.collections import PolyCollection

map = Basemap()

fig = plt.figure()
ax = Axes3D(fig)

ax.azim = 270
ax.elev = 50
ax.dist = 8

ax.add_collection3d(map.drawcoastlines(linewidth=0.25))
ax.add_collection3d(map.drawcountries(linewidth=0.35))

polys = []
for polygon in map.landpolygons:
    polys.append(polygon.get_coords())


lc = PolyCollection(polys, edgecolor='black',
                    facecolor='#DDDDDD', closed=False)

ax.add_collection3d(lc)


plt.show()
```

- The coast lines and the countries are drawn as in the previous example
- To create the PolyCollection, the polygons are needed, but the *Basemap* object has it in the field landpolygons. (There are others for the rest of the included polygons, such as the countries)
- For each of the polygons, the coordinates can be retrieved as a list of floats using the get_coords method. (he are _geoslib.Polygon objects)
- Once a list of coordinates list is created, the PoilyCollection can be built
- The PolyCollection is added using add_collection3d, as we did with the lines
- If the original polygons are added using fillcontinents, matplotlib says that doesn't has the methods to convert it to 3D



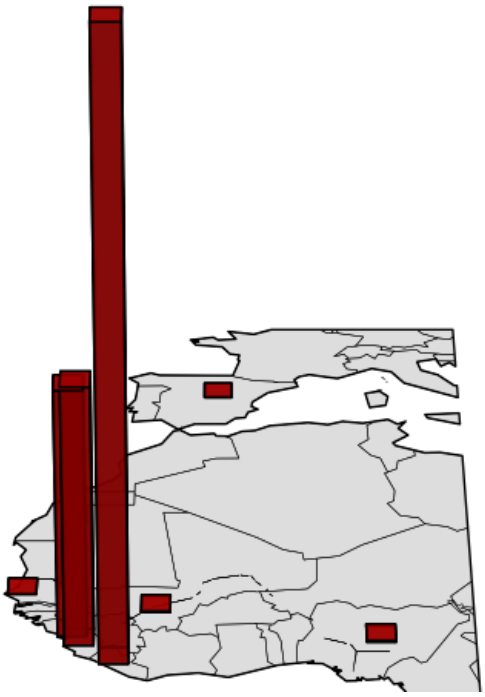
3.3.3 Adding 3D bars

Creating a 3D map hasn't got sense if no 3D data is drawn on it. The Axes3D class has the bar3d method that draws 3D bars. It can be added on the map using the 3rd dimension:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.basemap import Basemap
from matplotlib.collections import PolyCollection
import numpy as np

map = Basemap(llcrnrlon=-20,llcrnrlat=0,urcrnrlon=15,urcrnrlat=50,)

fig = plt.figure()
ax = Axes3D(fig)

ax.set_axis_off()
ax.azim = 270
ax.dist = 7

polys = []
for polygon in map.landpolygons:
    polys.append(polygon.get_coords())
```

```
lc = PolyCollection(polys, edgecolor='black',
                    facecolor='#DDDDDD', closed=False)

ax.add_collection3d(lc)
ax.add_collection3d(map.drawcoastlines(linewidth=0.25))
ax.add_collection3d(map.drawcountries(linewidth=0.35))

lons = np.array([-13.7, -10.8, -13.2, -96.8, -7.99, 7.5, -17.3, -3.7])
lats = np.array([9.6, 6.3, 8.5, 32.7, 12.5, 8.9, 14.7, 40.39])
cases = np.array([1971, 7069, 6073, 4, 6, 20, 1, 1])
deaths = np.array([1192, 2964, 1250, 1, 5, 8, 0, 0])
places = np.array(['Guinea', 'Liberia', 'Sierra Leone','United States', 'Mali',
↪'Nigeria', 'Senegal', 'Spain'])

x, y = map(lons, lats)

ax.bar3d(x, y, np.zeros(len(x)), 2, 2, deaths, color= 'r', alpha=0.8)

plt.show()
```

- The map is zoomed to fit the needs of the Ebola cases dataset
- The axes are eliminated with the method set_axis_off
- The bar3d needs the x, y and z positions, plus the delta x, y and z. To be properly drawn, the z position must be 0, and the delta z, the final value



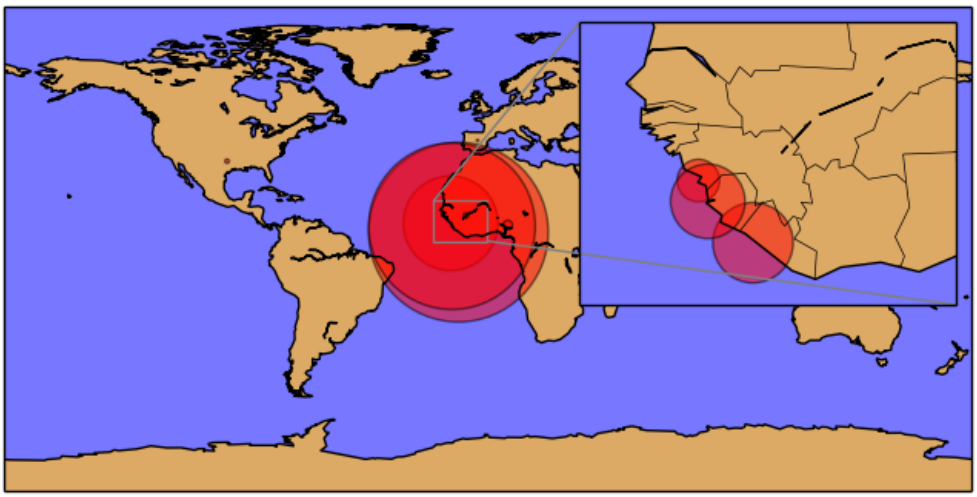
3.4 Inset locators

Insert locator is a cool class that zooms a part of the plot and draws it on the plot itself, showing the zoomed zone.

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111)

map = Basemap(projection='cyl',
              lat_0=0, lon_0=0)

map.drawmapboundary(fill_color='#7777ff')
map.fillcontinents(color='#ddaa66', lake_color='#7777ff', zorder=0)
map.drawcoastlines()

lons = np.array([-13.7, -10.8, -13.2, -96.8, -7.99, 7.5, -17.3, -3.7])
lats = np.array([9.6, 6.3, 8.5, 32.7, 12.5, 8.9, 14.7, 40.39])
cases = np.array([1971, 7069, 6073, 4, 6, 20, 1, 1])
```

```
deaths = np.array([1192, 2964, 1250, 1, 5, 8, 0, 0])
places = np.array(['Guinea', 'Liberia', 'Sierra Leone','United States', 'Mali',
↪'Nigeria', 'Senegal', 'Spain'])

x, y = map(lons, lats)

map.scatter(x, y, s=cases, c='r', alpha=0.5)

axins = zoomed_inset_axes(ax, 7, loc=1)
axins.set_xlim(-20, 0)
axins.set_ylim(3, 18)

plt.xticks(visible=False)
plt.yticks(visible=False)

map2 = Basemap(llcrnrlon=-20,llcrnrlat=3,urcrnrlon=0,urcrnrlat=18, ax=axins)
map2.drawmapboundary(fill_color='#7777ff')
map2.fillcontinents(color='#ddaa66', lake_color='#7777ff', zorder=0)
map2.drawcoastlines()
map2.drawcountries()

map2.scatter(x, y, s=cases/5., c='r', alpha=0.5)

mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="0.5")

plt.show()
```

- The base scatter and map is a regular *scatter* plot
- **The inset locator is created using the method zoomed_inset_axes**
  - ax is the axis to be zoomed by the inset locator
  - 7 is a zoom level (to be overwritten later)
  - loc is the position of the inset locator (upper right in this case)
- set_xlim and set_ylim change the zone to cover by the inset locator. Since we are working with the cyl projection, longitudes and latitudes can be used directly, without any transformation
- xticks and yticks are set to false to delete the new axis labels, which are not necessary on a map
- A new map is created using the zoomed limits and the axis created by zoomed_inset_axes. Now, all the methods will be using this new zoomed map, matching the correct zone
- The map is drawn again using the new basemap instance, to plot the zoomed zone
- mark_inset draws the lines showing the zoomed zone



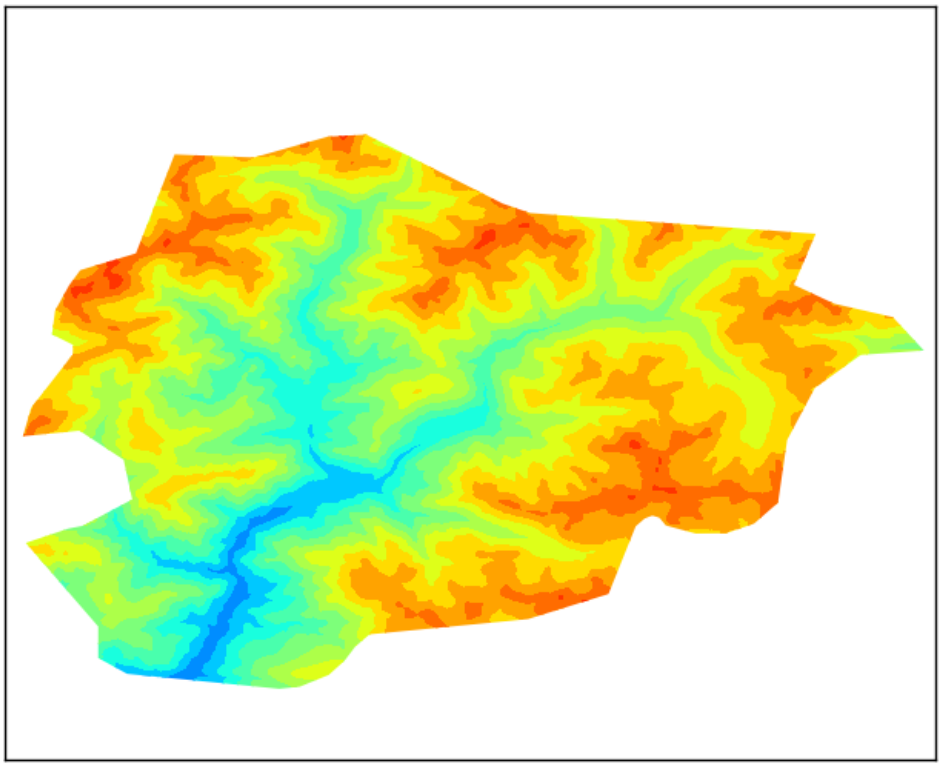
3.5 Clipping a raster with a shapefile

3.5.1 Getting some data

The example plots some elevation data, taken from the SRTM. After looking for some options, the easiest to work with was this one: http://srtm.csi.cgiar.org/SELECTION/inputCoord.asp or download the file directly: http://srtm.csi.cgiar.org/SRT-ZIP/SRTM_V41/SRTM_Data_GeoTiff/srtm_37_04.zip

The shapefile will be the border of Andorra, taken from Natural Earth

The result is a little poor because the resolution is low, but works well for the example.



3.5.2 The code

The example script uses pyshp for reading the shapefile. Of course, ogr could be used too, but not fiona, since fails when used with gdal in the same script.

```
from mpl_toolkits.basemap import Basemap
from matplotlib.path import Path
from matplotlib.patches import PathPatch
import matplotlib.pyplot as plt
from osgeo import gdal
import numpy
import shapefile

fig = plt.figure()
ax = fig.add_subplot(111)

sf = shapefile.Reader("ne_10m_admin_0_countries")

for shape_rec in sf.shapeRecords():
    if shape_rec.record[3] == 'Andorra':
        vertices = []
        codes = []
        pts = shape_rec.shape.points
        prt = list(shape_rec.shape.parts) + [len(pts)]
```

```
        for i in range(len(prt) - 1):
            for j in range(prt[i], prt[i+1]):
                vertices.append((pts[j][0], pts[j][1]))
            codes += [Path.MOVETO]
            codes += [Path.LINETO] * (prt[i+1] - prt[i] -2)
            codes += [Path.CLOSEPOLY]
        clip = Path(vertices, codes)
        clip = PathPatch(clip, transform=ax.transData)


m = Basemap(llcrnrlon=1.4,
    llcrnrlat=42.4,
    urcrnrlon=1.77,
    urcrnrlat=42.7,
    resolution = None,
    projection = 'cyl')

ds = gdal.Open('srtm_37_04.tif')
data = ds.ReadAsArray()

gt = ds.GetGeoTransform()
x = numpy.linspace(gt[0], gt[0] + gt[1] * data.shape[1], data.shape[1])
y = numpy.linspace(gt[3], gt[3] + gt[5] * data.shape[0], data.shape[0])

xx, yy = numpy.meshgrid(x, y)

cs = m.contourf(xx,yy,data,range(0, 3600, 200))

for contour in cs.collections:
        contour.set_clip_path(clip)

plt.show()
```

Let's see each part:

Reading the shapefile and clipping

To clip the image, a Basemap path is needed. The hightlighted lines in the code are the ones that do the job.

- A Matplotlib path is made by two arrays. One with the points (called vertices in the script), and the other with the functions for every point (called codes).
- In our case, only straight lines have to be used, so there will be a MOVETO to indicate the beginning of the polygon, many LINETO to create the segments and one CLOSEPOLY for closing it
- Of course, only the polygon for Andorra has to be used. I get it from the shapefile attributes
- The prt array is for managing multipolygons, which is not the case, but the code will create correct clipping for multipolygons
- The path is created using the Path function, and then added to a PathPatch, to be able to use it as a closed polygon. Note the trasnform=ax.transData attribute. This assumes the polygon coordinates to be the ones used in the data (longitudes and latitudes in our case). More information here
- The clipping itself is made in the lines 48 and 49. For each drawn element, the method set_clip_path is applied, which erases all the parts outside the clipping object

Drawing the map

The map is drawn as usual. I have used a latlon projection, so all the values for the raster and shapefile can be used directly. If the output raster was in an other projection, the shapefile coordinates should be appended to the path using the output projection (*m(pts[j][0], pts[j][1])*).

The x and y coordinates are calculated from the GDAL geotransform, and then turned into a matrix using meshgrid

3.6 Reading WRF model data

Basemap is specially good at drawing numerical weather prediction models outputs, such as WRF. WRF is a widely used model, but most of the example can be run with other model outputs, just adapting the variable names.

At the UCAR website is possible to download a WRF sample output file

The output file descriptor (cdl) contains all the information about the model size, fields, projection, etc. For instance, we will need the projection information to project the output properly:

:CEN_LAT = 34.83158f ;

:CEN_LON = -81.02756f ;

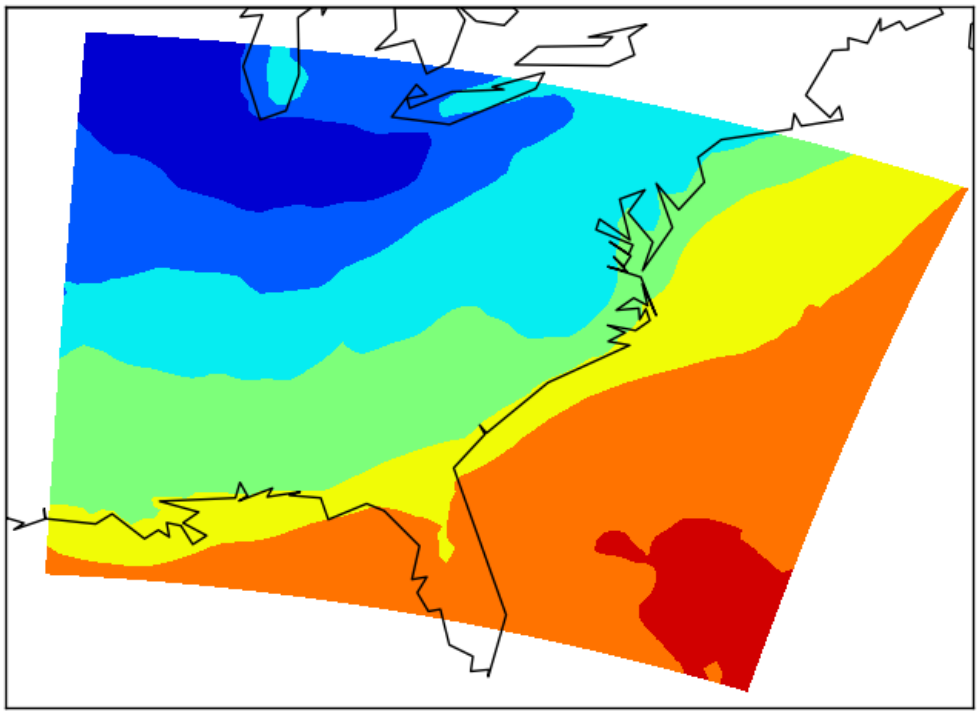
:TRUELAT1 = 30.f ;

:TRUELAT2 = 60.f ;

Note: Make sure that the gdal library installed can read NetCDF files. Check it writing *gdalinfo –formats*

3.6.1 Plotting a field as a contour

```
from osgeo import gdal
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

wrf_out_file = "wrfout_v2_Lambert.nc"

ds_lon = gdal.Open('NETCDF:"'+wrf_out_file+'":XLONG')
ds_lat = gdal.Open('NETCDF:"'+wrf_out_file+'":XLAT')
ds_t2 = gdal.Open('NETCDF:"'+wrf_out_file+'":T2')

map = Basemap(llcrnrlon=-95.,llcrnrlat=24.,urcrnrlon=-66.,urcrnrlat=45.)

map.contourf(ds_lon.ReadAsArray()[1], ds_lat.ReadAsArray()[1], ds_t2.ReadAsArray()[1])

map.drawcoastlines()

plt.show()
```



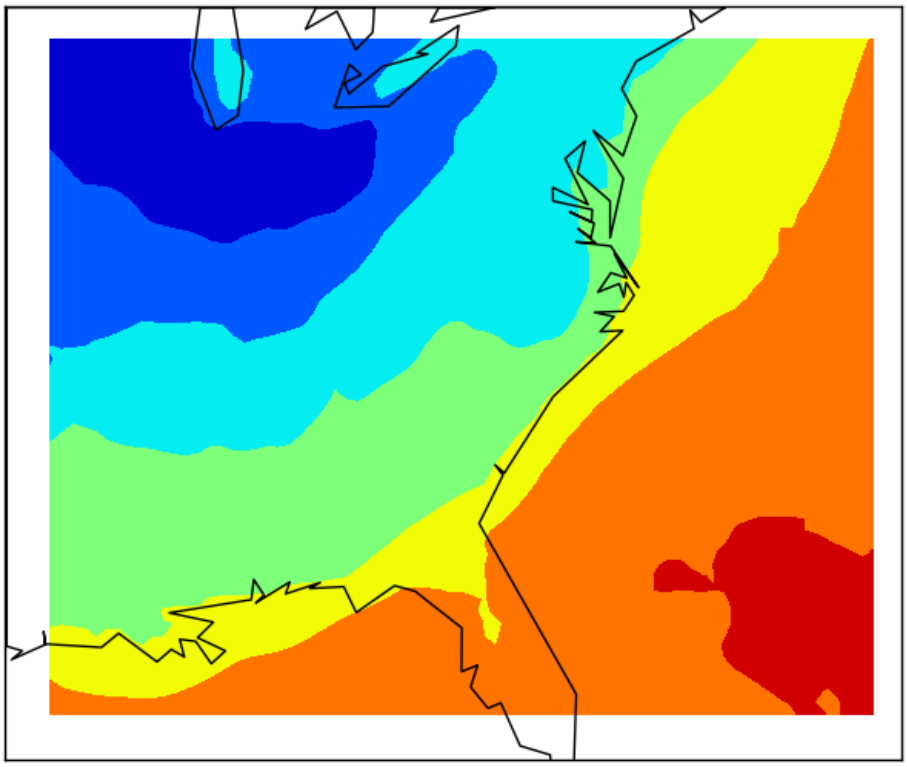
- Note how does GDAL open the files. When dealing with NetCDF files, it uses what it calls *subdatasets*, so every variable acts as an independent file
- XLONG and XLAT contain the information about the longitude and latitude of every point in the matrix. This is very useful using Basemap, since the fields needed for the methods to indicate those positions are already calculated.
- When drawing the contour, the first band from each variable is taken, since in this case, the model has several bands for longitudes, latitudes and temperature
- The strange shape of the data is because the output map is in the default Basemap projection (Equirectangular), but the model is not, so it has to be re-projected.

3.6.2 Projecting the map

As its name suggests, the model uses a Lambert conformal projection. The metadata file tells us how is this Lambert Conformal projection defined, so plotting the map is easy:

```
from osgeo import gdal
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

wrf_out_file = "wrfout_v2_Lambert.nc"

ds_lon = gdal.Open('NETCDF:"'+wrf_out_file+'":XLONG')
```

```
ds_lat = gdal.Open('NETCDF:"'+wrf_out_file+'":XLAT')

ds_t2 = gdal.Open('NETCDF:"'+wrf_out_file+'":T2')

map = Basemap(llcrnrlon=-95.,llcrnrlat=27.,urcrnrlon=-65.,urcrnrlat=40.,
              projection='lcc', lat_1=30.,lat_2=60.,lat_0=34.83158,lon_0=-98.)

x, y = map(ds_lon.ReadAsArray()[1], ds_lat.ReadAsArray()[1])

map.contourf(x, y, ds_t2.ReadAsArray()[1])

map.drawcoastlines()
plt.show()
```



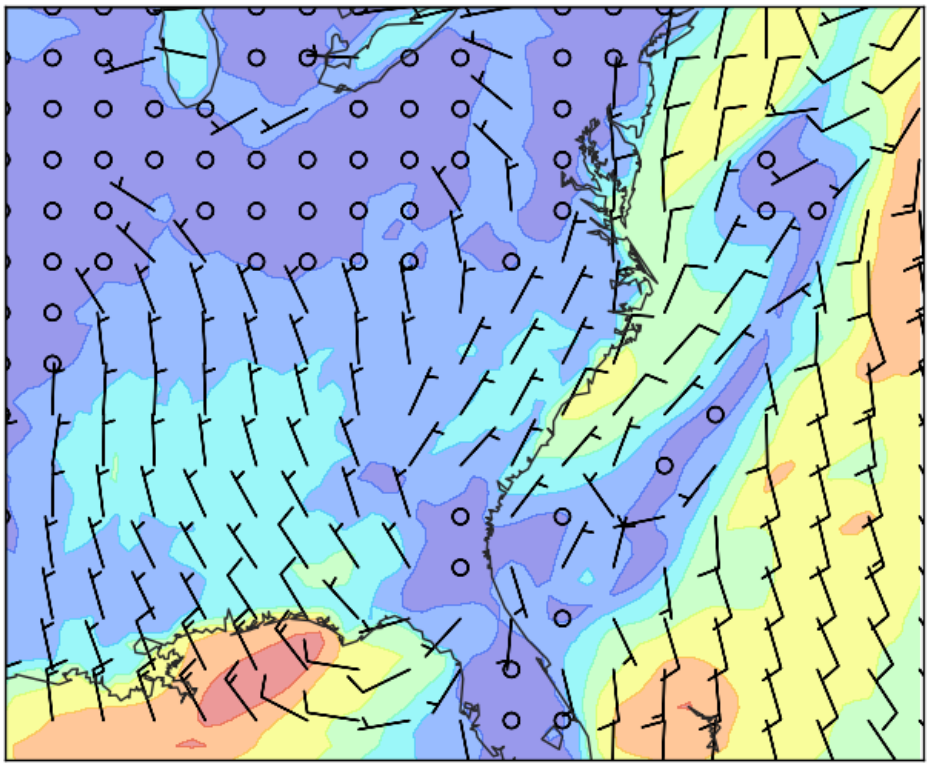
- The limits in the map don't match those of the data to show that now, the shape of the data is rectangular, so the projection is properly configured

3.6.3 Wind barbs

Basemap has a function that makes drawing wind barbs very simple (unlike using other GIS libraries, btw):

```
from osgeo import gdal
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
```

```
import numpy as np

wrf_out_file = "wrfout_v2_Lambert.nc"

ds_lon = gdal.Open('NETCDF:"'+wrf_out_file+'":XLONG')
ds_lat = gdal.Open('NETCDF:"'+wrf_out_file+'":XLAT')

ds_u = gdal.Open('NETCDF:"'+wrf_out_file+'":U10')
ds_v = gdal.Open('NETCDF:"'+wrf_out_file+'":V10')

map = Basemap(llcrnrlon=-93.7, llcrnrlat=28., urcrnrlon=-66.1, urcrnrlat=39.5,
              resolution = 'l',
              projection='lcc', lat_1=30., lat_2=60., lat_0=34.83158, lon_0=-98.)

x, y = map(ds_lon.ReadAsArray()[1], ds_lat.ReadAsArray()[1])


u = ds_u.ReadAsArray()[1]
v = ds_v.ReadAsArray()[1]

yy = np.arange(0, y.shape[0], 4)
xx = np.arange(0, x.shape[1], 4)

points = np.meshgrid(yy, xx)

map.contourf(x, y, np.sqrt(u*u + v*v), alpha = 0.4)
map.barbs(x[points], y[points], u[points], v[points])

map.drawcoastlines(color = '0.15')
plt.show()
```



- Now, the limits of the map match those of the data, so no white space is shown. The resolution parameter is also changed, to get a prettier coastline.
- **Not all the possible barbs are drawn, since the map would be difficult to understand.**
  - To eliminate some of the points, the *numpy.arange* function is used, to select the pixels to be used
  - After having to arrays with the positions, a 2d matrix is created with this values, using *numpy.meshgrid*. Now is possible to select only these points from any array, as shown when using the *barbs* method
- A contour with the wind speed is plotted under the barbs, to make the map more interesting. Note that since the fields are numpy arrays, the module is easily calculated

CHAPTER 4

Other

4.1 Running scripts from the crontab

The following script will fail when is run from the crontab:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

map = Basemap()
map.drawcoastlines()
plt.savefig('out.png')
```

The error will be something like

> RuntimeError: could not open displayX

This is because *matplotlib.pyplot* assumes that a, active X server is opened to be able to open the window with the graph when *plt.show()* is called.

To avoid the problem, simply add importing pyplot:

```
import matplotlib as mpl
mpl.use('Agg')
```

So the code will work even from the *cron* when written as:

```
from mpl_toolkits.basemap import Basemap
import matplotlib as mpl
mpl.use('Agg')
import matplotlib.pyplot as plt

map = Basemap()
map.drawcoastlines()
plt.savefig('out.png')
```

There is a post at StackOverflow explaining it better

4.2 External resources

- Official basemap documentation Some parts of the tutorial are taken directly from there.

Installation:

- The official docs
- A good blog post

Basic functions:

- Basemap page with all the marker information
- Basemap page with all the information about color codes
- Colormesh docs
- Setting colormesh levels
- **imshow**
  - Combining an image and a plot in matplotlib
  - Adding small images to basemap
  - Image rotation using imshow
  - Complex example using imshow, shiftgrid and transform_scalar
- Vector rotation without regular grids using numpy

Zoom:

- Zooming in Basemap
- Matplotlib portion zooming techniques

3D:

- mplot3d home page
- Matplotlib 3d examples (no maps yet)
- Matplotlib 3d (Axes3D) API
- StackOverflow question explaining how to fill the polygons in matplotlib3d and basemap

Clipping:

- The entry at GeoExamples
- Clipping a matplotlib path
- Creating paths using matplotlib
- Creating polygons from a shapefile
- Understanding Basemap transformations

Shapefiles:

- StackOverflow question about drawing polygons
- Patch collection example
- Inverse function for zip

Styling:

- Colors in matplotlib
- Arrow styles in matplotlib
- How Bad Is Your Colormap?
- Creating custom color maps

Colorbar:

- Colorbar official docs

Subplots:

- Nice matplotlib tutorial with a good subplots section
- GridSpec and SubplotSpec detailed explanation

Crontab:

- Generating matplotlib graphs without a running X server

Utilities:

- Alternative way to get is_land functionality, much faster
- Example of maskoceans usage
- Good example of interp usage

External examples:

- Drawing heat maps in Basemap
- Another Basemap tutorial

Other python/GIS resources:

- Python GDAL/OGR Cookbook
- Huge set of Matplotlib examples