EASC2410 Lecture 19

Introduction to 2D Visualization using Python: Pseudocolor, contour, gridding on maps

Dr. Binzheng Zhang
Department of Earth Sciences

Review of Lecture 18:

- Pandas
  - Merge data frames together based on column index
- Geopandas
  - Map values onto a map using geodataframe

In Lecture 19, you will learn:

- Concepts: grid mesh, 2-D pseudocolor plots, contours
- Python tricks: matplotlib 2-D visualization on maps

Recall 2-D NumPy Arrays

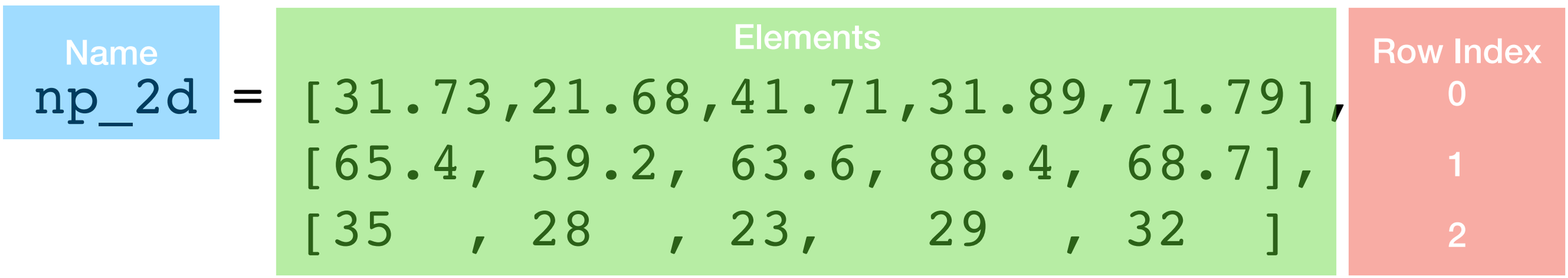
Remember the np.array() function for generating 1-D NumPy arrays, we can use that to generate 2-D arrays as well

```
1  import numpy as np # load the numpy module
2
3  # define a 2-D numpy array using the .array() method
4  np_2d = np.array([[31.73, 21.68, 41.71, 31.89, 71.79],
5                    [65.4, 59.2, 63.6, 88.4, 68.7],
6                    [35,   28  , 23  , 29  , 32  ]])
7
8  print(np_2d) # print the results
9  print(np_2d.shape)
10 print(np_2d.size)
```

outputs:

```
[[31.73 21.68 41.71 31.89 71.79]
 [65.4  59.2  63.6  88.4  68.7 ]
 [35.   28.   23.   29.   32.  ]]
(3, 5)
15
```

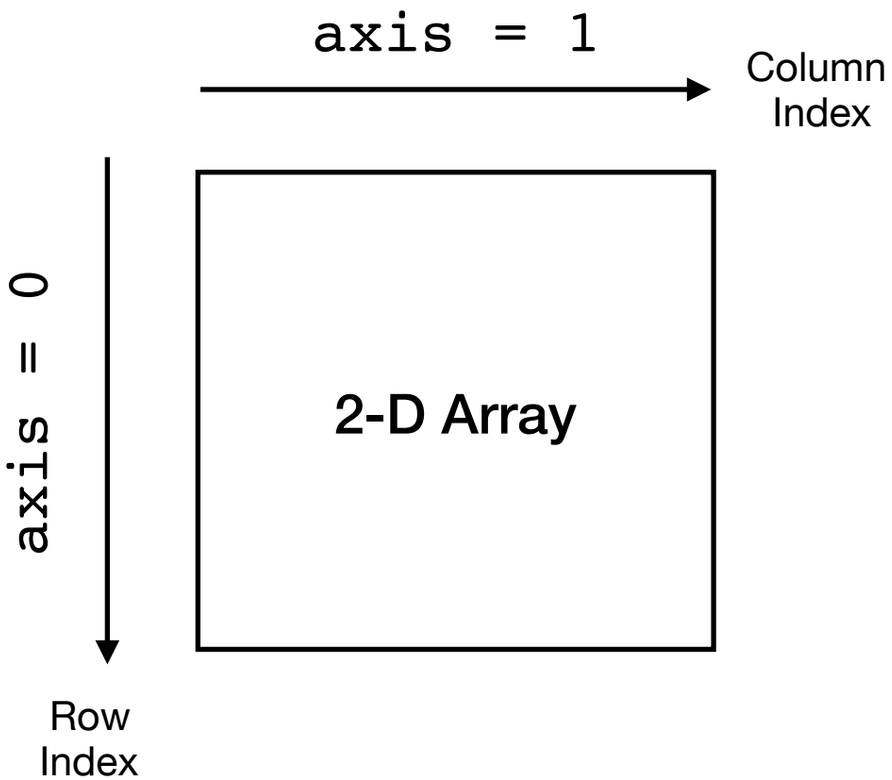
2-D Arrays:



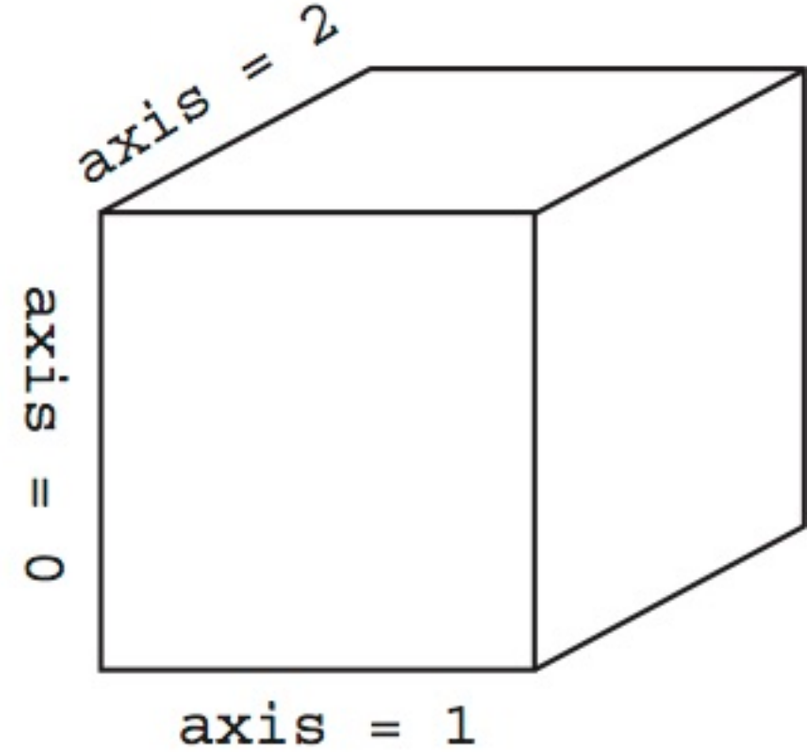
three rows, five columns

Recall: 2-D NumPy Arrays - Axis

Axises of arrays:



2-D Arrays



3-D Arrays

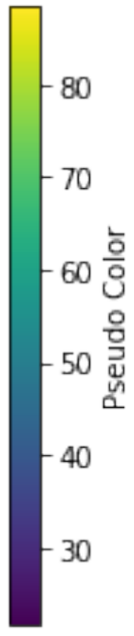
Color a 2-D NumPy Array

The 2-d NumPy Array

row index					
0	31.73	21.68	41.71	31.89	71.79
1	65.4	59.2	63.6	88.4	68.7
2	35	28	23	29	32
	0	1	2	3	4

column index

Colormap



A “Pseudo Color” plot for the array

Let’s assign color for each element in the array

Differences between **True Color**, **False Color** and **Pseudo Color**

True Color: the color you get with your typical home camera, or what you would see with your own eyes

False Color: colors that are shown are not what you would see if you were to look at the object with your own two eyes; e.g., filters being used to the picture

Pseudo Color: Pseudocolor begins with a single band of data or data from one single filter. This means that for a single pixel, there are 256 possible values. The next step is to take each of these values and map them to a specific color thus creating a color image.

How to create Pseudo Color plots for 2-D NumPy arrays?

Need two things:

1. **locations form a grid (mesh)**
2. **data for each grid location**

How to create a 2-D grid?

```
1  x = np.arange(-1.0, 1.1, 0.1) # make a 1D array from -1.0 to 1.0 with a spacing of .1
2  y = np.arange(0.0, 1.1, 0.1) # ditto
3  xx, yy = np.meshgrid(x, y) # make a meshgrid
4  print(x.shape)
5  print(y.shape)
6  print(xx.shape)
7  print(yy.shape)
```

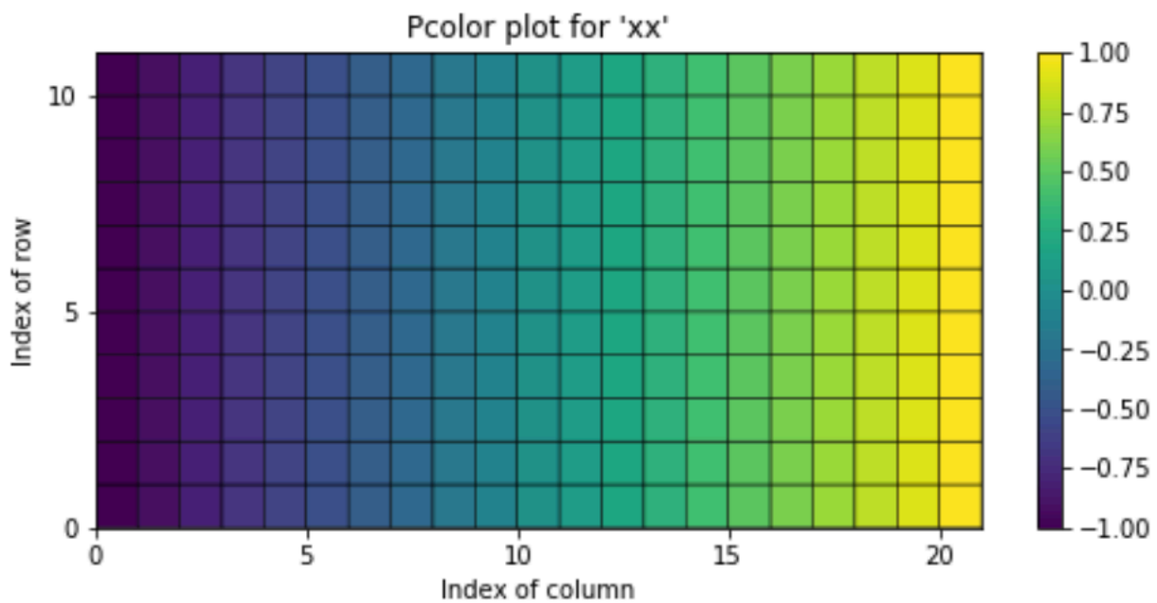
```
(21,)
(11,)
(11, 21)
(11, 21)
```

xx here is a 2-D NumPy Array contains all the x locations of the grid

Take a look at the xx array using the pcolor() function

```
1  plt.figure(figsize=(18,3.6))
2  plt.subplot(1,2,1)
3  plt.pcolor(xx, edgecolors='k', linewidths=0.5)
4  plt.xticks(np.arange(0,21,5))
5  plt.yticks(np.arange(0,11,5))
6  plt.xlabel('Index of column')
7  plt.ylabel('Index of row')
8  plt.title("Pcolor plot for 'xx'")
9  plt.colorbar()
```

Syntax: **plt.pcolor(x)**



How to create Pseudo Color plots for 2-D NumPy arrays?

Need two things:

1. **locations form a grid (mesh)**
2. **data for each grid location**

How to create a 2-D grid?

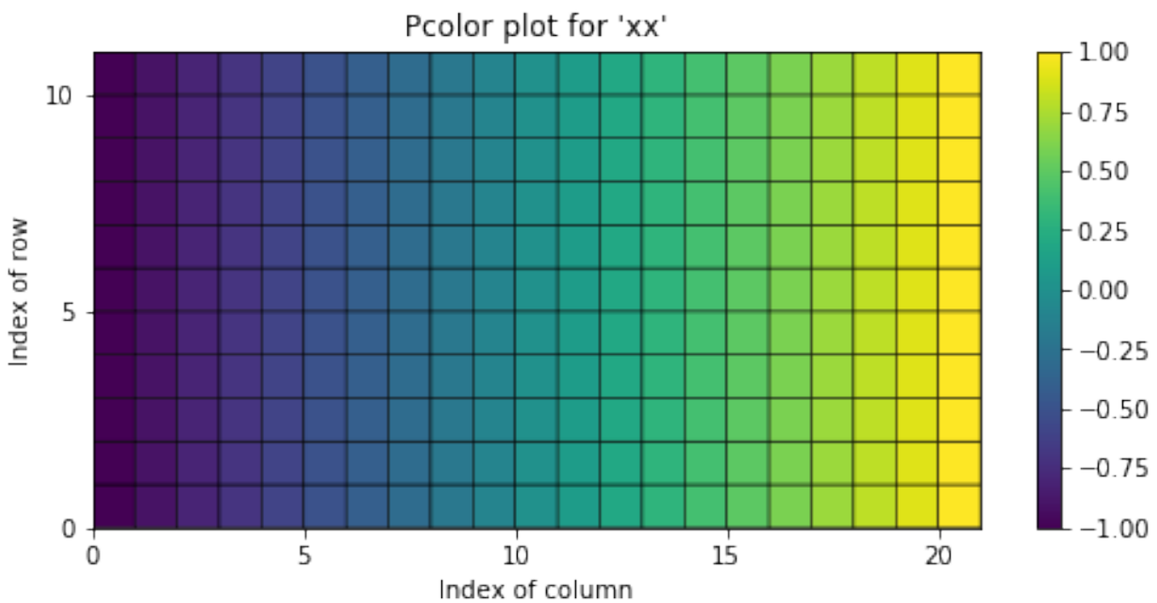
```
1  x = np.arange(-1.0, 1.1, 0.1) # make a 1D array from -1.0 to 1.0 with a spacing of .1
2  y = np.arange(0.0, 1.1, 0.1) # ditto
3  xx, yy = np.meshgrid(x, y) # make a meshgrid
4  print(x.shape)
5  print(y.shape)
6  print(xx.shape)
7  print(yy.shape)
```

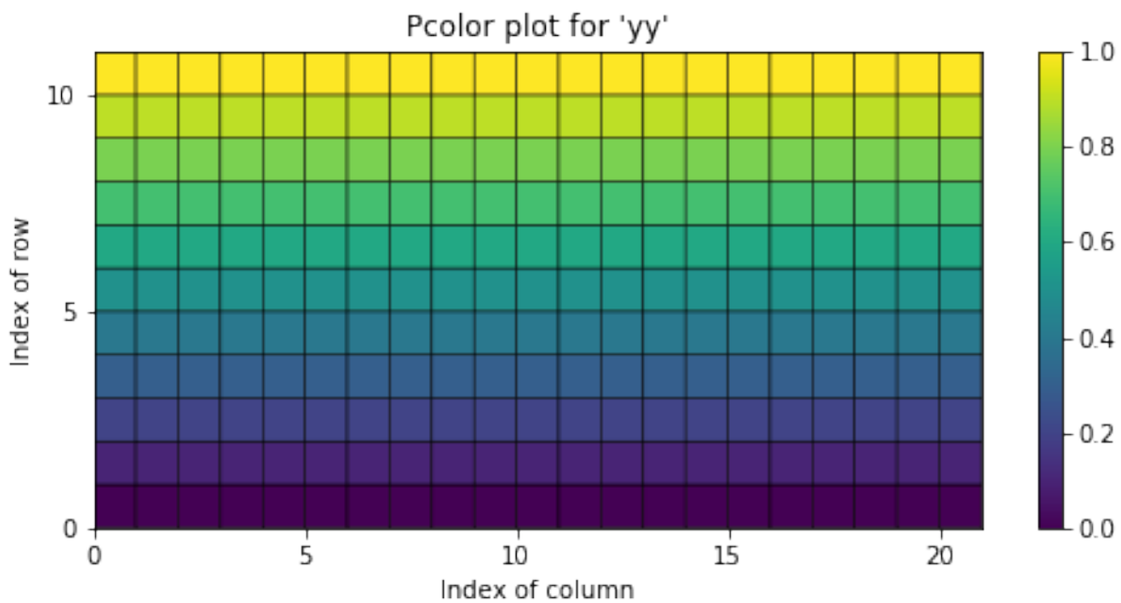
```
(21,)
(11,)
(11, 21)
(11, 21)
```

Syntax:

- meshgrid(x,y)
- x,y are 1-D NumPy Arrays

Take a look at the (xx, yy) grid





How to create Pseudo Color plots for 2-D NumPy arrays?

Need two things:

1. **locations form a grid (mesh)**
2. **data for each grid location**

How to create data on the 2-D grid (xx,yy)?

Use NumPy element-wise operations, we can generate data as a function of the 2-D grid (xx,yy) we just created using the meshgrid() function. For example:

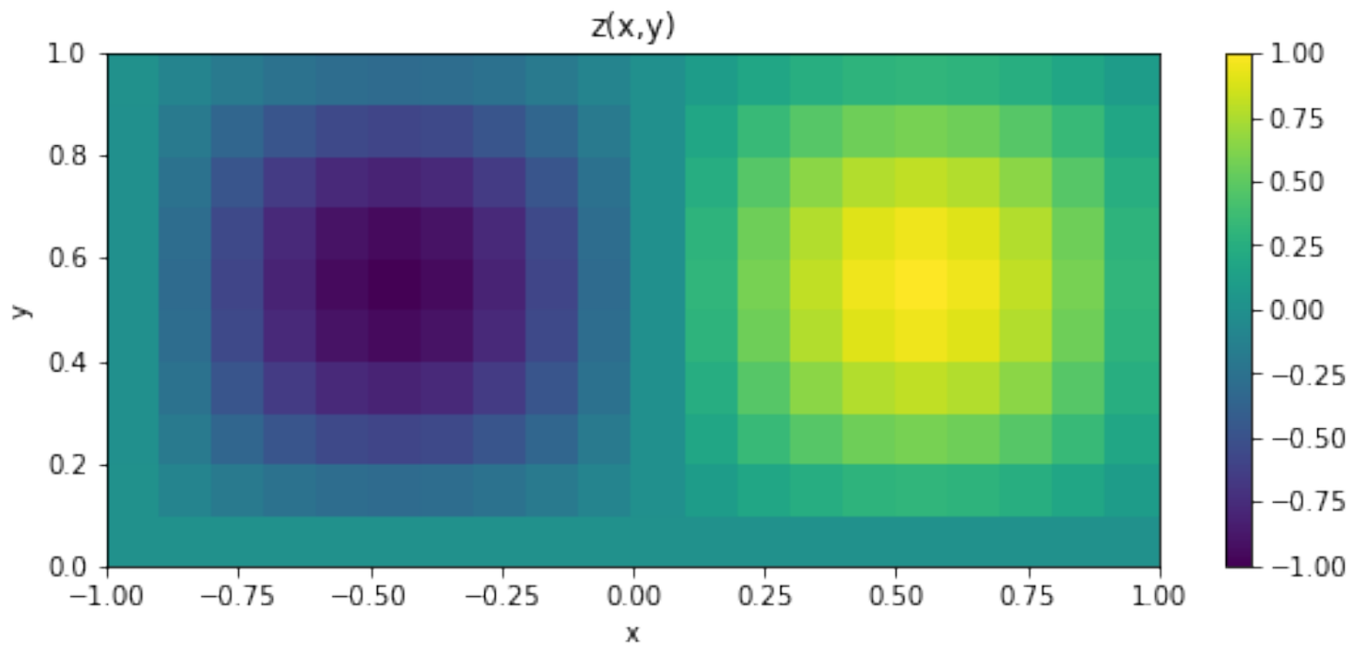
$$z(x, y) = \sin(\pi x)\sin(\pi y)$$

In Python, it is simply

```
1 z = np.sin(xx*np.pi)*np.sin(yy*np.pi)
```

Now let's use the pcolor() function to visualize the new array z:

```
2 plt.pcolor(xx,yy,z)
3 plt.xlabel('x')
4 plt.ylabel('y')
5 plt.title('z(x,y)')
6 plt.colorbar()
```

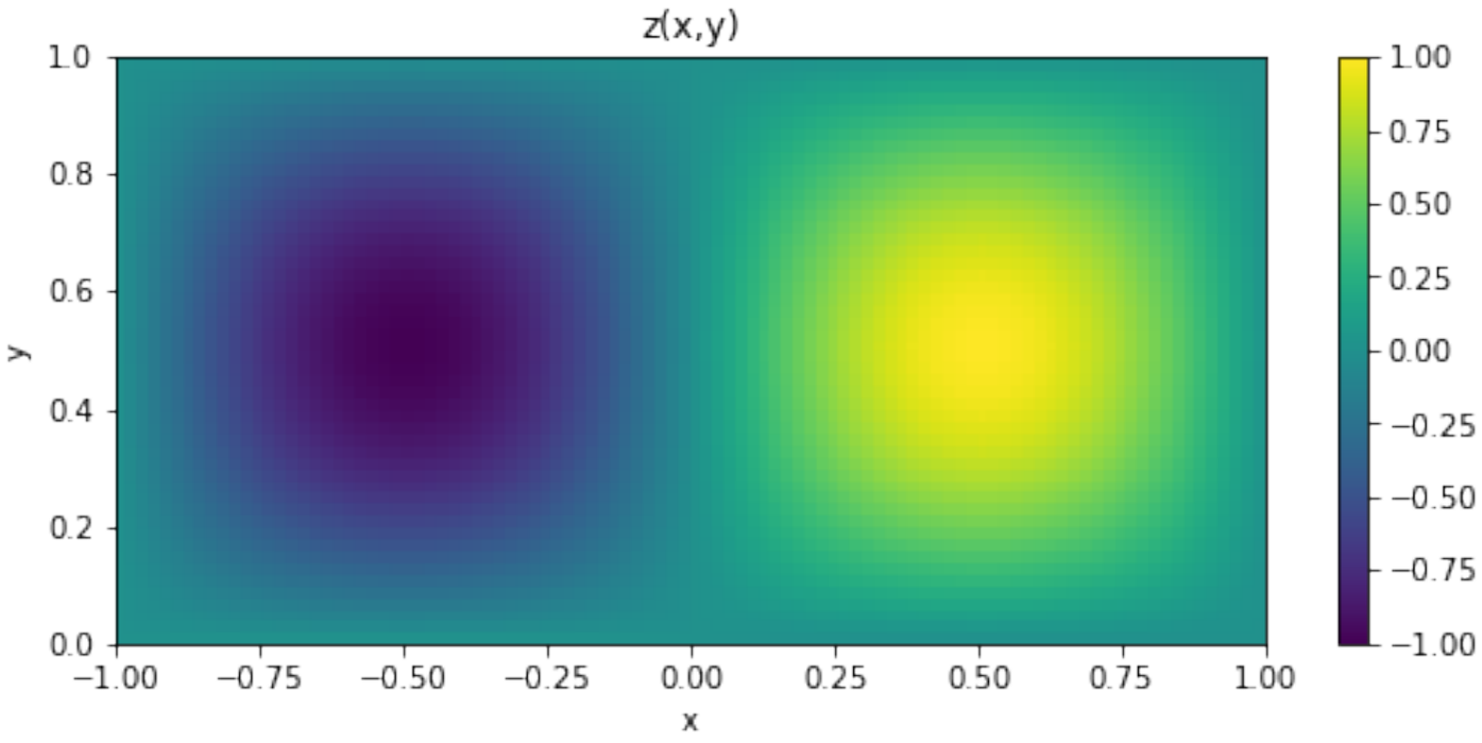


Syntax: **plt.pcolor(x, y, z)**

x,y,z are 2-D NumPy arrays with the **same size**

Try a high-resolution figure using more grid points

```
x = np.arange(-1.0, 1.02, 0.02) # make a 1D array from -1.0 to 1.0 with a spacing of
y = np.arange(0.0, 1.02, 0.02) # ditto
xx, yy = np.meshgrid(x, y) # make a meshgrid
z = np.sin(xx*np.pi)*np.sin(yy*np.pi)

plt.figure(figsize=(8.6,3.6))
plt.pcolor(xx,yy,z,vmin=-1,vmax=1)
plt.xlabel('x')
plt.ylabel('y')
plt.title('z(x,y)')

plt.colorbar()
```



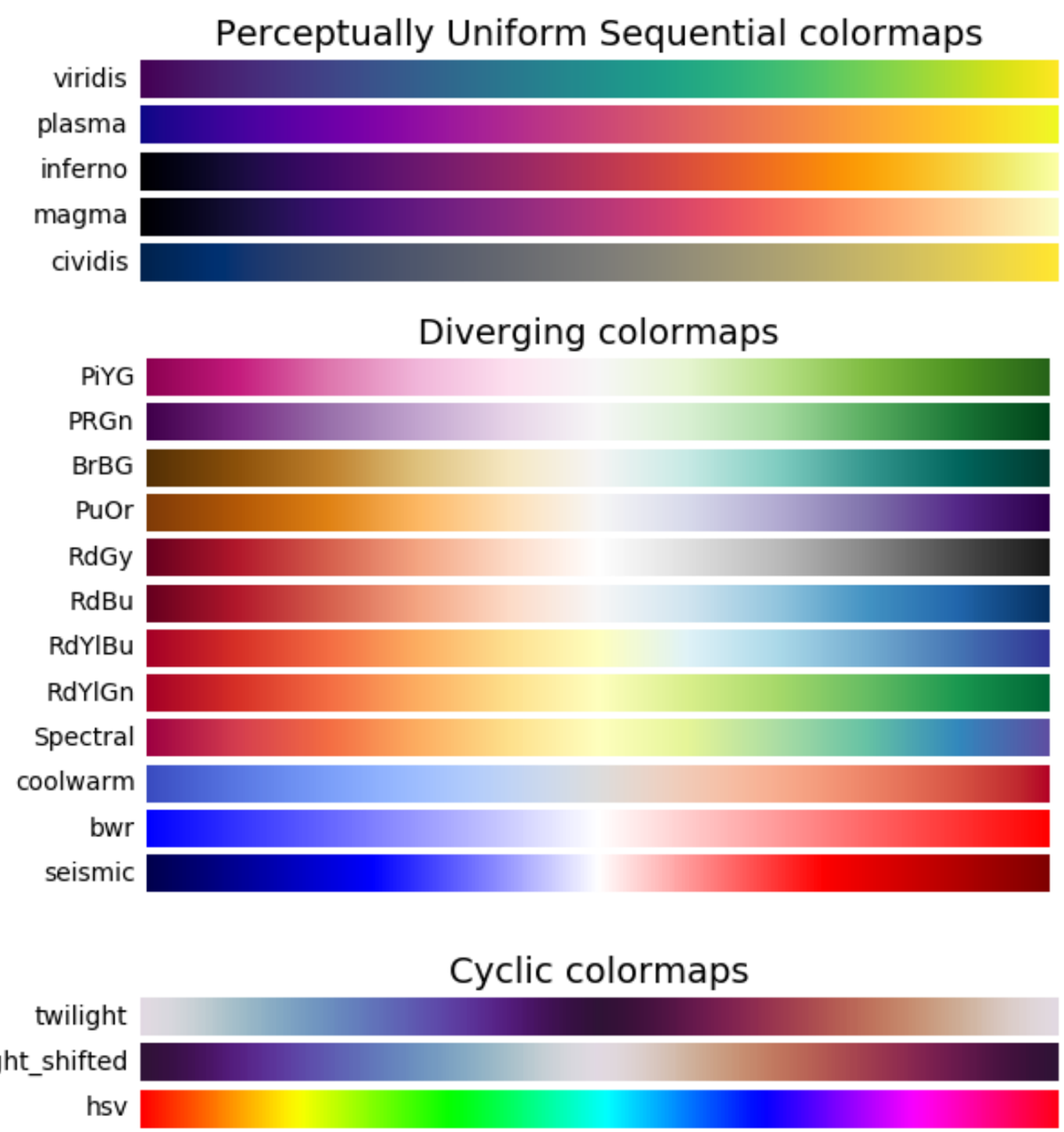
Notes:

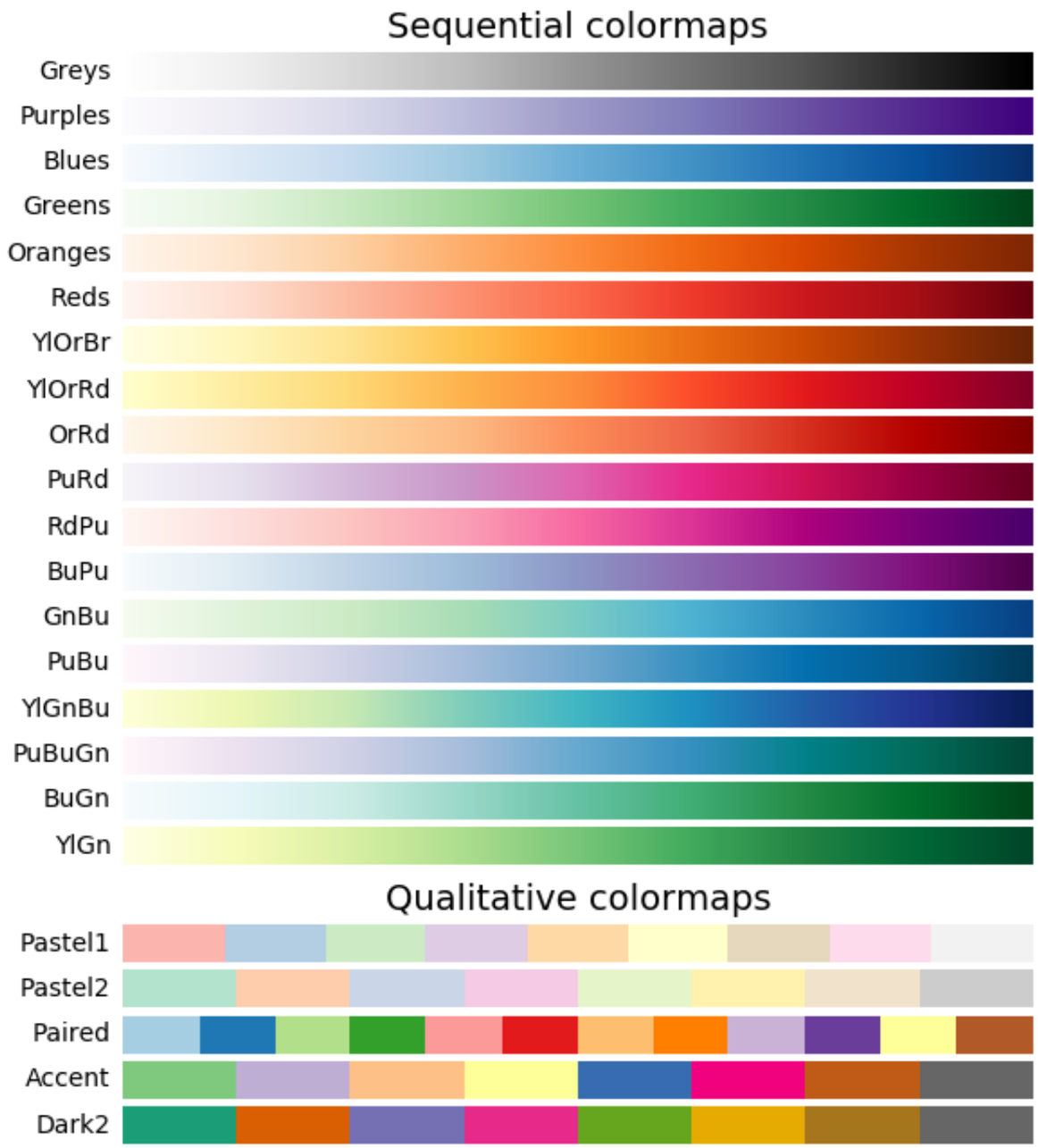
- now it's a figure with 100x50 grid pints
- default colormap is "viridis"
- vmin, vmax set the limits of the color
- colorbar( ) function displays the color scale

Colormaps

Now we can choose different colormaps. Here's a more detailed documentation about Python colormaps: https://matplotlib.org/users/colormaps.html. The default one is called "viridis", other commonly used ones are:

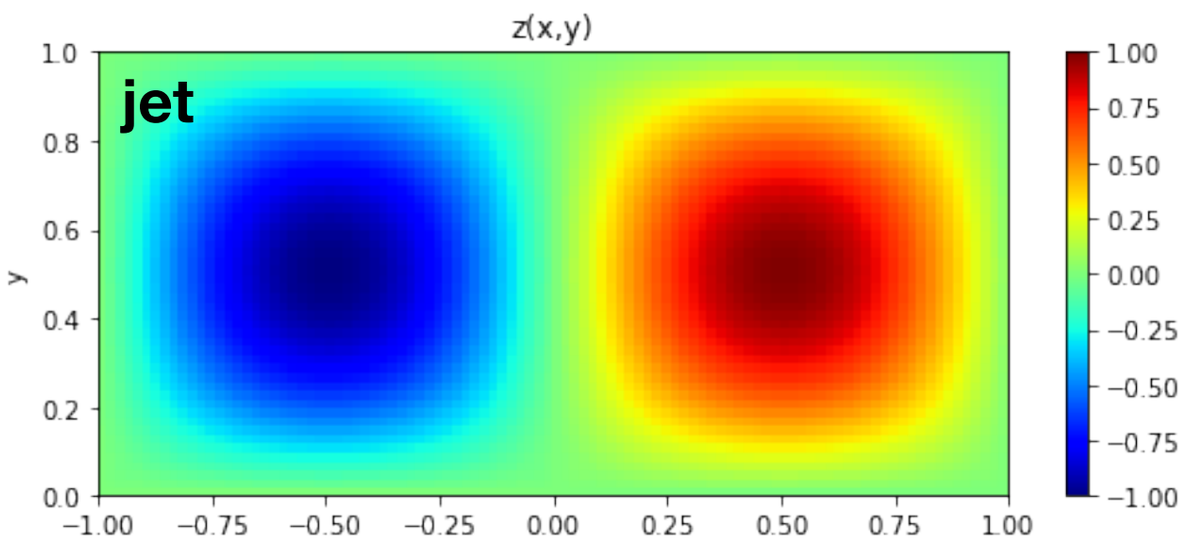
- jet (be careful with this one), hoT, bwr

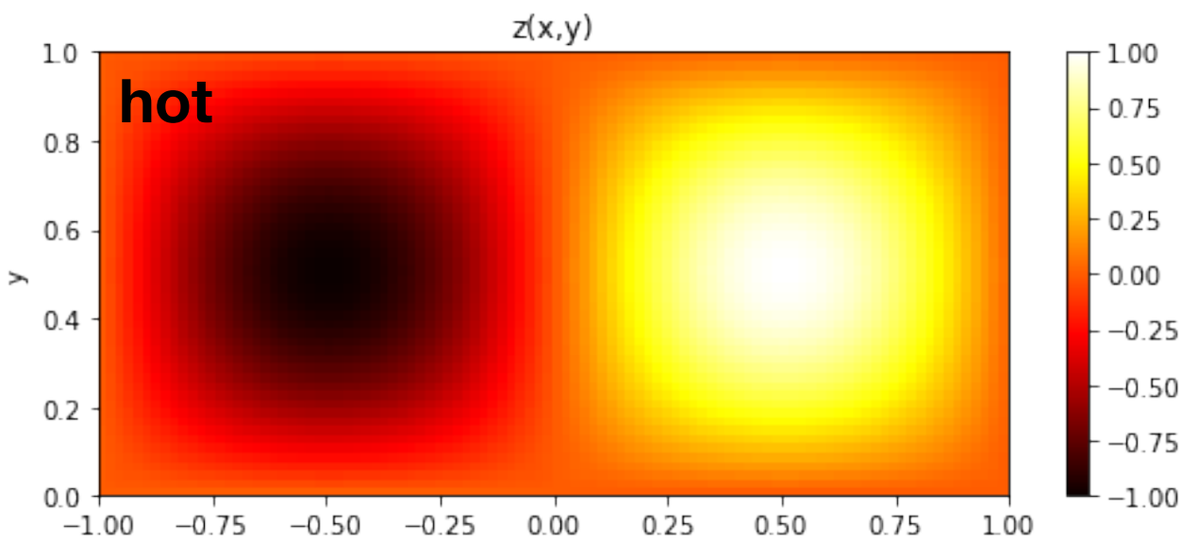


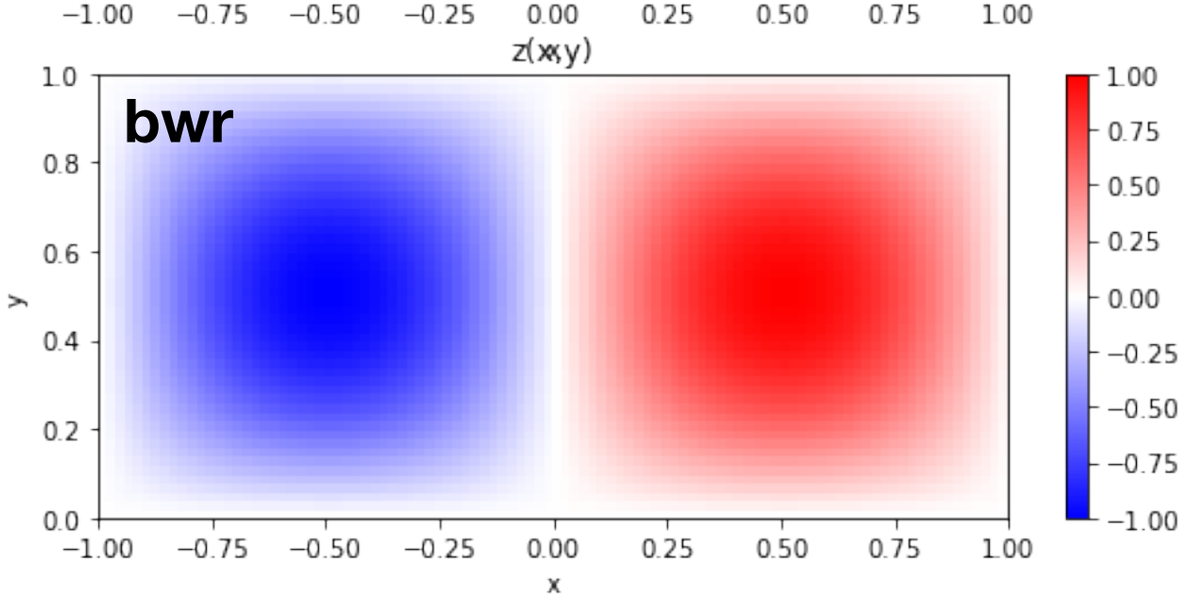


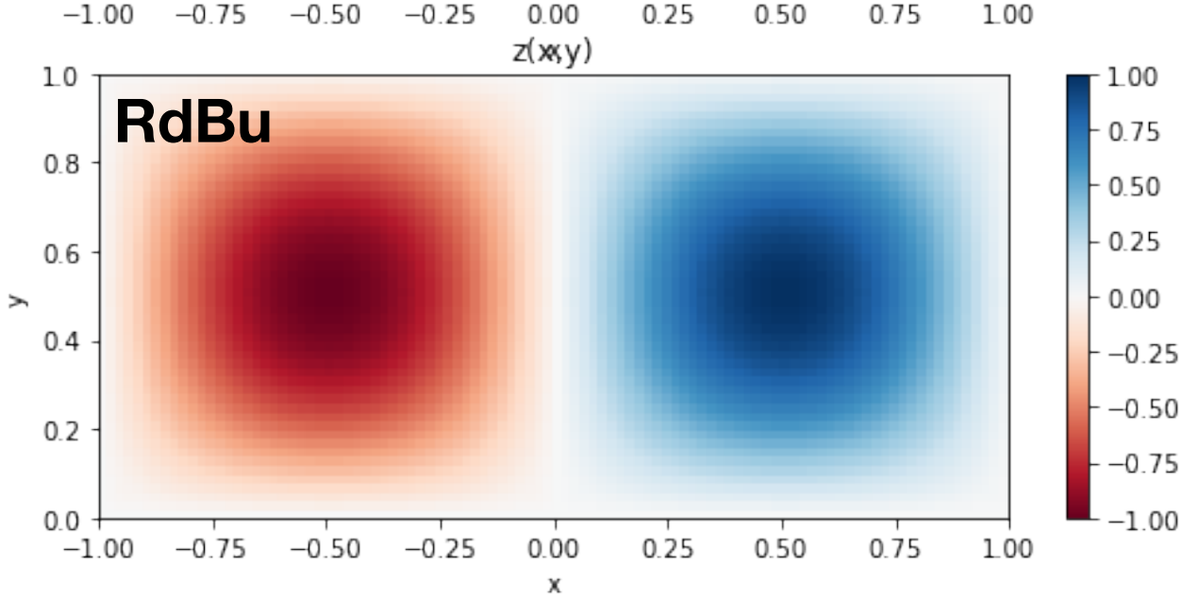
Colormaps

Figures may look quite differently with different colormaps. Here are four examples of some commonly used colormaps when doing 2-D visualization:





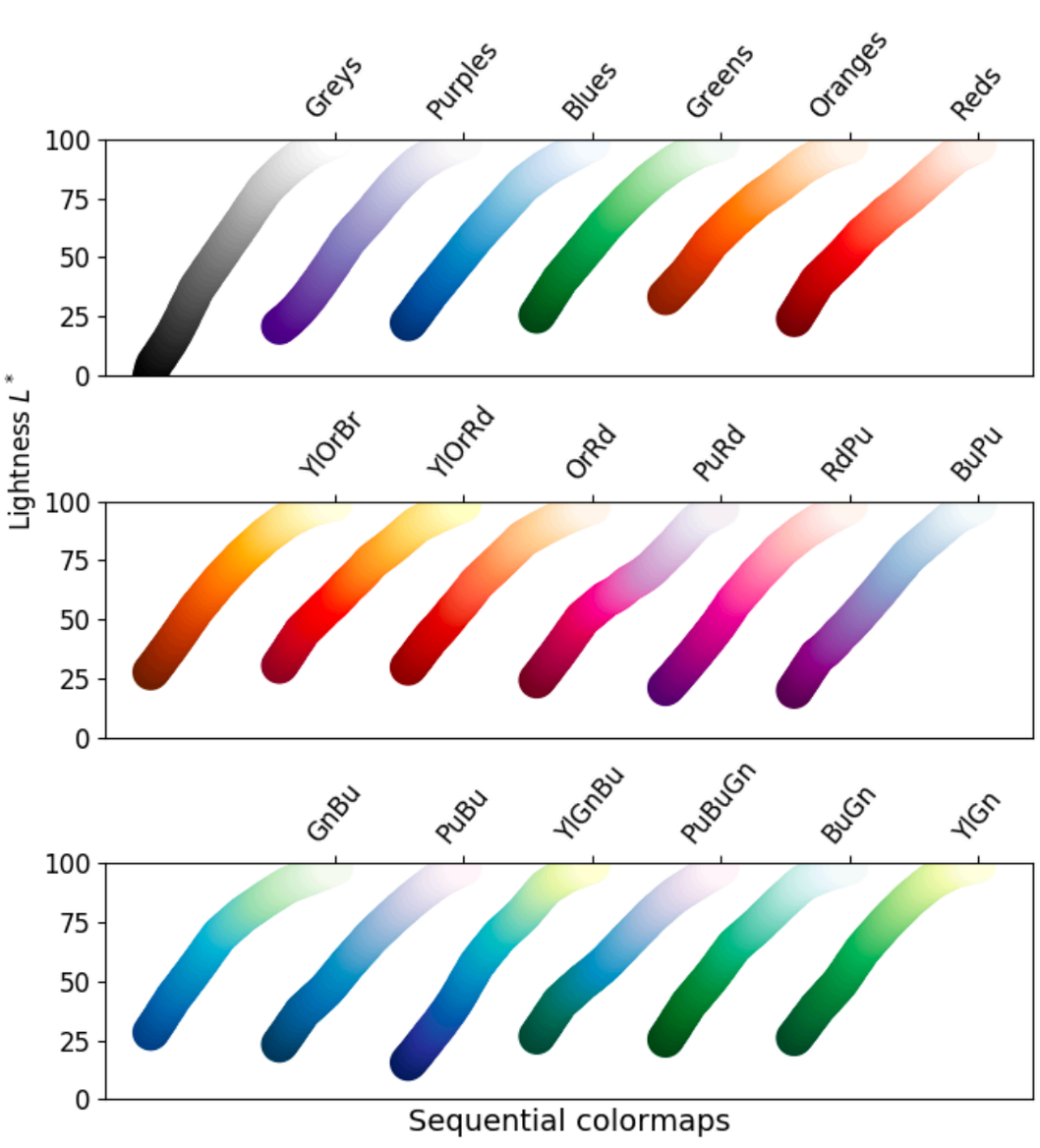


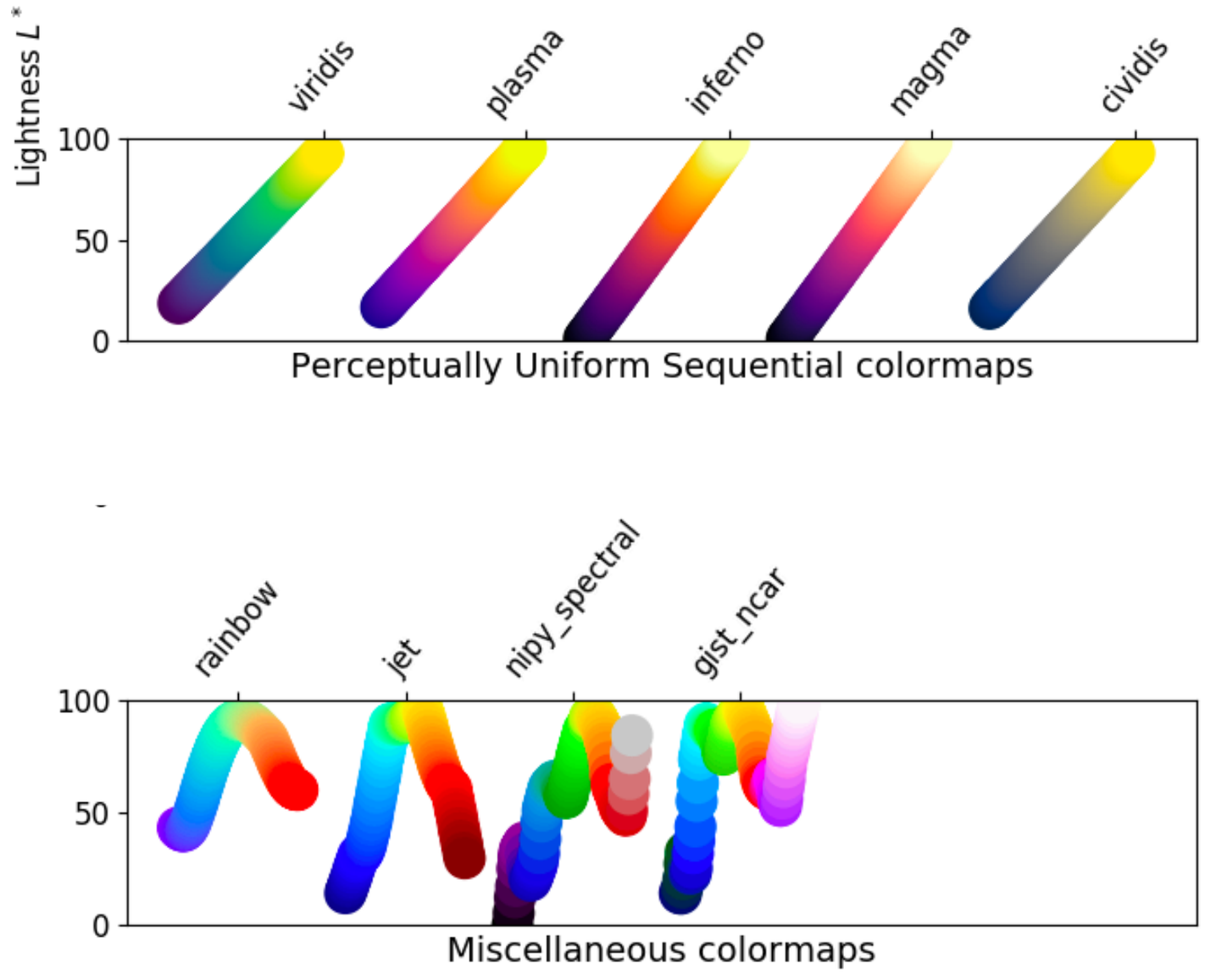


Question: Which one do you think is the most suitable colormap for the data z?

My personal preference: for signed data (positive and negative values), use blue-white-red
for non-signed data (viridis, plasma)

Be careful with the Rainbow (jet) colormap



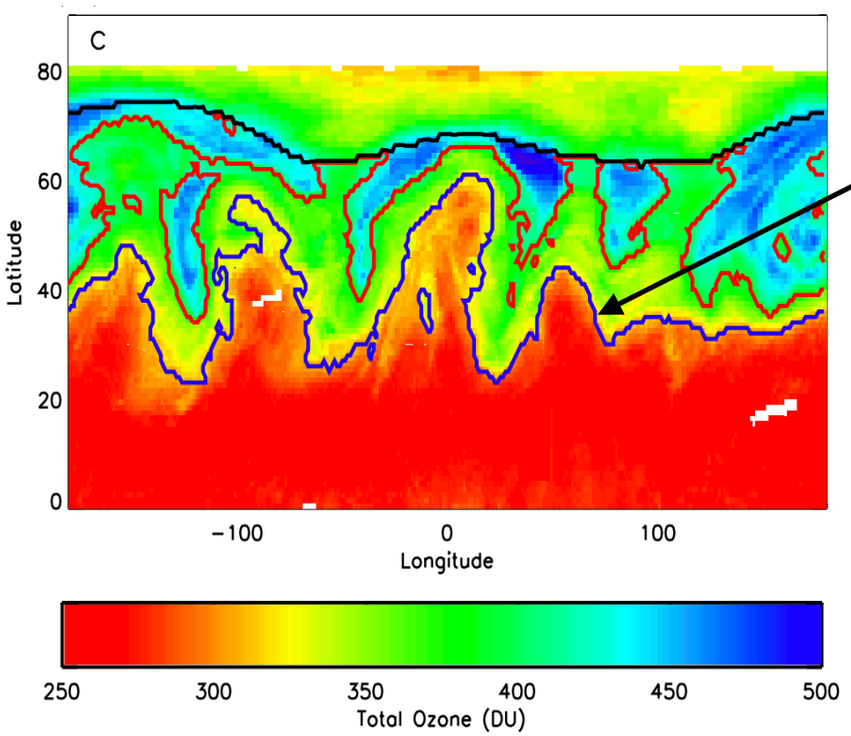


Problems of a rainbow colormap:

As it is explain in several visualisation articles the rainbow color map can be misleading when visualising data. The main issues with this kind of color map are:

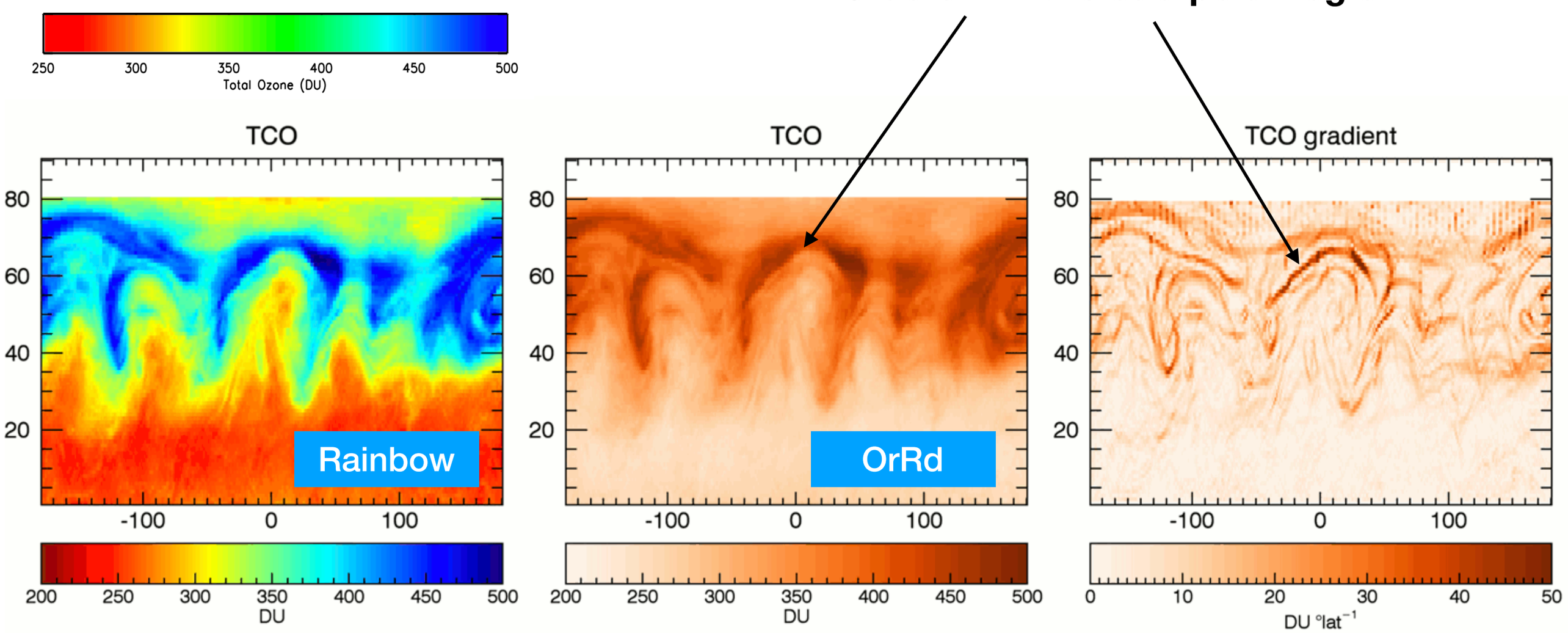
1. It is confusing because it doesn't have a monotonic perceptual ordering,
2. structures in the data can be hidden, since not all data variations are represented visually,
3. it introduces gradients not related to the data,
4. it artificially divides the data into a small number of categories, one for each color.

Be careful with the Rainbow (jet) colormap



TCO Gradient in the sub-tropical region

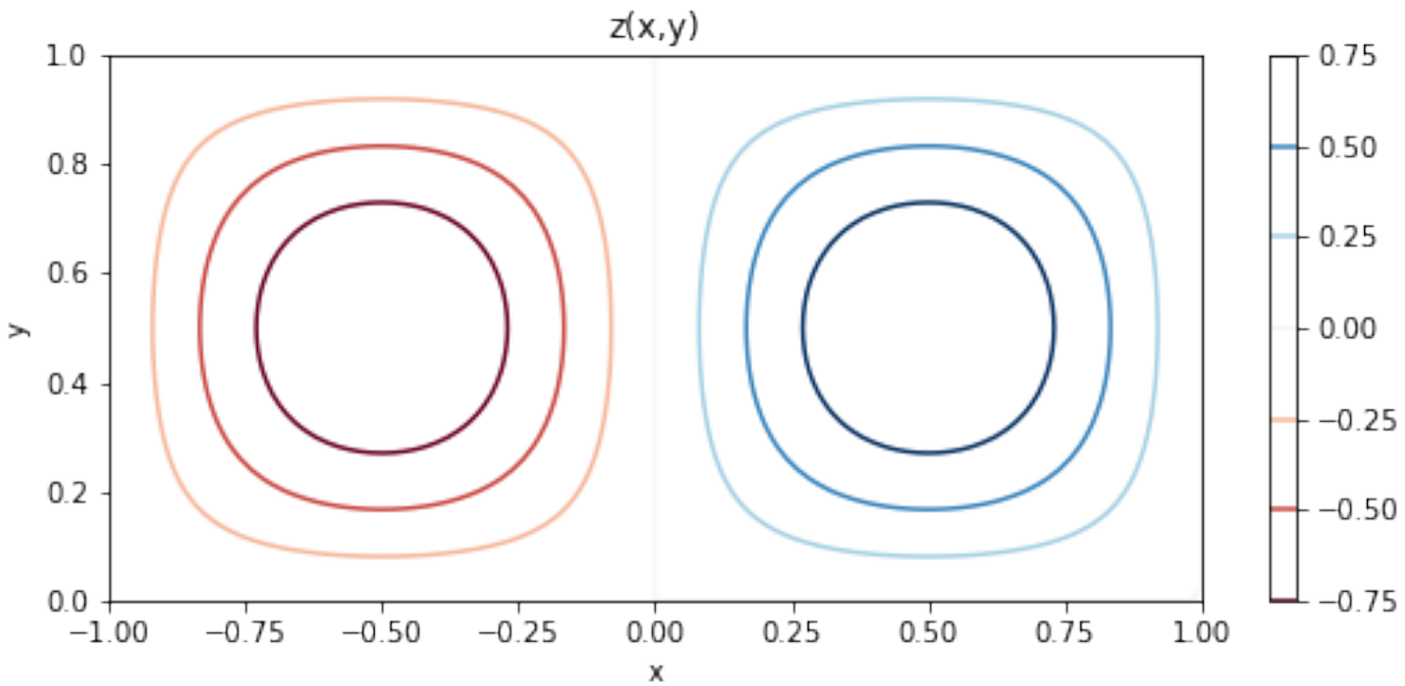
Gradient in the sub-polar region



How to create Contour plots for 2-D NumPy arrays?

The pcolor( ) function gives you pixel-like color plots by coloring each rectangular grid with color corresponding to its value in the colormap. Another useful way to visualize 2-D data is using **contours**:

```
x = np.arange(-1.0, 1.02, 0.02) # make a 1D array from -1.0 to 1.0 with a spacing of .02
y = np.arange(0.0, 1.02, 0.02) # ditto
xx, yy = np.meshgrid(x, y) # make a meshgrid
z = np.sin(xx*np.pi)*np.sin(yy*np.pi)

plt.figure(figsize=(8.8,3.6))
h = plt.contour(xx,yy,z,cmap='RdBu') # plot the contours
plt.axis('equal');
plt.colorbar()
plt.xlabel('x')
plt.ylabel('y')
plt.title('z(x,y)')
```



Syntax:

- contour(x,y,z,*Options*)

Notes:

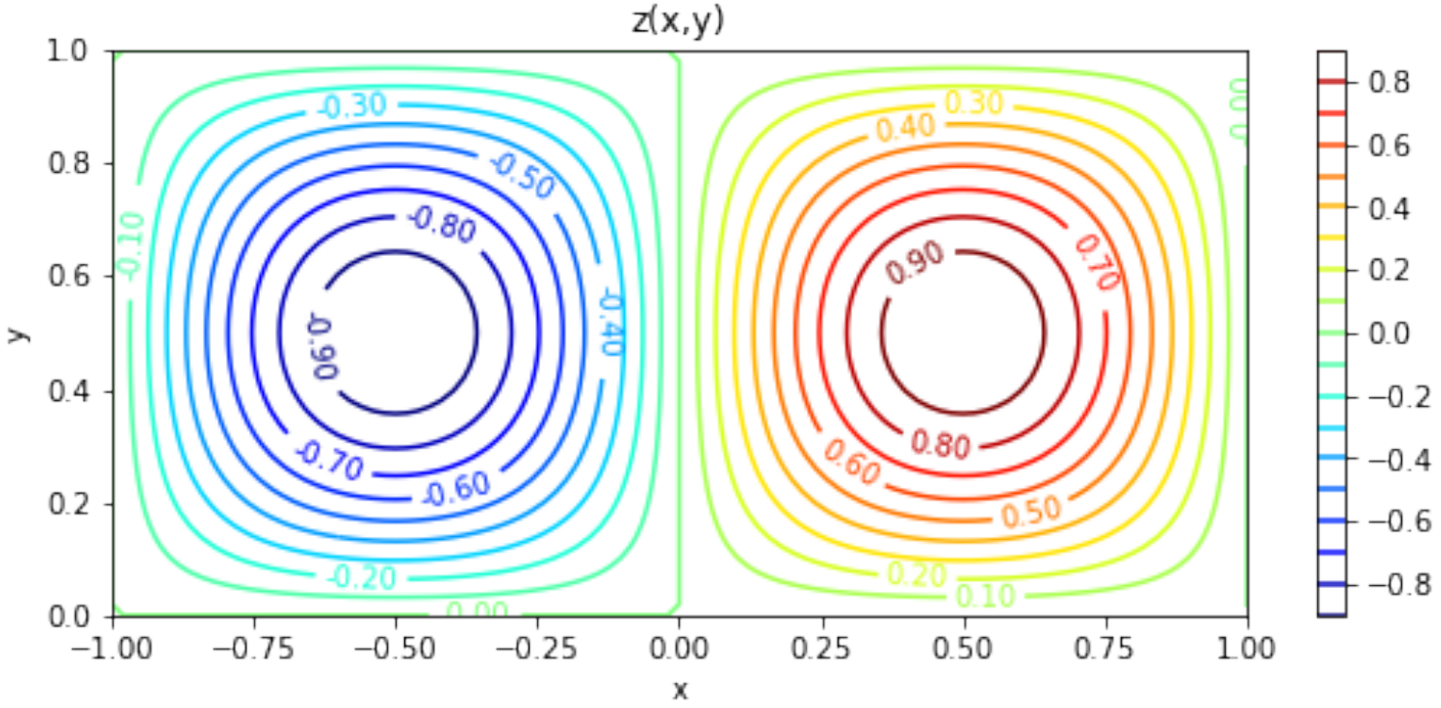
- now it's a figure with 7 contours
- used colormap "RdBu"
- colorbar( ) function displays the color scale, in a contour plot, the scale is discrete
- The **MOST** important option in a contour plot is specifying the **contour levels**

How to specify Contour levels in contour( )

Syntax:

- contour(x,y,z,*levels = Your_Specified_Contour_Levels, Other_Options*)

```
plt.figure(figsize=(8.8,3.6))
fig = plt.contour(xx,yy,z,levels=np.arange(-1.0,1.0,0.1),cmap='jet')
plt.axis('equal')
plt.clabel(fig, inline=1, fontsize=10,fmt='%3.2f');
plt.colorbar()
plt.xlabel('x')
plt.ylabel('y')
plt.title('z(x,y)')
```



Notes:

- now it's a figure with 19 contours specified by the numpy array np.arange(-1.0, 1.0, 0.1)
- used colormap "jet" (is it bad?)
- colorbar( ) function displays the color scale, in a contour plot, the scale is discrete
- clabel( ) function put contour levels on each contour - sometimes it could be too much!
- axis('equal') function sets the aspect ratio of the plot to be 1.0

You can also fill colors between contour levels using contourf( )

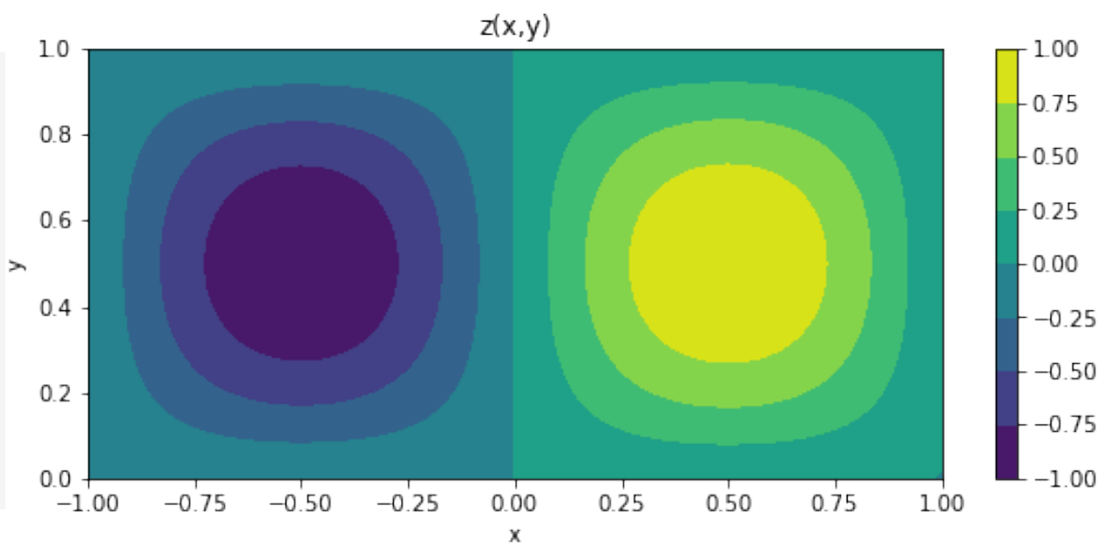
Contour plots are all well and good, but if you were after a **continuous gradation** of color, not the contour lines, so for that we use the function **contourf** instead.

Syntax:

- contourf(x,y,z,*levels = Your_Specified_Contour_Levels, Other_Options*)

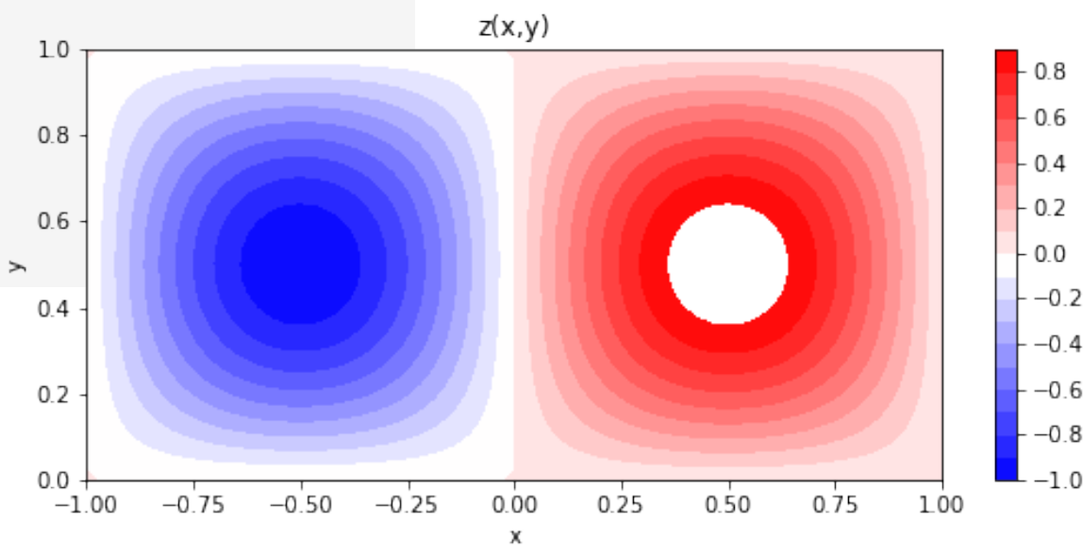
Again, default = lame

```
plt.figure(figsize=(9.6*2,3.6))

plt.subplot(1,2,1)
fig = plt.contourf(xx,yy,z) # all default options
plt.axis('equal') # this makes the axes square
plt.colorbar()
plt.xlabel('x')
plt.ylabel('y')
plt.title('z(x,y)')
```



Let's specify contour levels in the contourf() function (using the "bwr" colormap, why?):

```
plt.subplot(1,2,2) # more options in contourf()
fig = plt.contourf(xx,yy,z,levels=np.arange(-1.0,1.0,0.1),cmap='bwr')
plt.axis('equal') # this makes the axes square
plt.axis('equal')
plt.colorbar()
plt.xlabel('x')
plt.ylabel('y')
plt.title('z(x,y)')
```



Question: Why there is a "hole" in the center of the red cell?

Put everything together: Elevation data on maps

So, let's put everything together - plot data on a map. For example, we want to plot the contoured elevation data onto an orthographic projection. First, we will generate a meshgrid using **np.meshgrid** from numpy and the method **contourf** of our map object m defined above. The method **m.contourf** just works like matplotlib we just learned.

Recall: Basemap projections

```
from mpl_toolkits.basemap import Basemap

# lon_0, lat_0 are the center point of the projection.
plt.figure(1,(6,6)) # make the figure instance with a size of 5x5
m = Basemap(projection='ortho',lon_0=-75,lat_0=42) # make an orthographic projection map
m.drawcoastlines() # put on the coastlines
# draw parallels and meridians.
m.drawparallels(np.arange(-90.,120.,30.))
m.drawmeridians(np.arange(0.,420.,60.));
```

How to put data on maps:

- STEP 1: Generate a map **m** with the desired projection
- STEP 2: Generate (or load) a longitude-latitude grid
- STEP 3: Generate (or load) data values on the grid
- STEP 4: map the longitude-latitude grid to map coordinates
- STEP 5: use **m**.contourf( ) function to visualize the data
- STEP 6: adjust color scale, add labels, show the colorbar

Understand the Etopo dataset

```
1 etopo=np.loadtxt('Datasets/Etopo/etopo20data.gz')
2 elons=np.loadtxt('Datasets/Etopo/etopo20lons.gz')
3 elats=np.loadtxt('Datasets/Etopo/etopo20lats.gz')
4 print (etopo.shape)
5 print (elons.shape)
6 print (elats.shape)
```

```
(540, 1081)
(1081,)
(540,)
```

elats is a 1D array with 540 latitudinal bins (this is a long and skinny array- 540 x 1).

elons is a 1D array with 1081 longitudinal bins (this as a fat and wide array- 1 x 1081).

And **etopo** is a 2D array with 540 rows and 1081 columns (540 x 1080).

So **etopo** has an elevation for each lat/lon cell.

In order to plot the elevation data onto a lat/lon grid, we have to first make a **2-D grid** (mesh) out of the 1D arrays of elats and elons. We use the **numpy** function **meshgrid( )** you've just learned for this.

Put everything together: plot data on maps

STEP 1: generate a map using Basemap

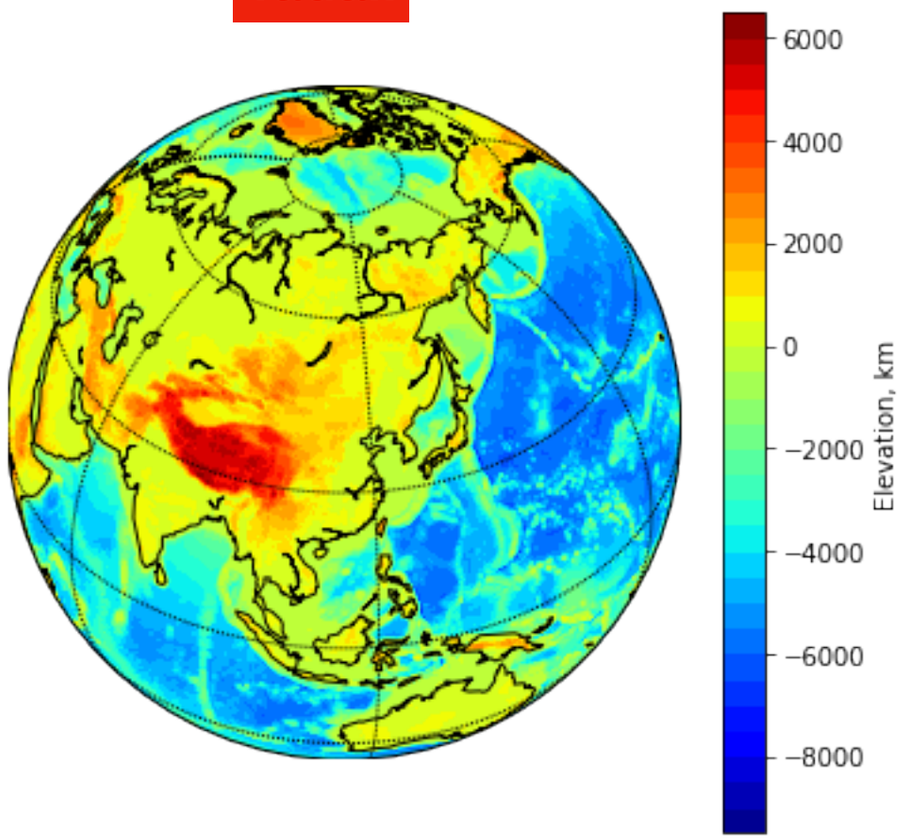
```
3 plt.figure(1,(6,6)) # make the figure instance with a size of 5x5
4 m = Basemap(projection='ortho',lon_0=114,lat_0=42) # make an orthographic
5 m.drawcoastlines() # put on the coastlines
```

STEP 2: Load a longitude, latitude data from file

```
1 etopo=np.loadtxt('Datasets/Etopo/etopo20data.gz')
2 elons=np.loadtxt('Datasets/Etopo/etopo20lons.gz')
3 elats=np.loadtxt('Datasets/Etopo/etopo20lats.gz')
4 print (etopo.shape)
5 print (elons.shape)
6 print (elats.shape)
```

```
(540, 1081)
(1081,)
(540,)
```

Tada!!



STEP 3: generate a 2-D longitude-latitude grid

```
1 xx, yy = np.meshgrid(elons,elats) # mesh a 2-D
```

STEP 4: map 2-D longitude-latitude grid to the map coordinates

```
6 x,y=m(xx,yy) # map to the Basemap coord
```

STEP 5: create filled contour plots with “jet” colormap on the map

```
9 cs=m.contourf(x,y,etopo,30,cmap='jet') # the 30 is the number of contours
```

STEP 6: Labeling etc..

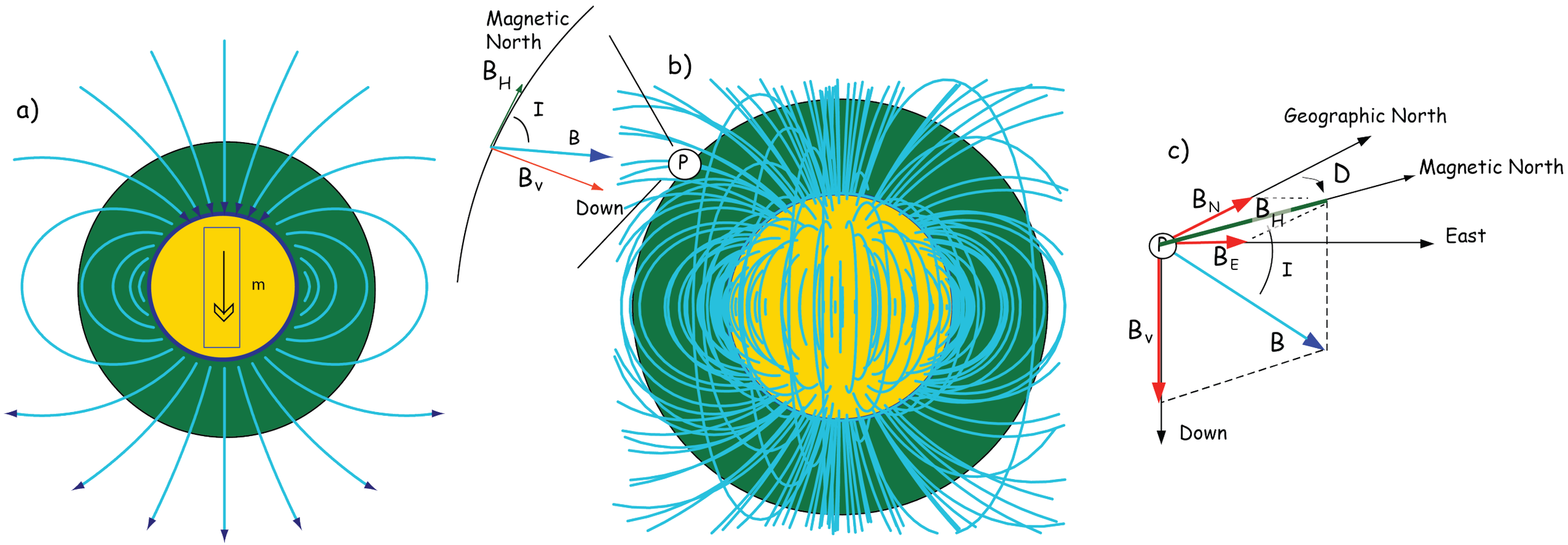
Put everything together: IGRF on maps

Now we know how to grid and contour data, we can try to plot other features of our planet, like the magnetic field strength or inclination (dip below or above the horizontal). Up until 2015, the data come from the International Geomagnetic Reference Field (IGRF) model and after that, the data are extrapolated. To learn more, I've prepared a mkigrf module. For more information, check out this website: https://www.ngdc.noaa.gov/IAGA/vmod/igrf.html.

You can use the function **mkigrf.doigrf** to find out the magnetic declination at a particular time and place. You could then use this function to set your compass. Handy for geology geeks…

But first, what is a magnetic field vector?. As with all vectors, it has both direction and length. We can express the vector in terms of cartesian coordinates (say, North, East and Down) or these polar coordinates:

- declination: the angle of the magnetic direction in the horizontal plane with respect to the North pole
- inclination: the angle of the magnetic direction in the vertical plane with respect to the horizontal
- intensity: the strength of the field, usually in units of tesla (either nano or micro). Tesla is is magnetic induction and is usually represented by the letter **B**.



Use the mkigrf module

```
1  import mkigrf         # import the mkigrf module
2                        # Author: Lisa Tauxe, ltauxe@ucsd.edu
3  help(mkigrf.doigrf) # help info
```

```
Help on function doigrf in module mkigrf:

doigrf(long, lat, date)
    Returns the x,y,z,f components of the geomagnetic field at location long/lat for decim
al year.
    x,y,z are the cartesian components of the field and f is the total field strength (in
nT).
```

mkigrf.doigrf returns (x,y,z) cartesian components of the magnetic field vector. But we want to plot the polar coordinates *declination, inclination, and strength*. So we need to convert from cartesian coordinates to polar coordinates. There is a handy function **mkigrf.cart2dir**() that will do this for you. For example, we find the declination, inclination and magnitude of geomagnetic field for Hong Kong in 2017 like this:

```
1  HK_lat=22
2  HK_lon=114
3  x,y,z,f=mkigrf.doigrf(HK_lon,HK_lat,2018)
4  Dec,Inc,B=mkigrf.cart2dir(x,y,z)
5  print ('The Declination of the geomagnetic field at Lat =',HK_lat,'and Lon =',HK_lon,'is %4.1f'%(Dec),'degrees') #
6  print ('The Inclination of the geomagnetic field at Lat =',HK_lat,'and Lon =',HK_lon,'is %4.1f'%(Inc),'degrees')
7  print ('The Magnitude   of the geomagnetic field at Lat =',HK_lat,'and Lon =',HK_lon,'is %4.1f'%(B),'nano-Tesla')
```

```
The Declination of the geomagnetic field at Lat = 22 and Lon = 114 is 357.2 degrees
The Inclination of the geomagnetic field at Lat = 22 and Lon = 114 is 32.8 degrees
The Magnitude   of the geomagnetic field at Lat = 22 and Lon = 114 is 45070.5 nano-Tesla
```

Put everything together: IGRF on maps

We can call magMap for year 2018:

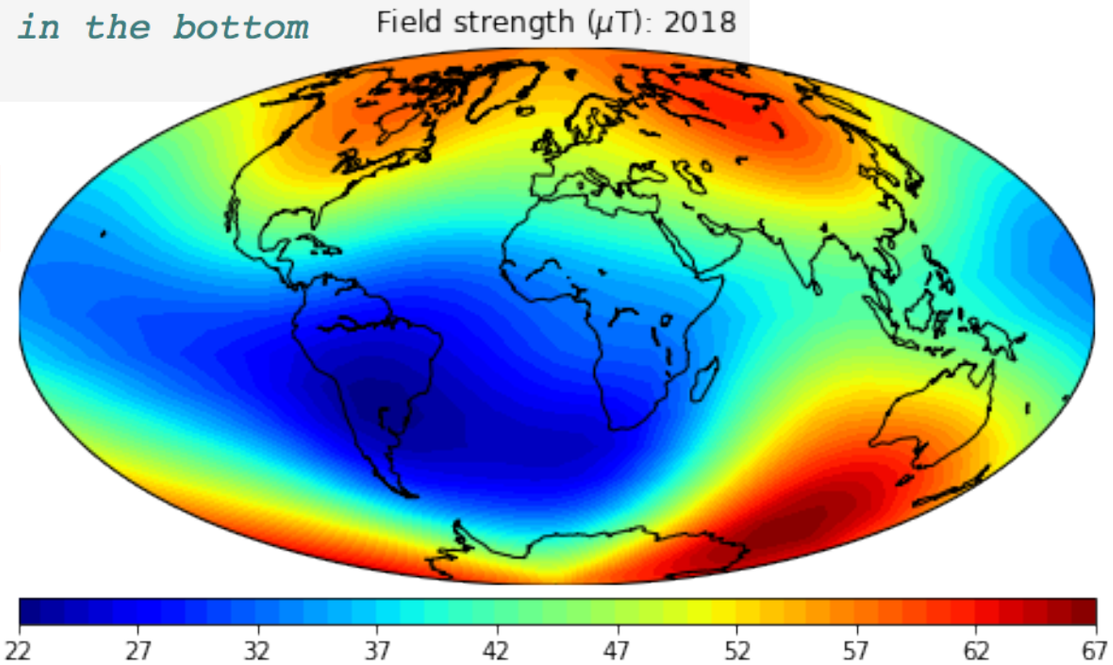
```
date=2018 # let's do this for 2018 (actually, this is the beginning of 2017)
lon_0=0 # we can specify the grid spacing and the intended 0 longitude for the plot
Ds,Is,Bs,lons,lats=magMap(date,lon_0=lon_0)
```

After calling the **magMap** function, we have three **2-D arrays** for Declination (Ds), Inclination (Is), Magnitude (Bs), together with two **1-D arrays** of the longitude-latitude grid (lons, lats). Now let's plot the magnitude of IGRF (Bs) on a contour map.

```
plt.figure(figsize=(8,6))
m = Basemap(projection='hammer',lon_0=lon_0) # create a map proj
m.drawcoastlines() # draw coastlines

xx, yy = np.meshgrid(lons,lats) # mesh a 2-D lon-lat grid

x,y=m(xx,yy) # convert to map coordinate

cs=m.contourf(x,y,Bs,30,cmap='jet') # filled contour with 30 (automatic) levels

# by assigning the contourf object to cs, we can add a colorbar
cbar=m.colorbar(cs,location='bottom') # display colorbar in the bottom
plt.title('Field strength ($\mu$T): '+str(date)); #label
```

Isn’t that fun?
Try IGRF and plot the inclination!

Tada!!



Let’s make some 2-D color plots using matplotlib!