EASC2410 Lecture 20

Introduction to 3D Visualization using Python: Lines, Surfaces, Vectors

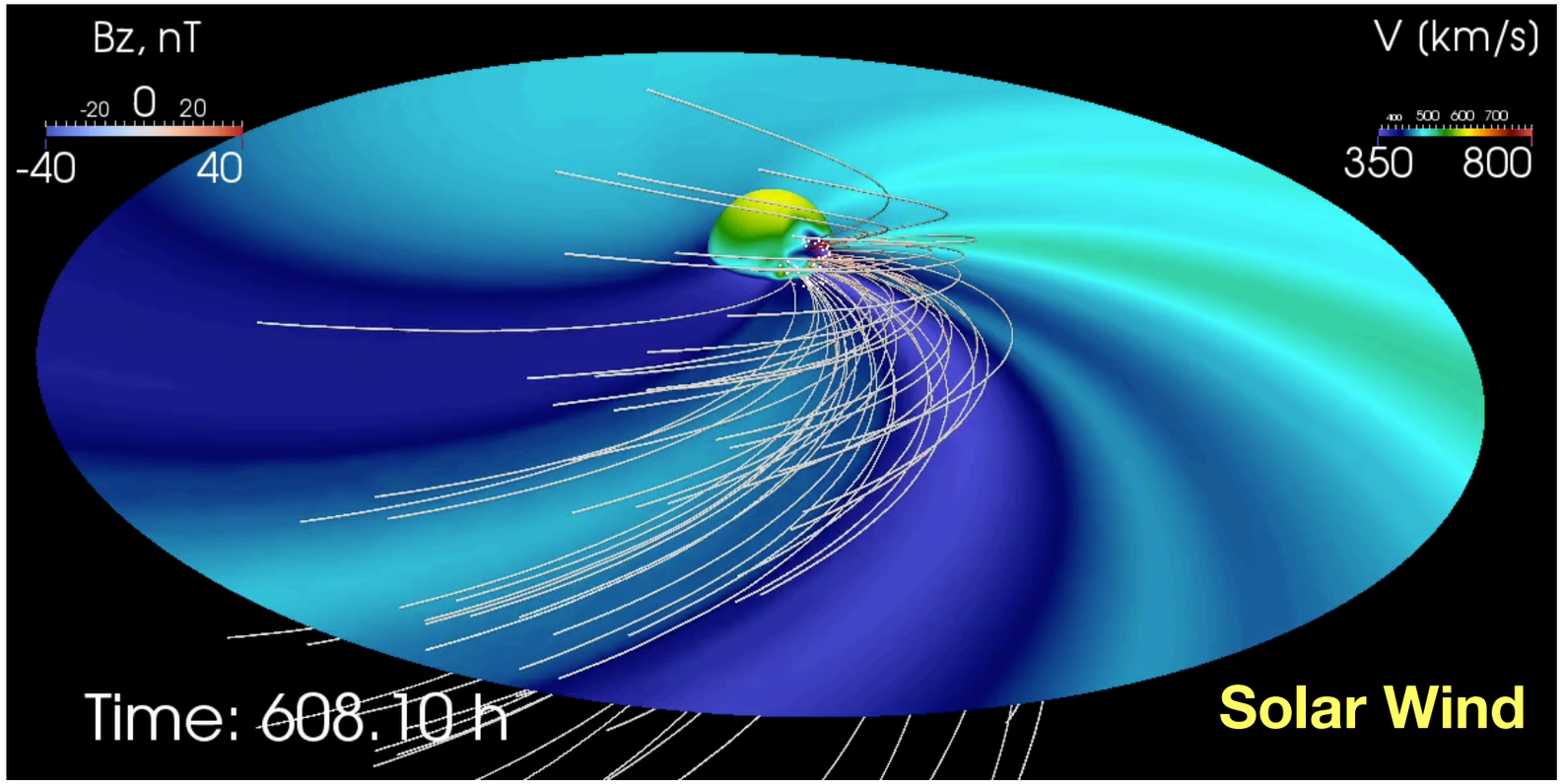
Dr. Binzheng Zhang
Department of Earth Sciences

Review of Lecture 19:

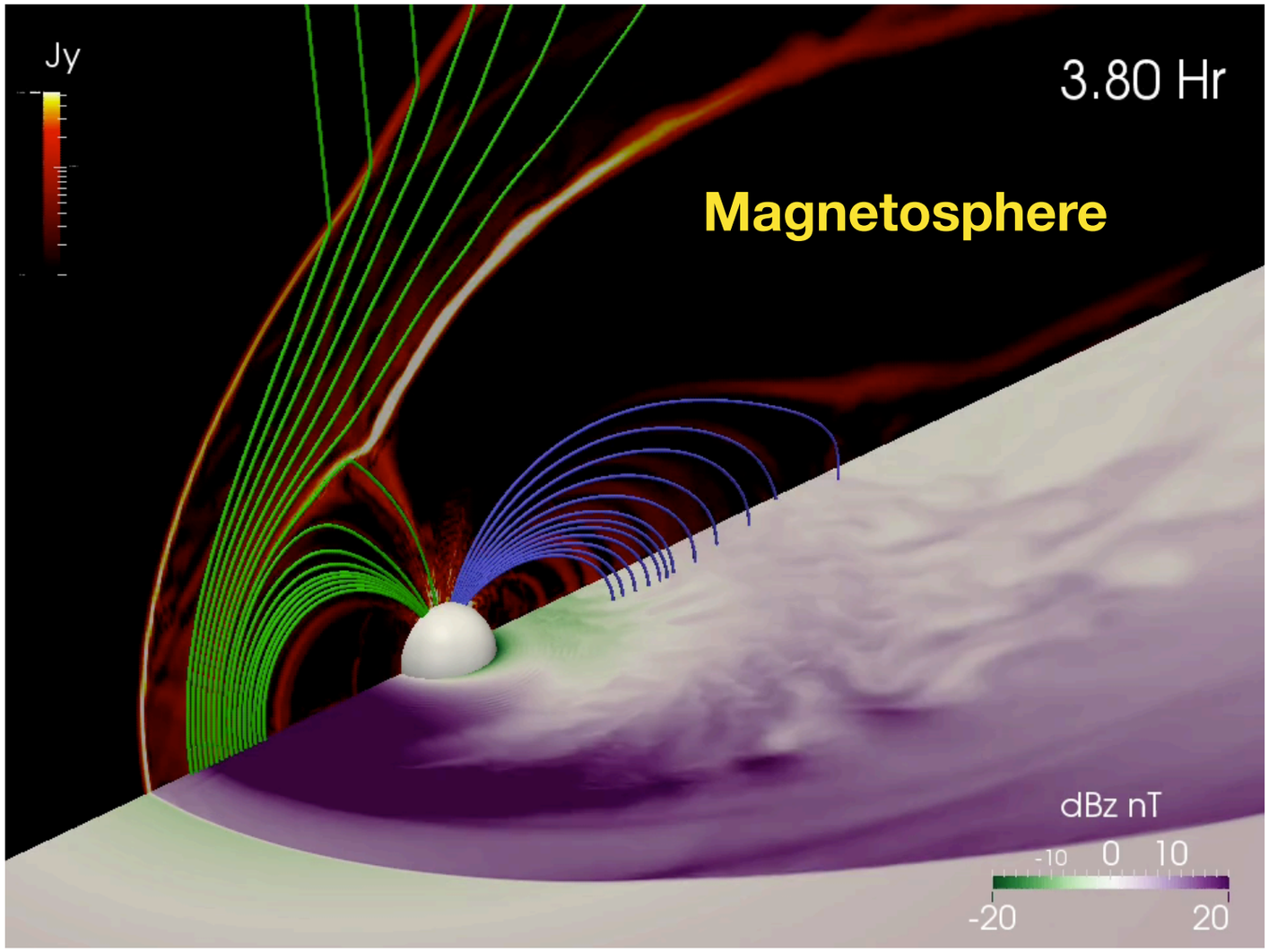
- Concepts: grid mesh
- 2-D pseudocolor plots, contours
- Python tricks: matplotlib 2-D visualization on maps

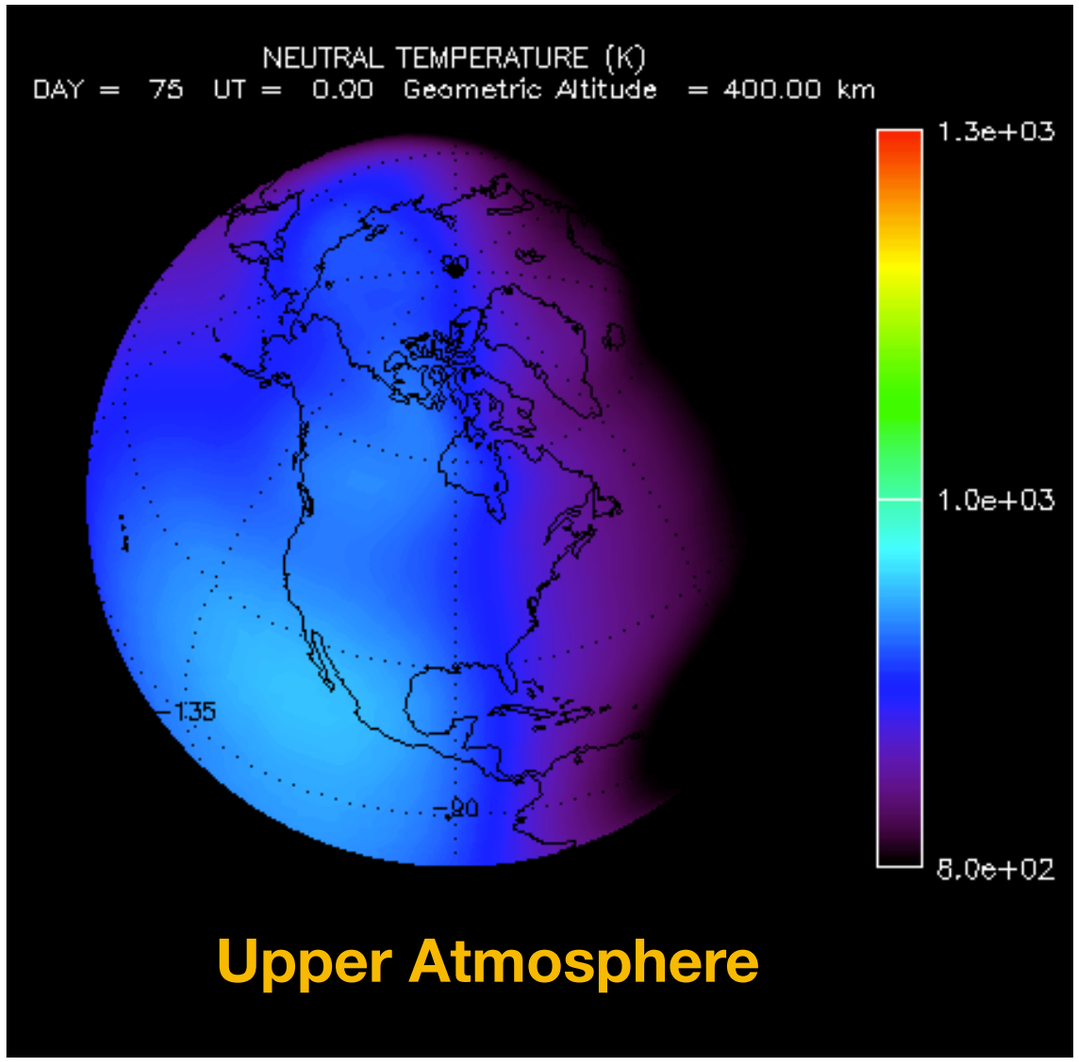
In Lecture 20, you will learn:

- plot data in polar coordinates
- 3-D line plots
- 3-D surface plots
- vector plots
- streamline plots



3-D line+color plots



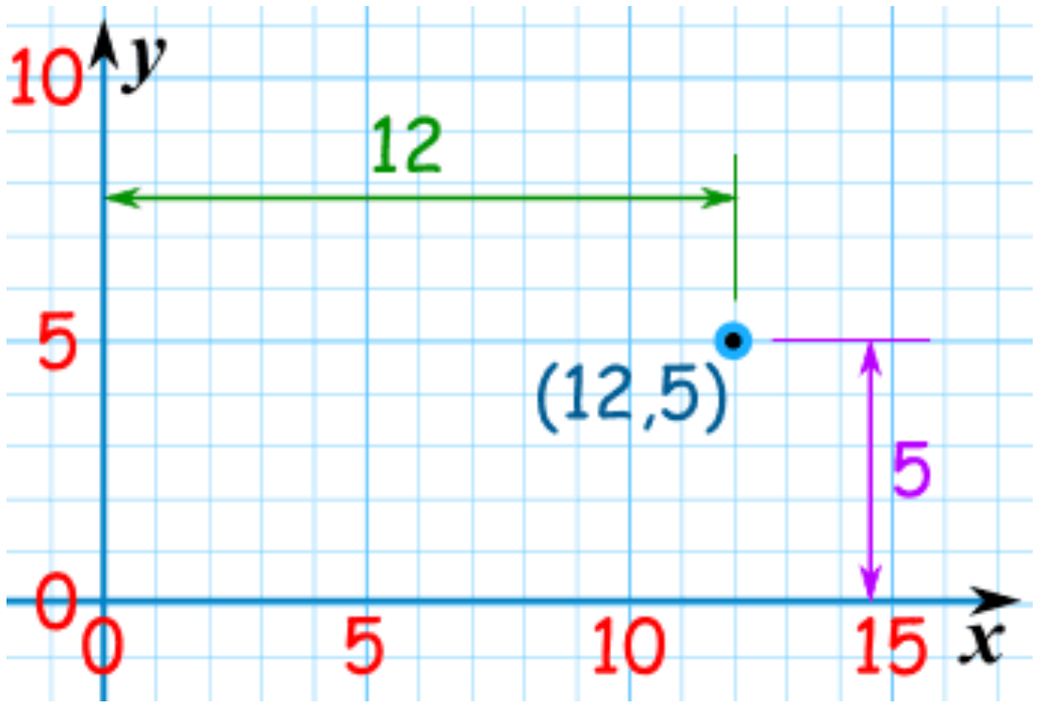


Types of 3-D plots

- 3-D Lines
- 3-D Surface
- 3-D Vectors/streamlines

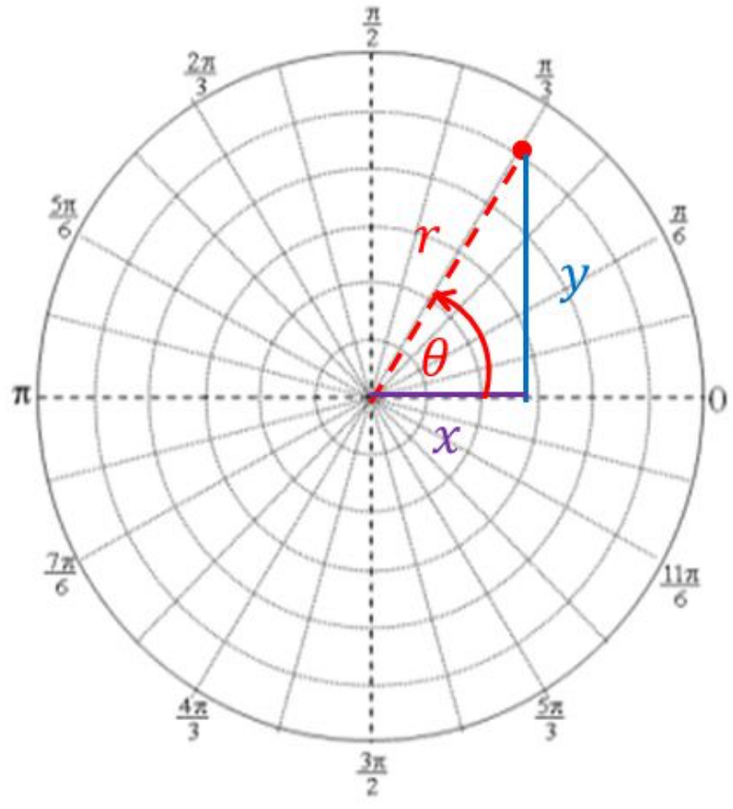
Before diving into 3-D line plots, let's first learn how to make 2-D line plots in polar coordinates

Cartesian Coordinates



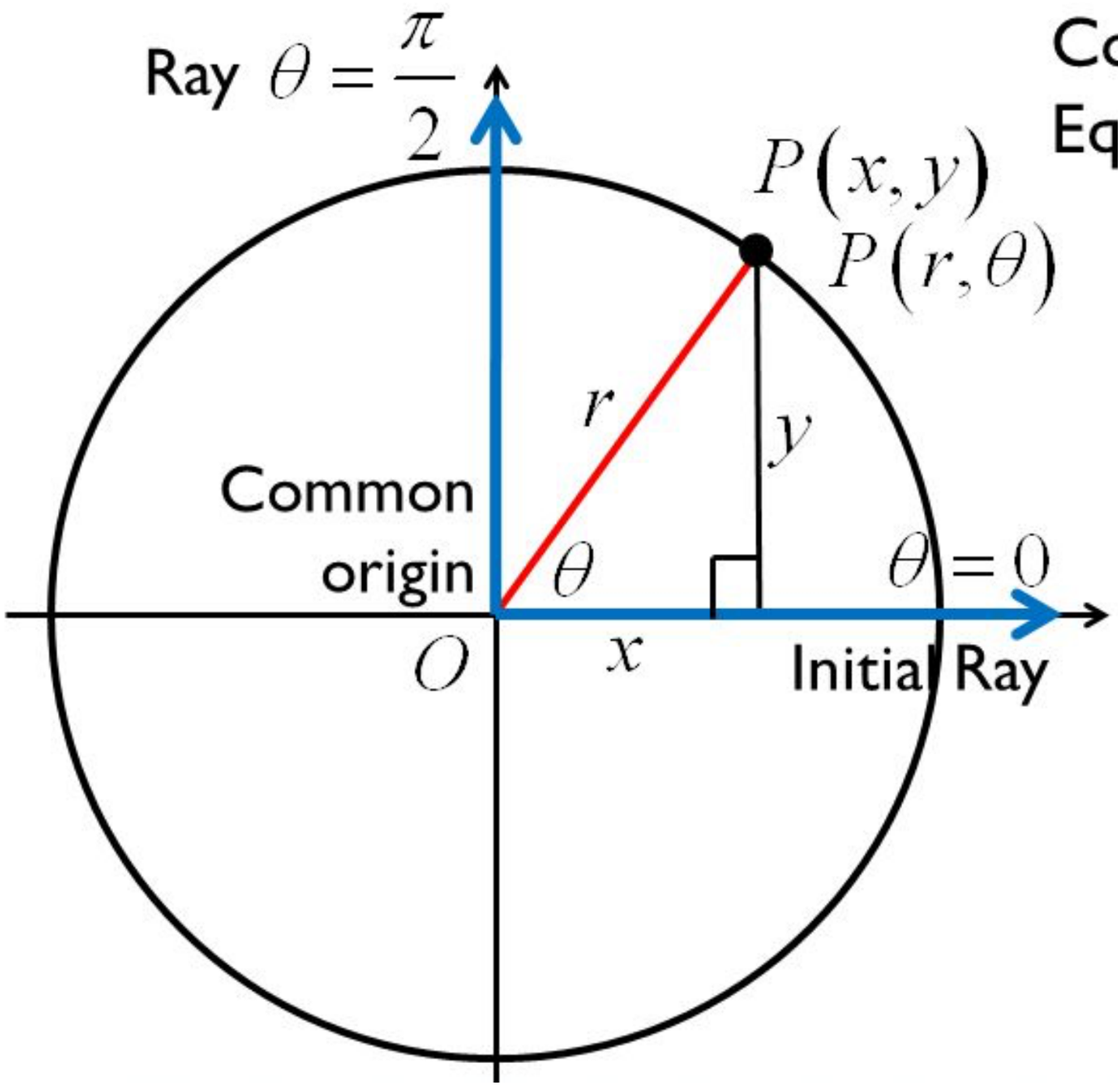
plt.pcolor(x,y)
plt.contour(x,y)
etc.

Polar Coordinates



?

Relating Polar Coordinates with Cartesian Coordinates



Coordinate Conversion Equations:

$$x = r\cos\theta$$

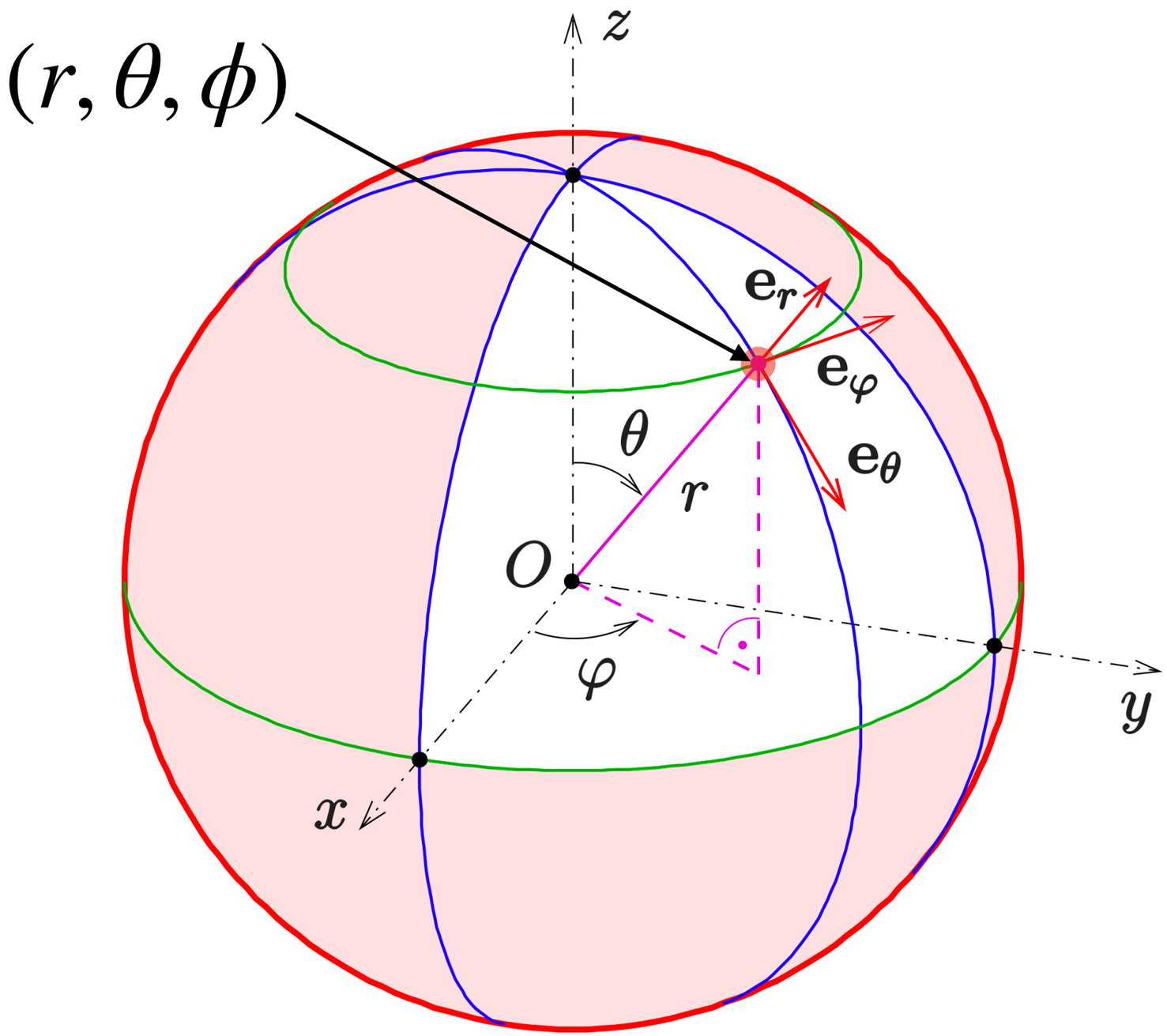
$$y = r\sin\theta$$

$$x^2 + y^2 = r^2$$

$$\frac{y}{x} = \tan\theta$$

if you have arrays for r, θ you can use the polar() function in matplotlib to generate lines in polar coordinates directly

Relating Spherical Coordinates with Cartesian Coordinates



$$x = r \cdot \sin\theta \cdot \cos\phi$$

$$y = r \cdot \sin\theta \cdot \sin\phi$$

$$z = r \cdot \cos\theta$$

$\theta : [-\frac{\pi}{2}, \frac{\pi}{2}]$ **(latitude)**

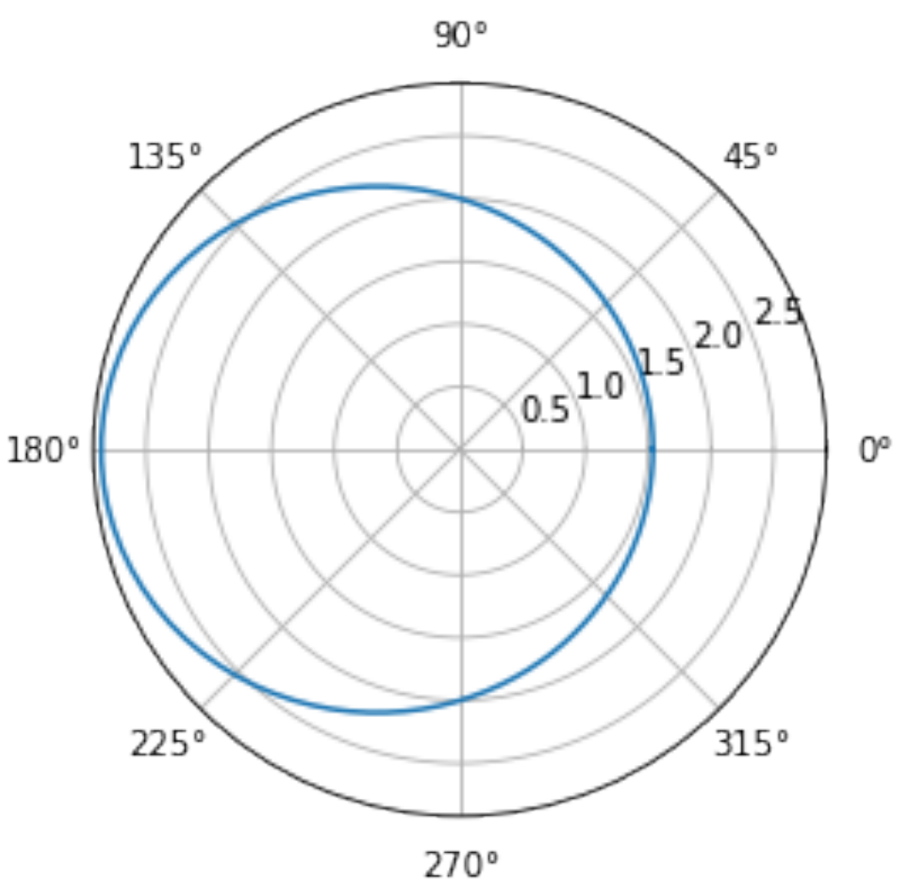
$\phi : [0, 2\pi]$ **(longitude)**

Let's make an ellipse

The equation of an ellipse in the polar geometry goes like: $$r = \frac{Ra}{1 + e\cos\theta}$$

Let's use a Python code to draw an ellipse with Ra = 2.0 and Eccentricity = 0.3:

```
1  theta = np.linspace(0,np.pi*2,100) # 1-D array of theta between 0 and pi*2
2  Ra = 2.0 # Ra
3  e = 0.3  # eccentricity
4  r = Ra/(1+e*np.cos(theta)) # ellipse equation
5  plt.polar(theta,r) # make a polar plot
```



Sytax: plt.polar(theta, r)

- theta is the azimuthal angle
- r is the radius, which is a function of theta
- matplotlib generates a polar grid together with a line plot

Let's try something more complicated

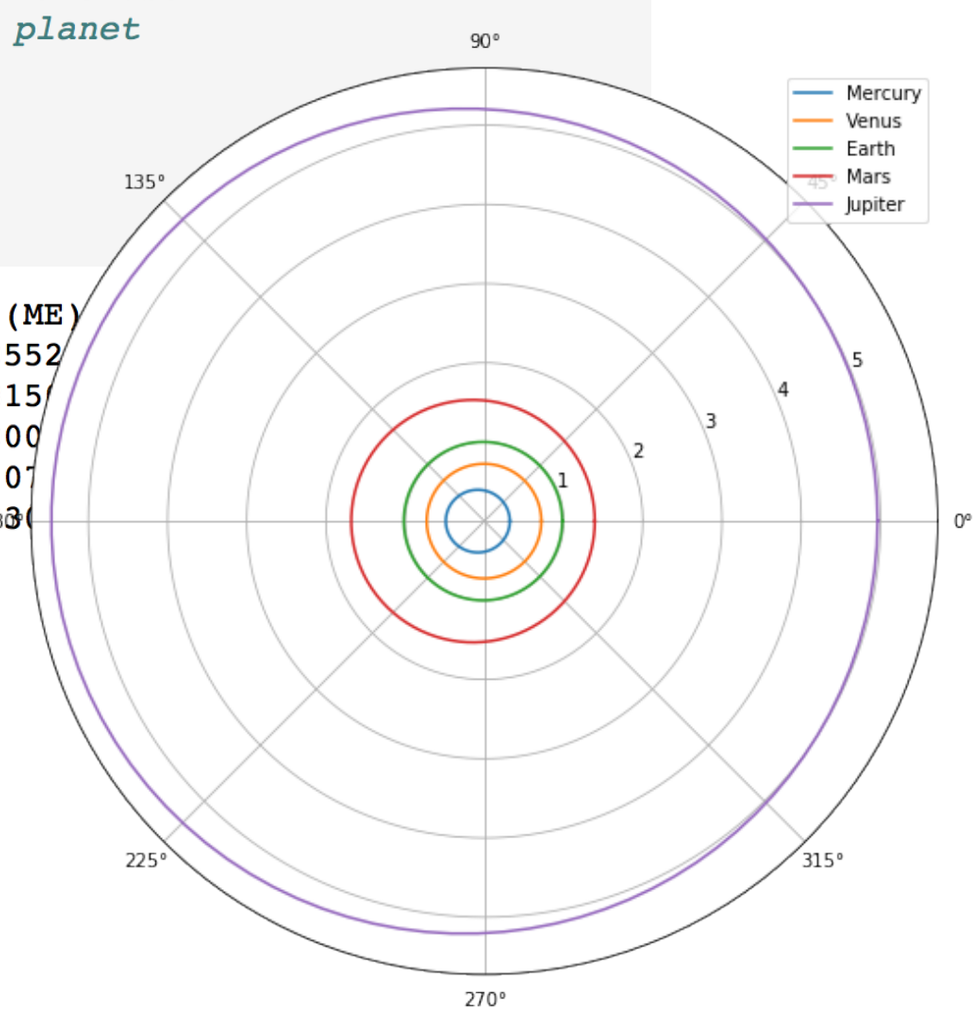
Plot the orbits of all the solar system planets, which requires:

- Pandas data frame
- for-loops
- matplotlib (polar)

Pay attention to the new trick: loop through rows of Pandas data frame

```
1  df = pd.read_excel('datasets/planetary_orbital.xlsx',index_col=0) # load data
2  print(df.head()) # take a look at the data frame, know what columns you've got
3
4  plt.figure(figsize=(9,9)) # create a new figure
5
6  theta = np.linspace(0,np.pi*2,100) # create an array for theta, between 0, 2*pi
7
8  for row in df.head().itertuples(): # iterate over rows of a data frame use
9      e = row.Eccentricity           # iterturples(), which is each planet
10     Ra = row.SemimajorAxis
11     r = Ra/(1+e*np.cos(theta))
12     plt.polar(theta,r,label=row.Index) # generate a polar plot
13
14 plt.legend() # showing labels
```

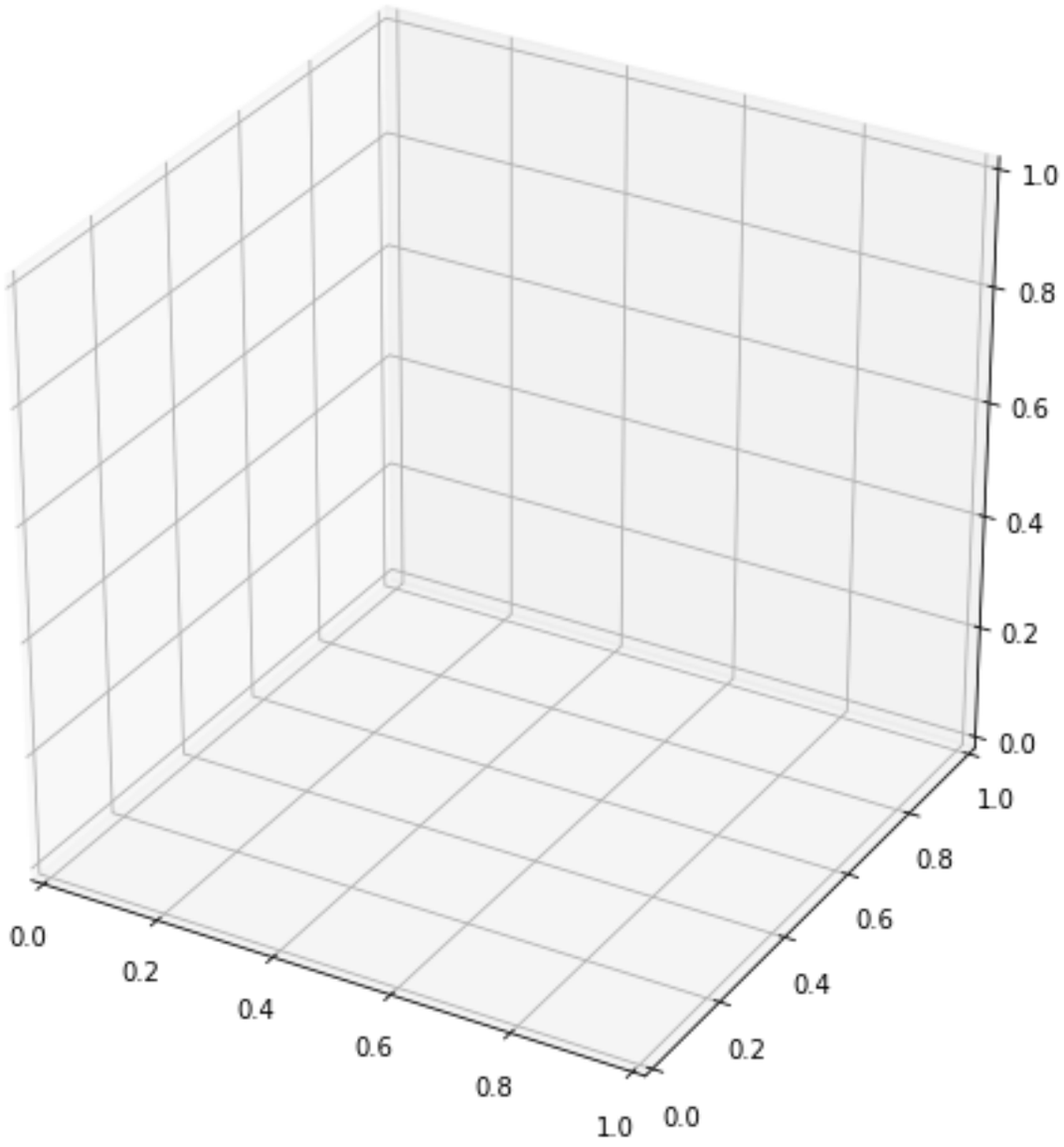
	SemimajorAxis	Eccentricity	Orbital Period (years)	Mass (ME)
Mercury	0.387099	0.20564	0.240847	0.0552
Venus	0.723336	0.00678	0.615197	0.815
Earth	1.000003	0.01671	1.000017	1.000
Mars	1.523710	0.09339	1.880816	0.107
Jupiter	5.202900	0.04840	11.862615	317.83



Now in the polar coordinates, you get orbits for five solar planets.

Let's try visualize the orbits in a 3-D figure

```
1  from mpl_toolkits import mplot3d
2  from mpl_toolkits.mplot3d import Axes3D
3
4  fig = plt.figure(figsize=(9,9))
5  ax = plt.axes(projection='3d')
```



Modules used:

- **mplot3d**
- **Axes3D**

The plt.axes( ) function creates a 3-D axes which looks like a "box" as shown on left. within the box, you can plot 3-D shapes for your visualization.

Let's try to plot the 3-D orbits of the solar planets in the 3-D axes.

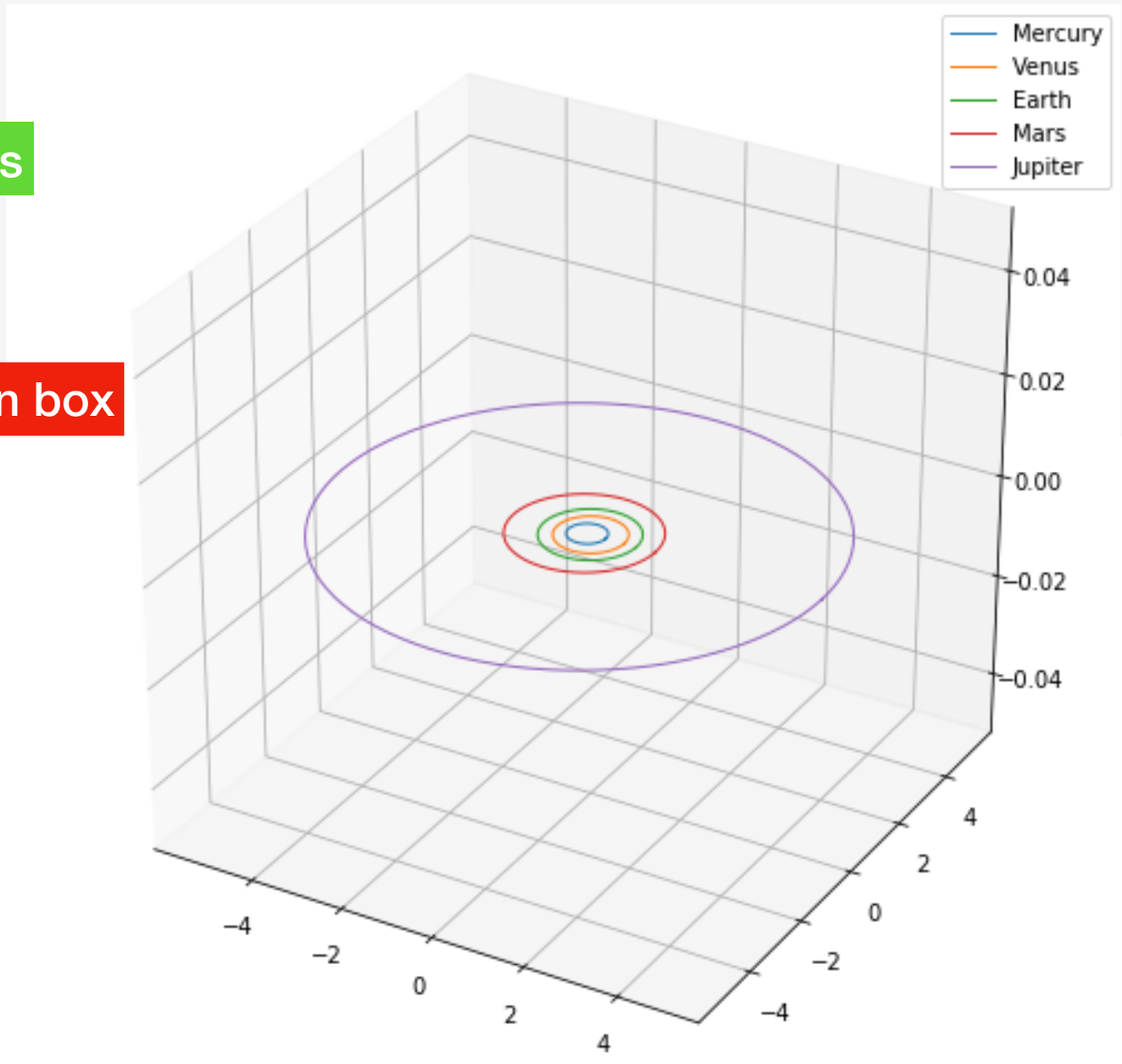
Let's try visualize the orbits in a 3-D figure

```
from mpl_toolkits import mplot3d
from mpl_toolkits.mplot3d import Axes3D

df = pd.read_excel('datasets/planetary_orbital.xlsx',index_col=0)
print(df.head())

fig = plt.figure(figsize=(9,9))
ax = plt.axes(projection='3d')

theta = np.linspace(0,np.pi*2,100)
for row in df.head().itertuples():
    e = row.Eccentricity
    Ra = row.SemimajorAxis
    r = Ra/(1+e*np.cos(theta))
    x = r*np.cos(theta)
    y = r*np.sin(theta)
    z = x*0
    ax.plot3D(x,y,z,label=row.Index,linewidth=1)

plt.legend()
```

Polar Coordinates

Cartesian

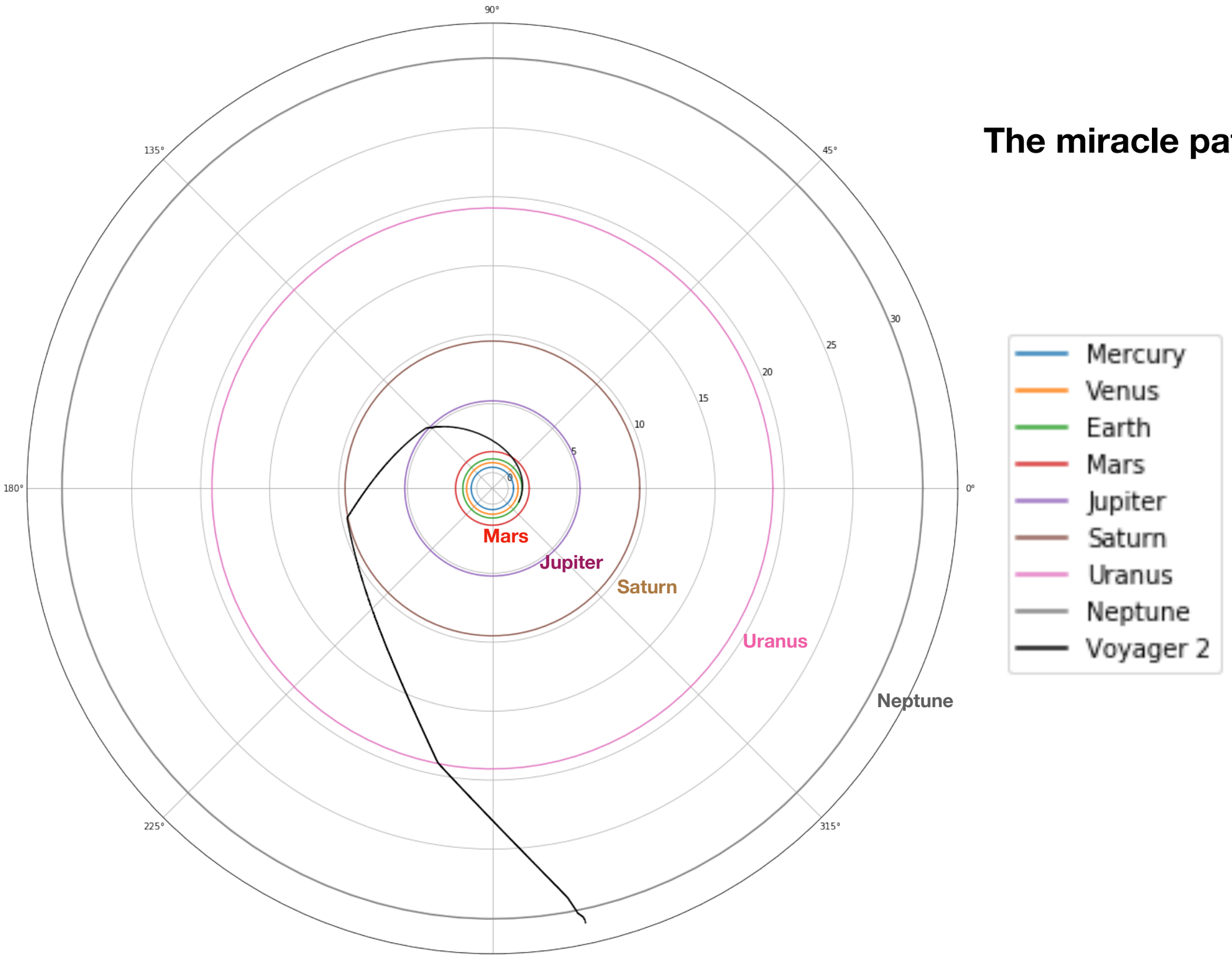
3-D line plot in Cartesian box



Sytax: plt.plot3D(x,y,z,options)

- x,y,z are the Cartesian locations for a 3-D line
- in this example, the z locations are set to zero
- loop over rows of pandas data frame for first 5 planets

Let’s try visualize the trajectory of Voyager 2



The miracle path!

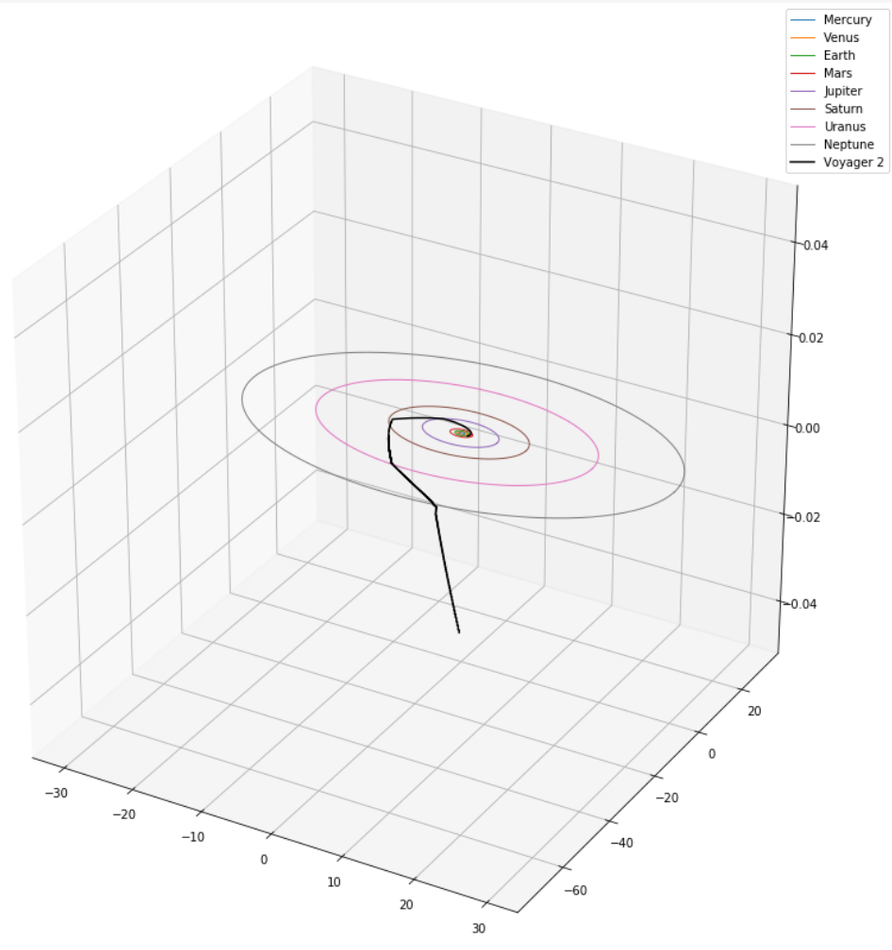
Let's try visualize the trajectory of Voyager 2

```
from mpl_toolkits import mplot3d

df = pd.read_excel('datasets/planetary_orbital.xlsx',index_col=0)
print(df.head())

fig = plt.figure(figsize=(15,15))
ax = plt.axes(projection='3d')

theta = np.linspace(0,np.pi*2,100)
for row in df.itertuples():
    e = row.Eccentricity
    Ra = row.SemimajorAxis
    r = Ra/(1+e*np.cos(theta))
    x = r*np.cos(theta)
    y = r*np.sin(theta)
    z = x*0
    ax.plot3D(x,y,z,label=row.Index,linewidth=1)

data = pd.read_csv('datasets/vy2trj_ssc_1d.asc.txt',delimiter="\s+")

print(data.head())
d = data#[data.YEAR<=1989]
x = d.DISTAN*np.cos(d.SECLON*np.pi/180)
y = d.DISTAN*np.sin(d.SECLON*np.pi/180)
z = x*0

ax.plot3D(x,y,z,color='k',label='Voyager 2')

plt.legend()
```

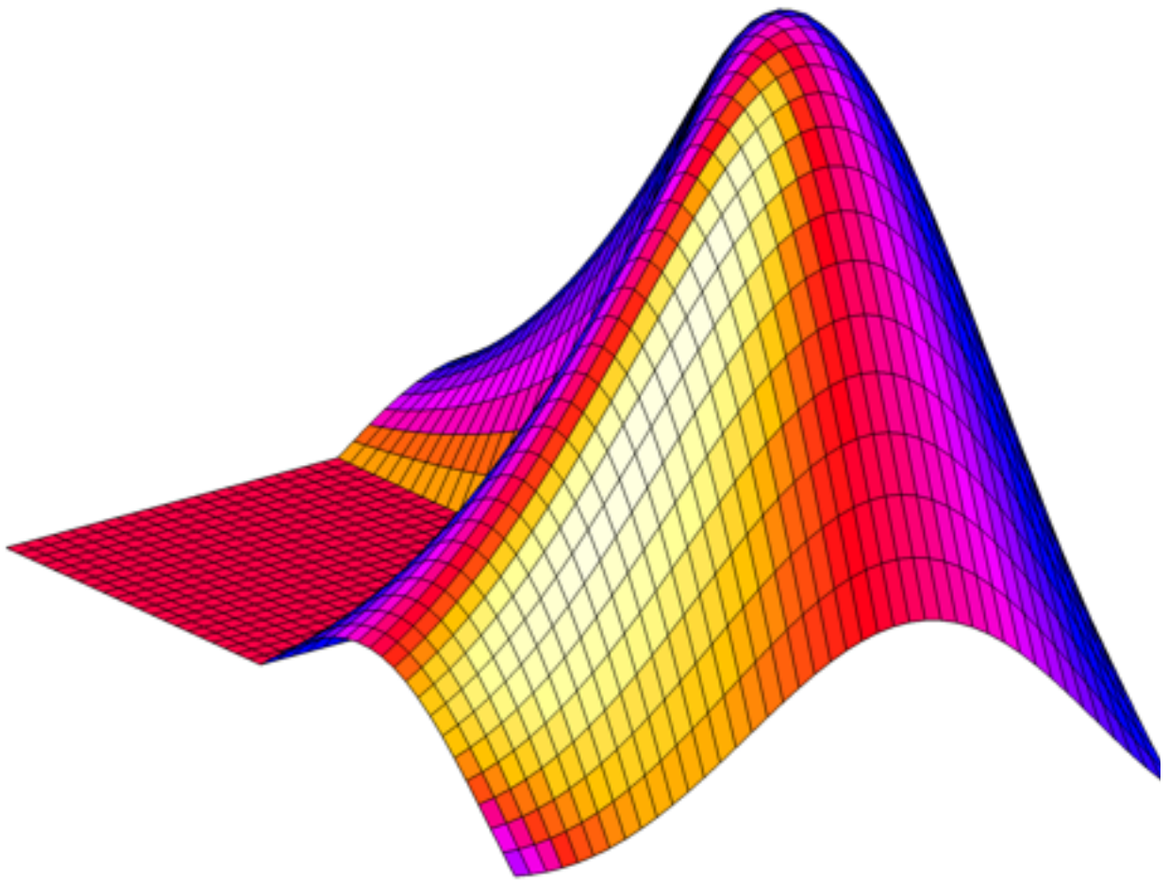
well, a 3-D trajectory is not necessary more illustrative than a 2-D trajectory!

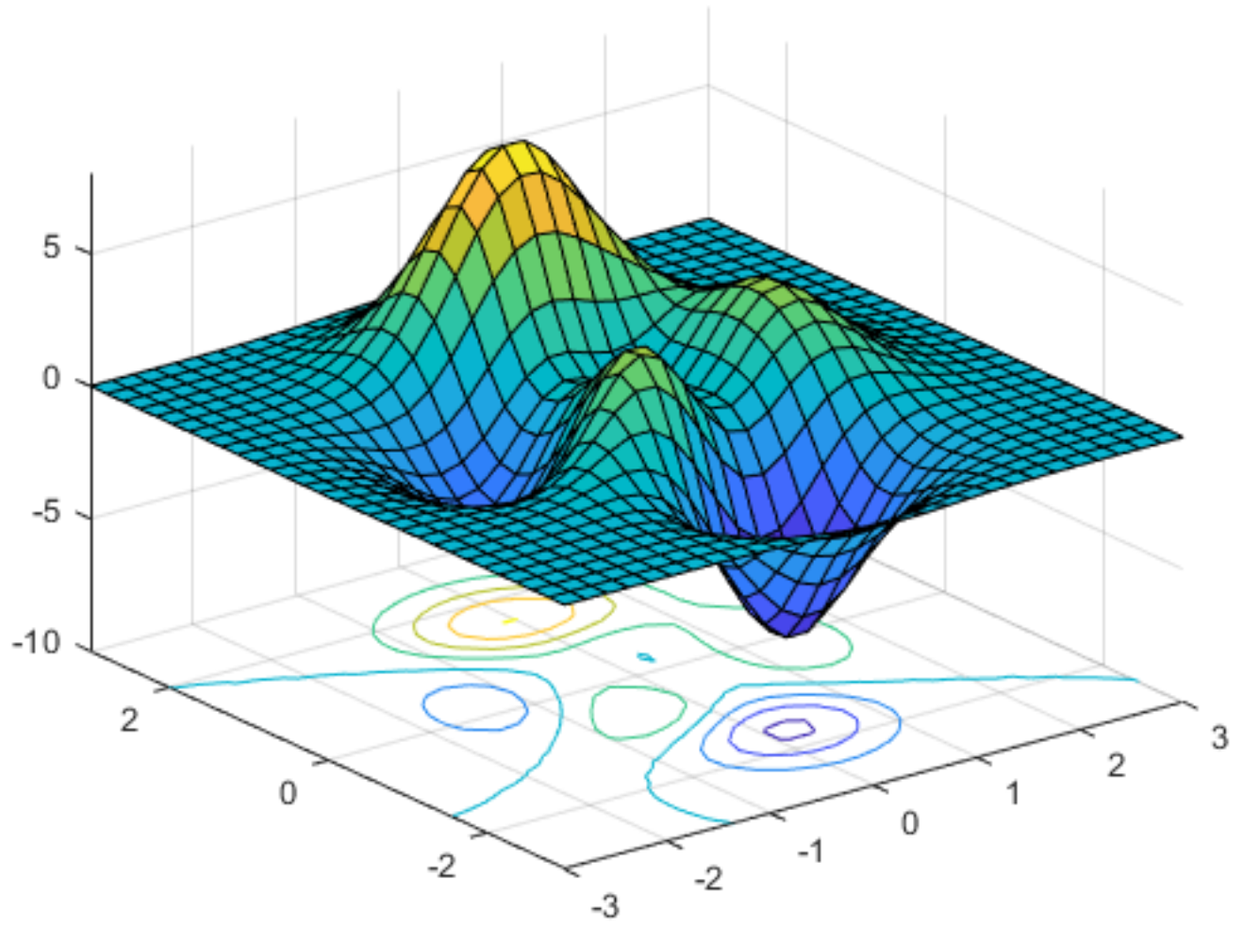
r theta

	YEAR	DAY	DISTAN	SECLAT	SECLON	HELLAT	HELLON	HILLON
0	1977	237.0	1.01	0.1	333.0	7.1673	122.98	257.45
1	1977	238.0	1.00	0.1	334.3	7.2010	110.10	258.76
2	1977	239.0	1.00	0.1	335.6	7.2311	97.23	260.07
3	1977	240.0	1.00	0.1	336.9	7.2575	84.35	261.38
4	1977	241.0	1.00	0.2	338.2	7.3802	71.48	262.68



3-D surface plots



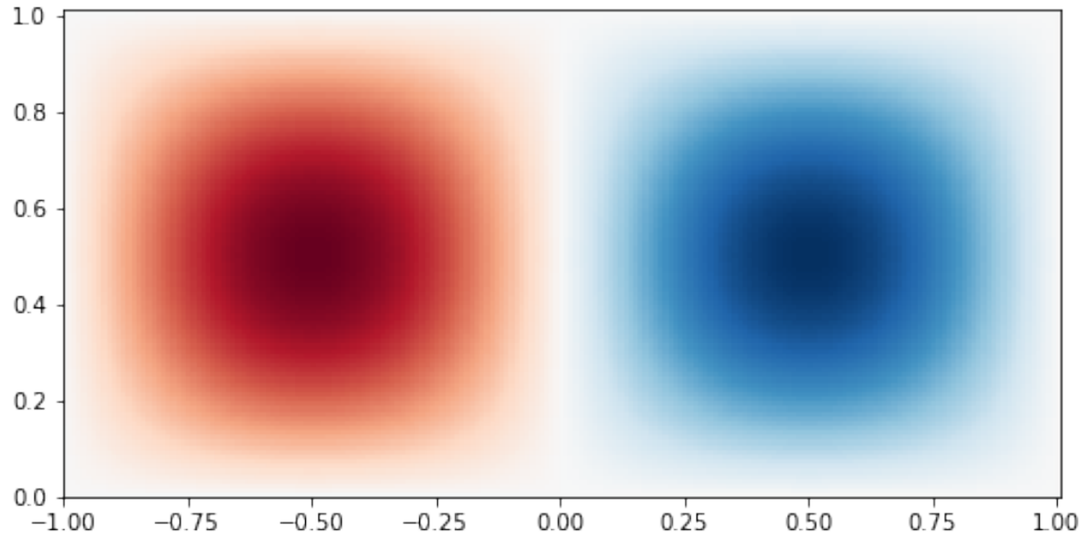


Recall: in 2-D color plots using pcolor(x,y,z), x,y are the 2-D grid and z is the data. If we interpret z as the "height" at each grid point, that's basically a 3-D surface plot

2-D pseudocolor plots and 3-D surface plots

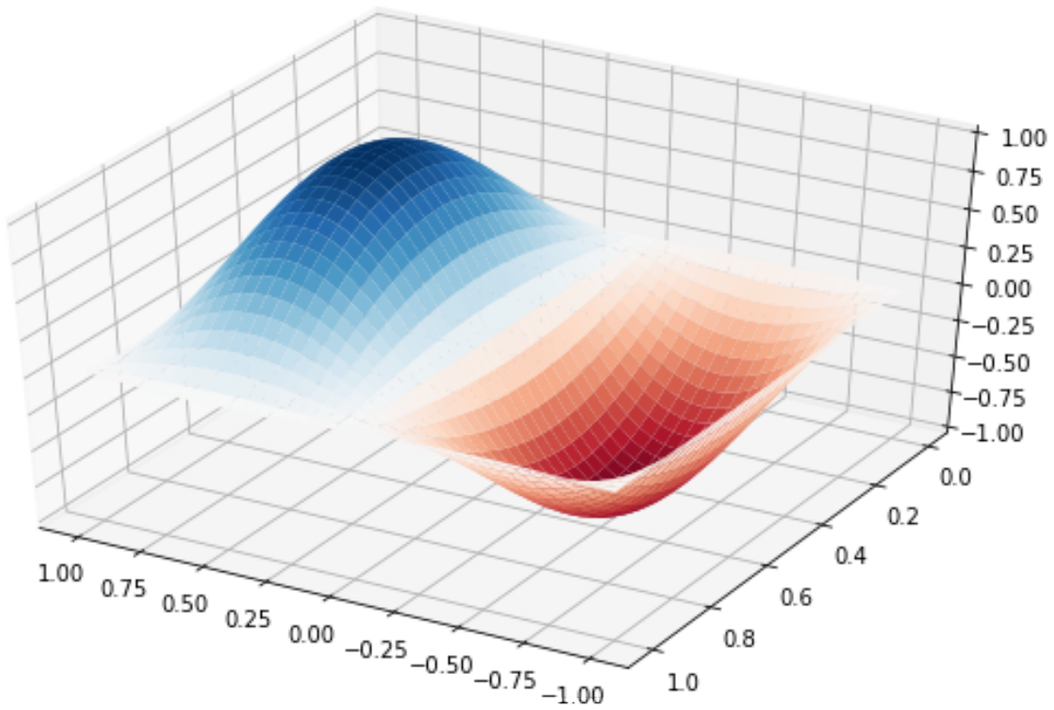
Recall: 2-D color plot

```
# Make data
x = np.arange(-1.0, 1.02, 0.01) # make a 1D array
y = np.arange(0.0, 1.02, 0.01) # ditto
X,Y = np.meshgrid(x, y) # make a meshgrid
Z = np.sin(X*np.pi)*np.sin(Y*np.pi)

# generate a figure
plt.figure(figsize=(8,4))
# Plot the color.
plt.pcolor(X, Y, Z, cmap='RdBu')
```



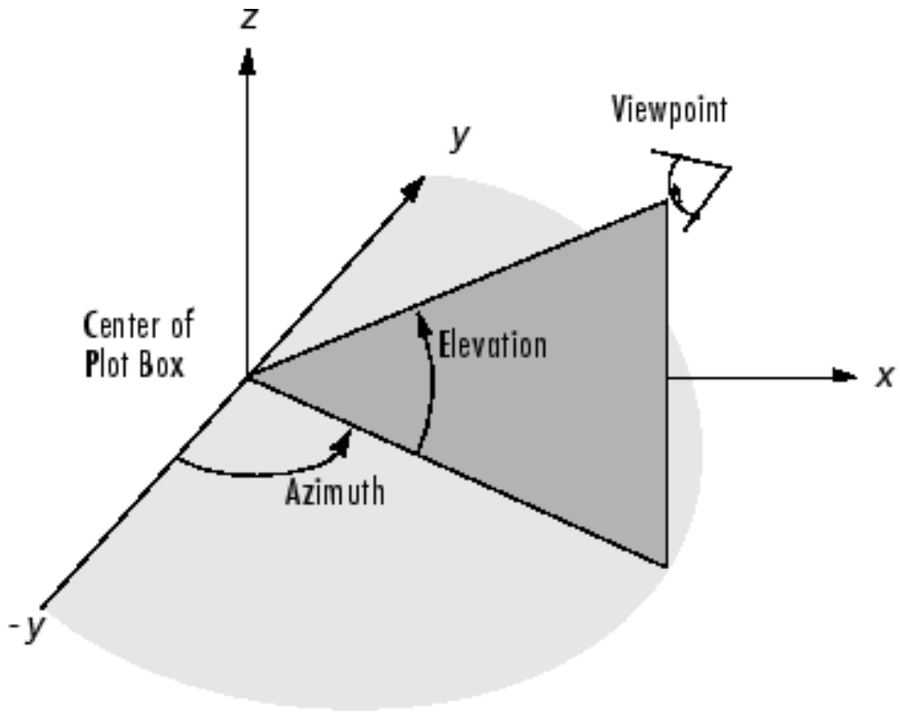
New trick: 3-D surface plot

```
# Make data
x = np.arange(-1.0, 1.02, 0.01) # make a 1D array
y = np.arange(0.0, 1.02, 0.01) # ditto
X,Y = np.meshgrid(x, y) # make a meshgrid
Z = np.sin(X*np.pi)*np.sin(Y*np.pi)

# generate a figure
fig = plt.figure(figsize=(10,6))
ax = fig.gca(projection='3d')

# Plot the surface.
surf = ax.plot_surface(X, Y, Z, cmap='RdBu')
ax.view_init(elev=40., azim=120.)
```

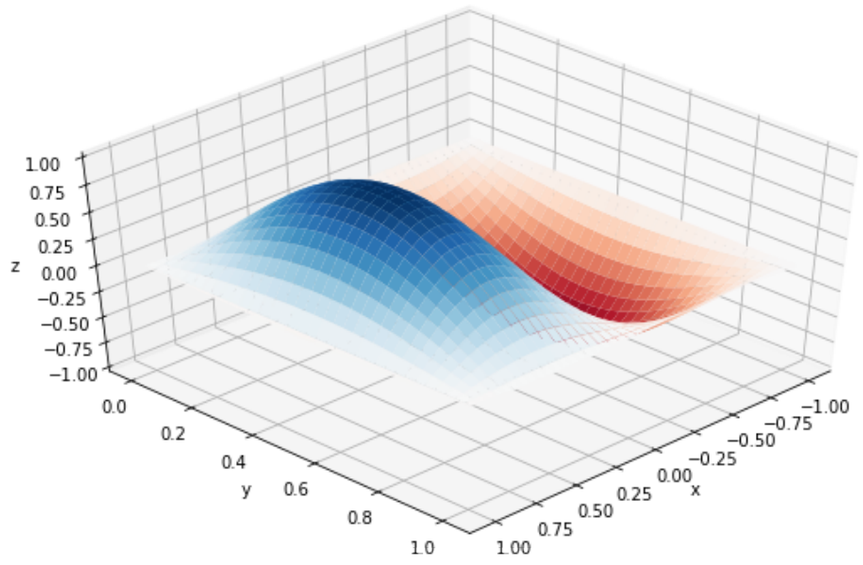


The view angle of a 3-D surface plot

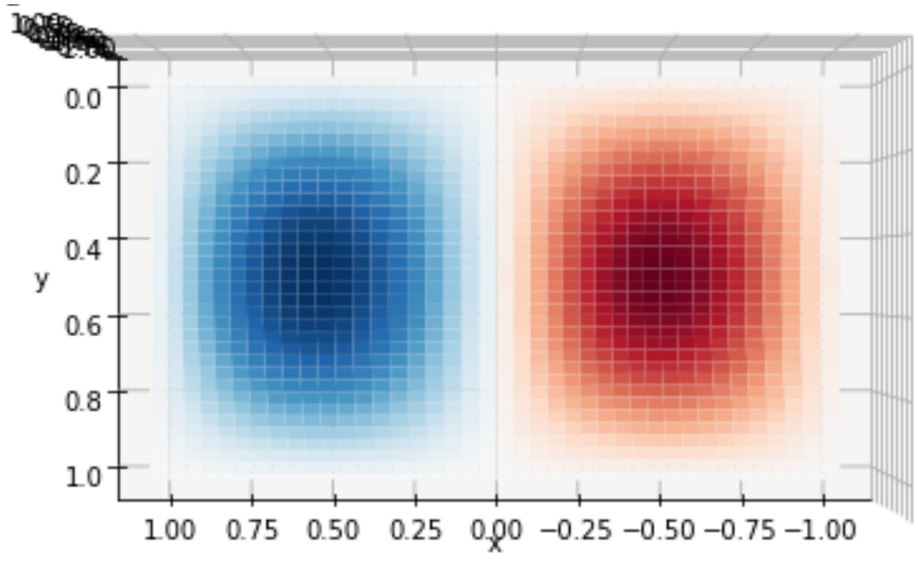
```
11 # Plot the surface.
12 surf = ax.plot_surface(X, Y, Z, cmap='RdBu')
13 ax.view_init(elev=40., azim=120.) # change view angle
```



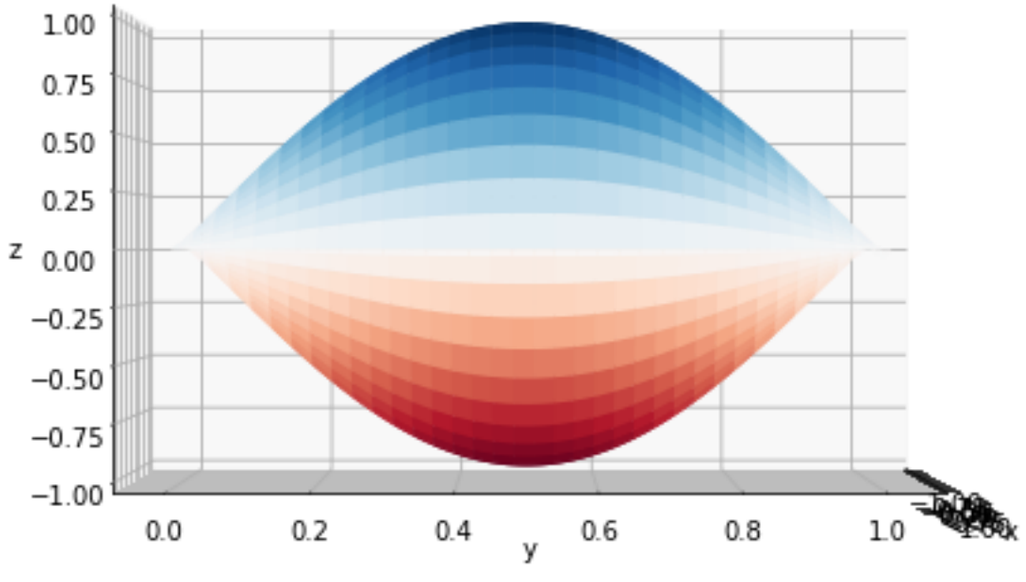

```
14 # Plot the surface.
15 surf = ax.plot_surface(X, Y, Z, cmap='RdBu')
16 ax.view_init(elev=45., azim=45.) # change view angle
```




```
14 # Plot the surface.
15 surf = ax.plot_surface(X, Y, Z, cmap='RdBu')
16 ax.view_init(elev=90., azim=90.) # change view angle
```




```
# Plot the surface.
surf = ax.plot_surface(X, Y, Z, cmap='RdBu')
ax.view_init(elev=0., azim=0.) # change view angle
```



An example of 3-D surface plots: Gravity Anomaly

In the last Lecture, we learned how to make 2D color contour plots, but 3D is much more fun, so let's try to make a 3D version of something geophysical, namely the gravity anomaly of a buried sphere.

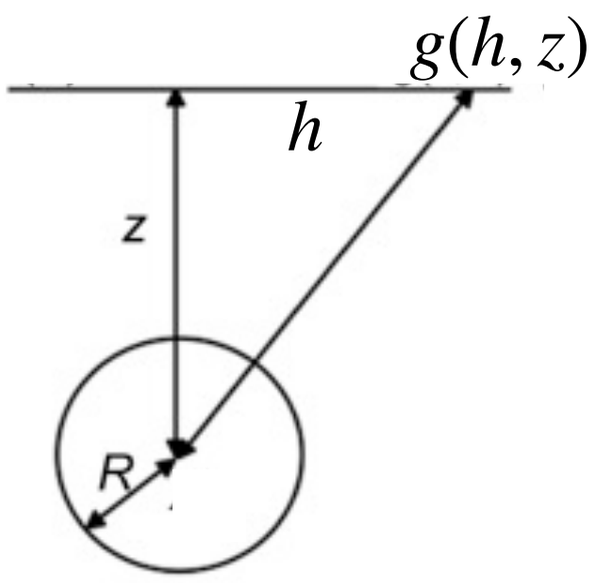
Let's choose a sphere with a radius R of 2 m (whose volume is $\frac{4\pi}{3}R^3$), that is buried $z = 3$ m deep. The sphere's density ($\Delta\rho$) is 500 kg/m^3 , which is for this purpose, much higher than the surrounding material. Oh and we'll need the universal gravitational constant G, which is 6.67x 10^{-11} Nm^2/kg^2 .

The formula for gravitational attraction of such a body is:

$$g = \frac{4\pi}{3}R^3\frac{G\Delta\rho}{(h^2+z^2)},$$

where h is the horizontal distance from the mass.

The units (from dimensional analysis remembering that Newtons are kg $\cdot$ m $\cdot$ s^2) are m $\cdot$ s^{-2}. 1 Gal (for Galileo) = 1 cm $\cdot$s^{-2}, so to convert g to the units of microgals, we multiply the above by 10^8 .



We can write the gravity equation as a lambda function.

```
gravity= lambda G,R,drho,h,z : (1e8*G*4.*np.pi*R**3.*drho)/(3.*(h**2+z**2)) # grav
```

And set up the grid mesh using the meshgrid( ) function through the same way as we did in the last lectures as a color contour on a 2D surface.

```
x=np.arange(-6.,6.,0.1) # range of x values
y=np.arange(-6,6.,0.1) # range of y values
X,Y=np.meshgrid(x,y) # make a meshgrid
```

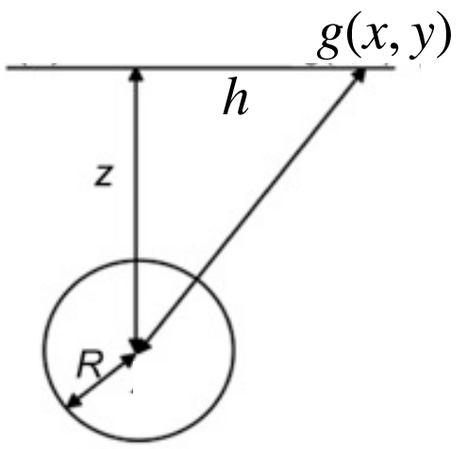
An example of 3-D surface plots: Gravity Anomaly

The formula for gravitational attraction of such a body is:

$$g = \frac{4\pi}{3} R^3 \frac{G\Delta\rho}{(h^2 + z^2)},$$

To the gravity array **g*, *we need z,G,R and* $\Delta\rho$ *and* *h** the horizontal distance from ground zero of the buried sphere, which is of course given by:

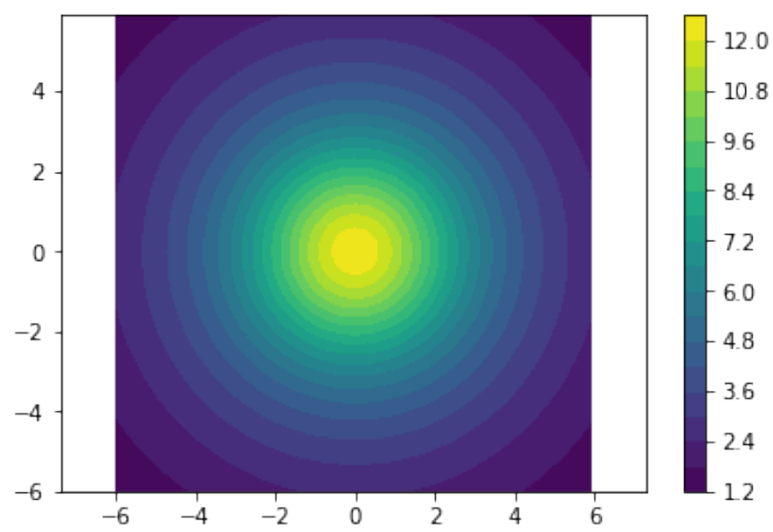
$$h = \sqrt{x^2 + y^2}.$$




```
# define the other parameters
z=3.
G=6.67e-11 # grav constant in Nm^2/kg^2 (SI)
R=2. # radius in meters
drho=500 # density contrast in kg/m^3

h=np.sqrt(X**2+Y**2) # get the horizontal distance from ground zero for x,y
# and make the g array
g=gravity(G,R,drho,h,z)
```

Now we have the gravity **g** is is a function of X and Y. We can make the plot of the gravitational attraction first using our old friend from the last lecture, plt.contourf( ):

```
plt.contourf(x,y,g,20,cmap='viridis')
plt.colorbar()
plt.axis('equal')
plt.show()
```



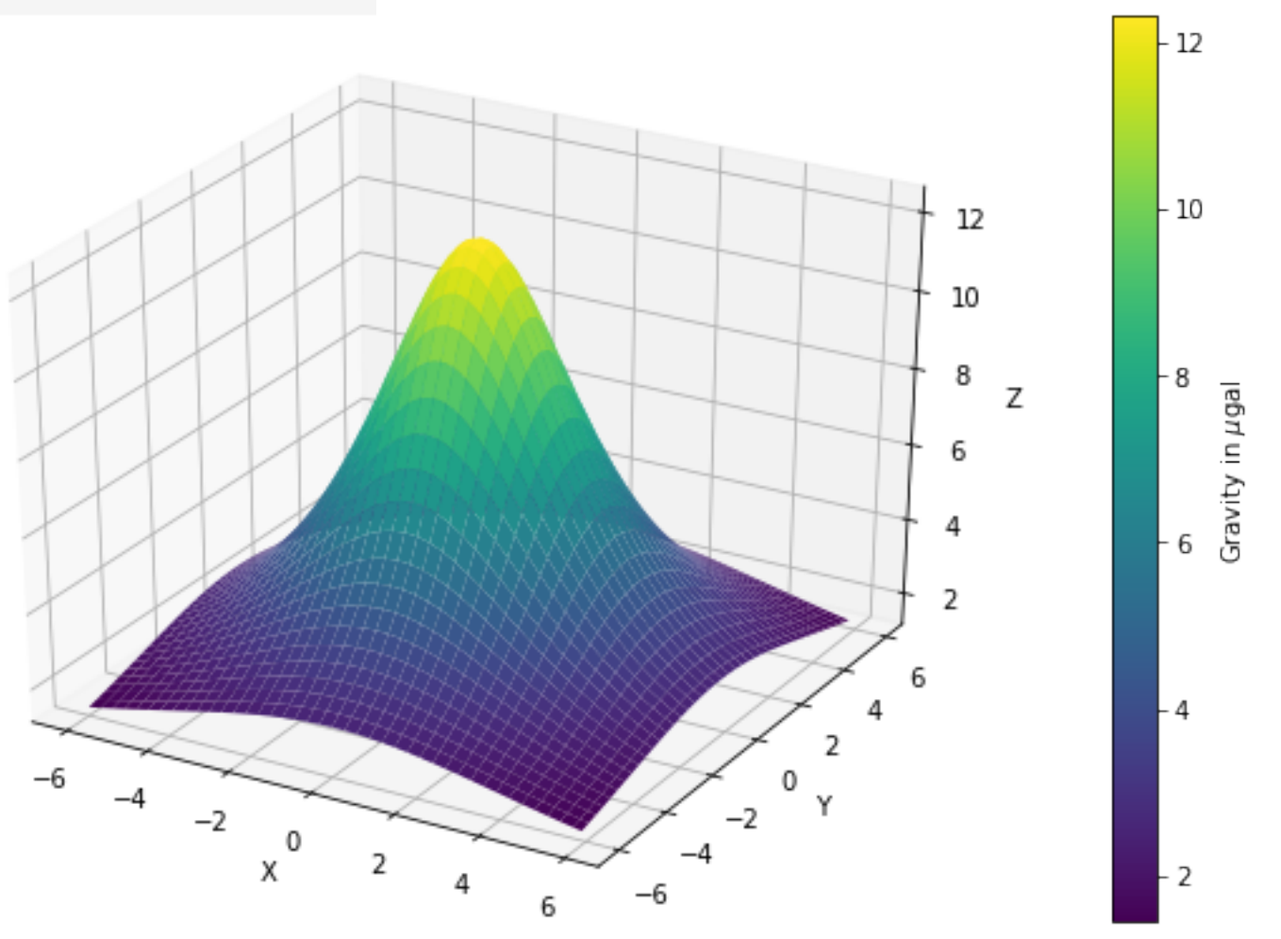
An example of 3-D surface plots: Gravity Anomaly

Now let's learn how to generate **3D Surfaces** using the function **g(X,Y)**

To plot this in 3D, let's use matplotlib **Axes3D**. There is an **Axes3D** method called **plot_surface( )** which might do the trick. The **Axes3D** method **plot_surface** plots a wireframe surface on the **meshgrid**, **X and Y** of the data in **g**.

In the following plot, we create an **Axes3D** instance called **ax** from the **figure** object, **fig**. We then call **plot_surface( )** on the **Axes3D** instance and set labels to the 3 axis. Note that if we assign the surface instance to a variable, e.g., **surf**, we can do other things to it, like add a color bar. Finally, please admire the use of a colormap (**cmap='jet'**).

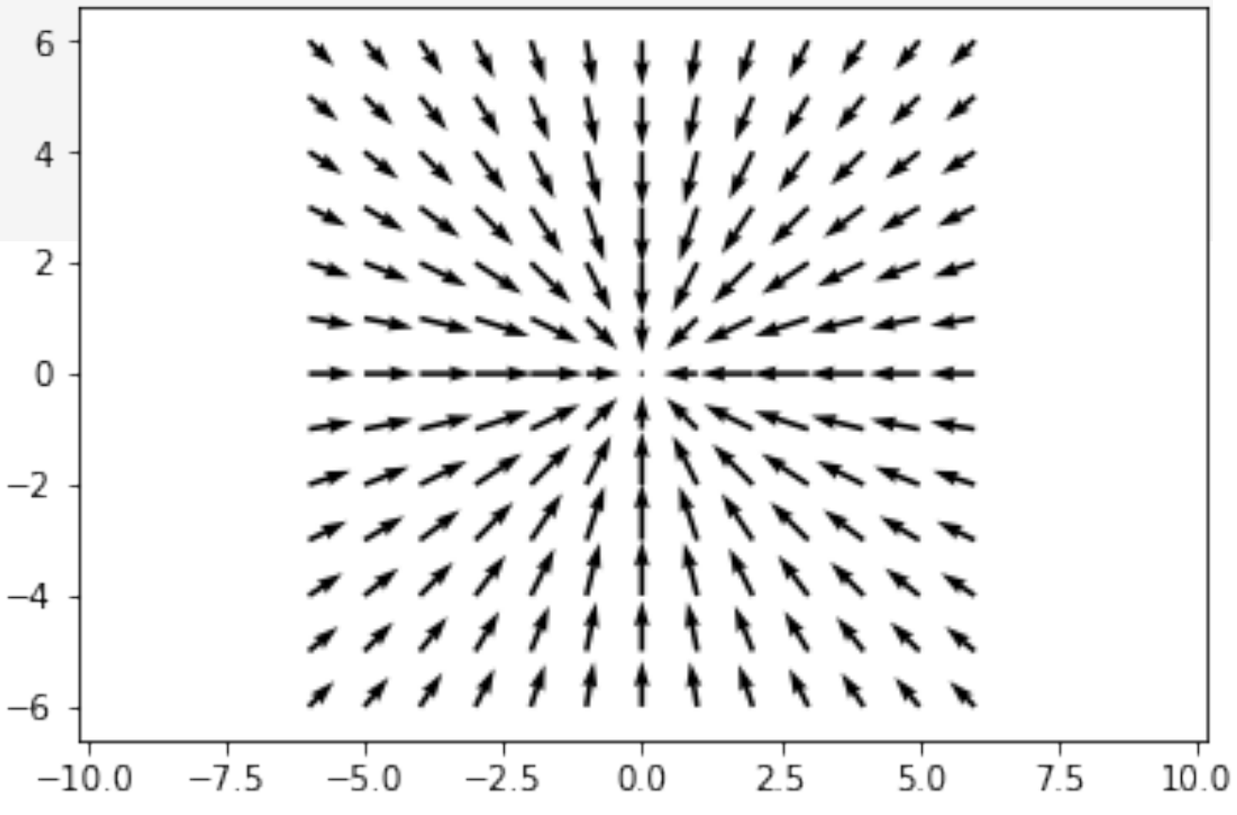
```
fig=plt.figure(4,(8,5)) # make a figure object
ax=Axes3D(fig) # give it the powers of an Axes3D object
surf=ax.plot_surface(X,Y,g,cmap='viridis') # use plot_surface
ax.set_xlabel('X') # and label the axes
ax.set_ylabel('Y')
ax.set_zlabel('Z')
bar=fig.colorbar(surf)
bar.set_label('Gravity in $\mu$gal');
```



Generate Vector plots in Python

Consider one more way to plot the data. Gravity data are inherently vectors, so we could plot them as arrows. This can be done using the **plt.quiver( )** method.

```
# we need to  redo what we already did.  but at lower resolution
z=3.
G=6.67e-11 # grav constant in Nm^2/kg^2 (SI)
R=2. # radius in meters
drho=500 # density contrast in kg/m^3

h=np.sqrt(X**2+Y**2) # get the horizontal distance from ground zero for x,y
# and make the g array
g=gravity(G,R,drho,h,z) # multipy by a million to get the units reasonable for the plots.

x=np.arange(-6,6.5,1) # range of x values
y=np.arange(-6.,6.5,1) # range of y values
X,Y=np.meshgrid(x,y) # make a meshgrid
h=np.sqrt(X**2+Y**2) # get the horizontal distance from ground zero for x,y
g=gravity(G,R,drho,h,z) # re-use our lambda function

#  now make the horizontal projections along X and Y
U=-X*g
V=-Y*g
# and plot
plt.quiver(x,y,U,V)
plt.axis('equal');
```



More on 3-D visualization in Python

Contour plots are really just a way to visualize something that is inherently 3D on a 2D surface. Think about our topographic map - the contour intervals are elevations and our brains can reconstruct the 3D world by looking at the contours on the map. But with computers we can visualize the 3D world in a more realistic manner. There are many 3D plotting packages that apply many different approaches to plot in 3D. For example, **mplot3d**, is a 3D toolkit of **matplotlib** that uses the same logic as for "regular" **matplotlib**. For more on this module, see:

http://matplotlib.sourceforge.net/mpl_toolkits/mplot3d/index.html

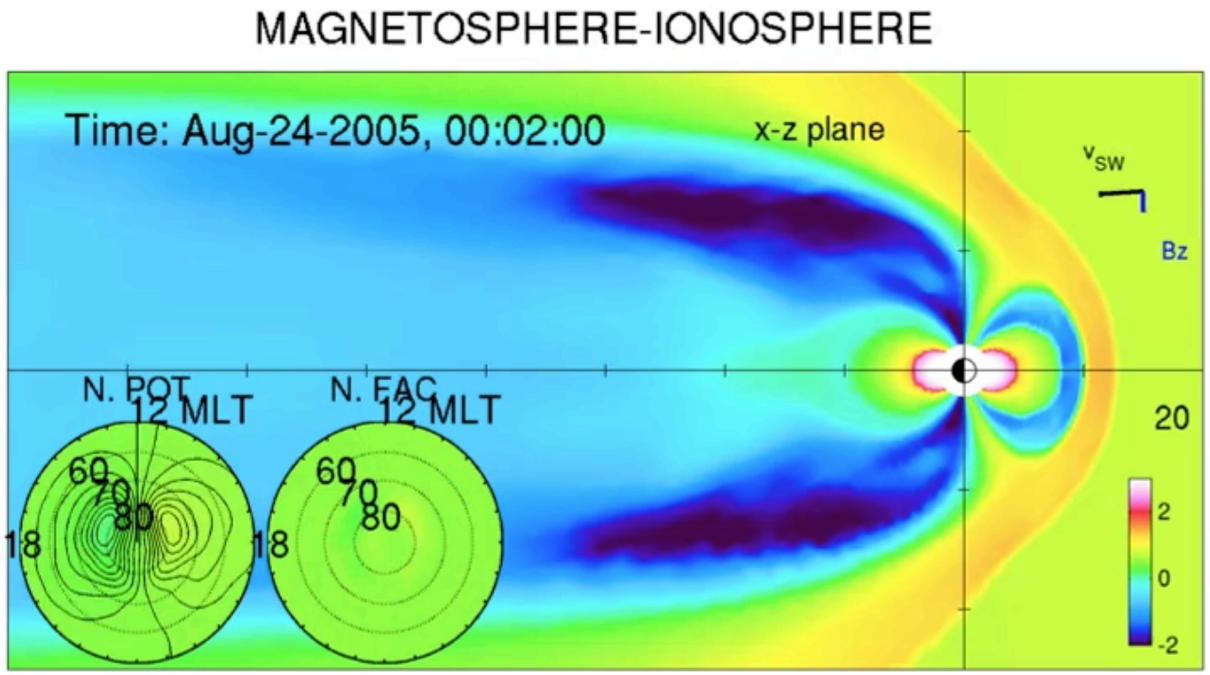
But for more 3D horsepower, there is a module called **mlab**, which is part of the **mayavi** package. See:
http://github.enthought.com/mayavi/mayavi/mlab.html

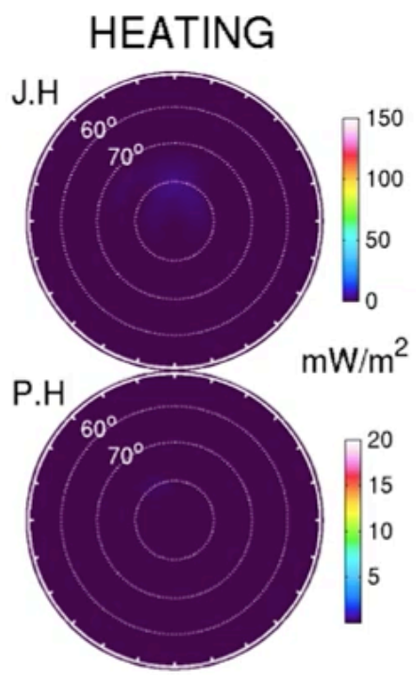
And then there is **Mayavi** itself, which comes with the Canopy Python Edition. This was way beyond what I know, but if you are curious, check out this website:

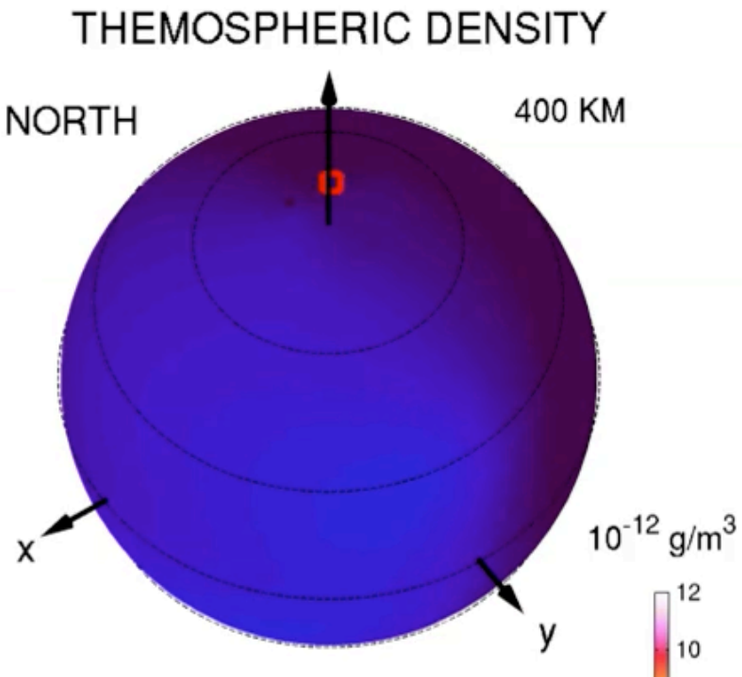
http://github.enthought.com/mayavi/mayavi/examples.html

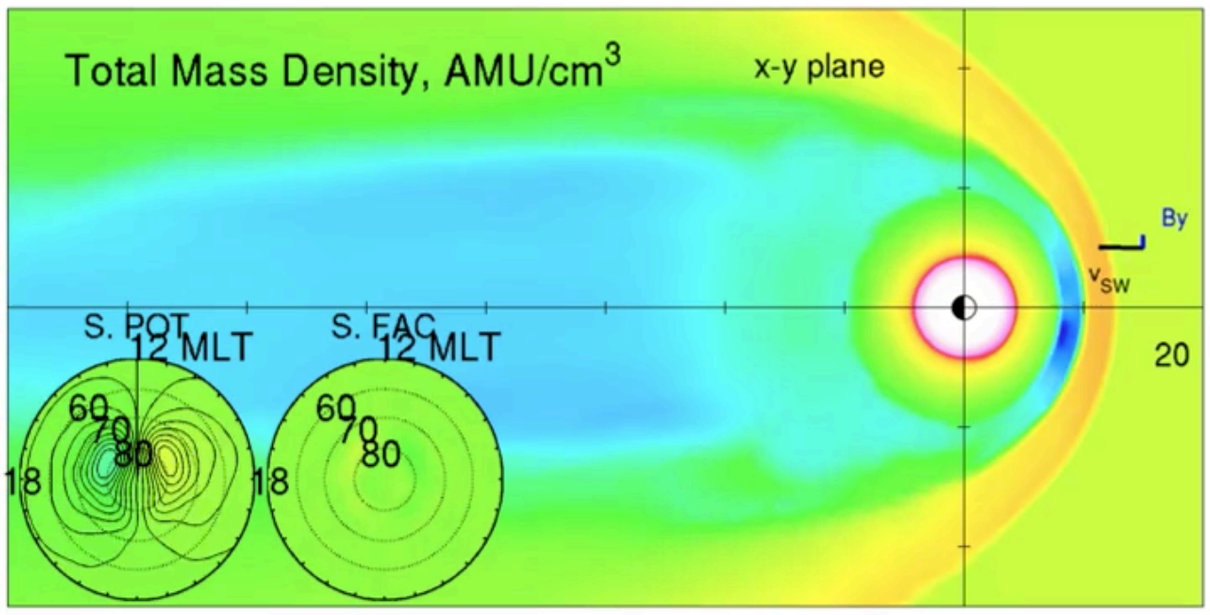
3D Plotting in Jupyter Notebooks¶
Until now, we have used **%matplotlib inline** to plot our matplotlib figures within the jupyter notebook. In order to plot a 3D figure within the jupyter notebook and allow interaction with a 3D plot, we must use this alternative: **%matplotlib notebook**

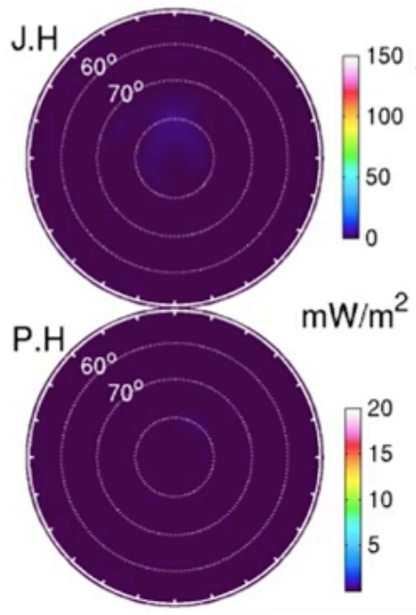
Animation: 1-D, 2-D and 3-D visualizations for a geomagnetic storm

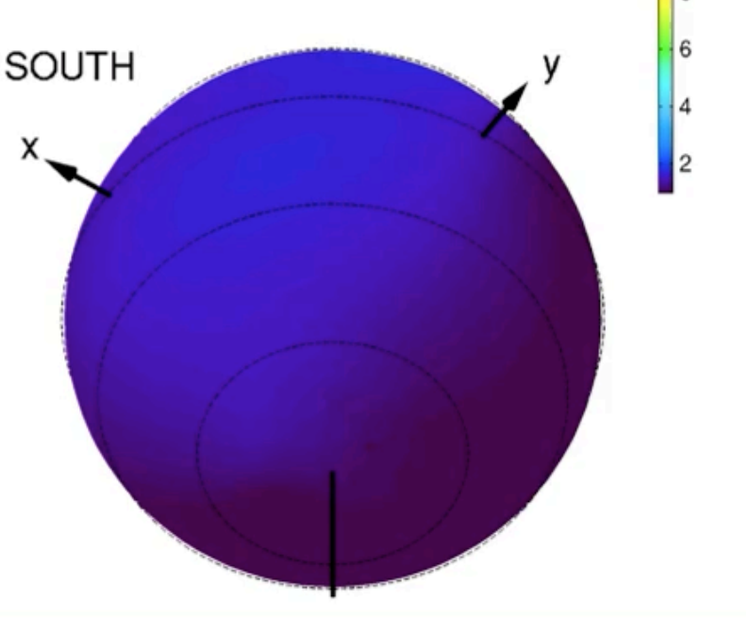


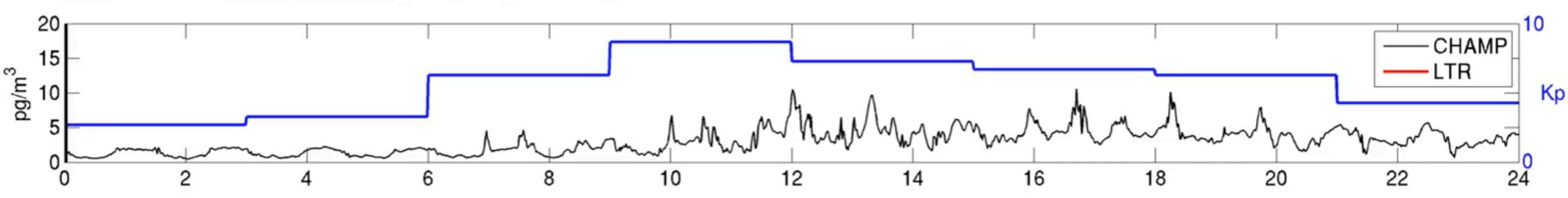




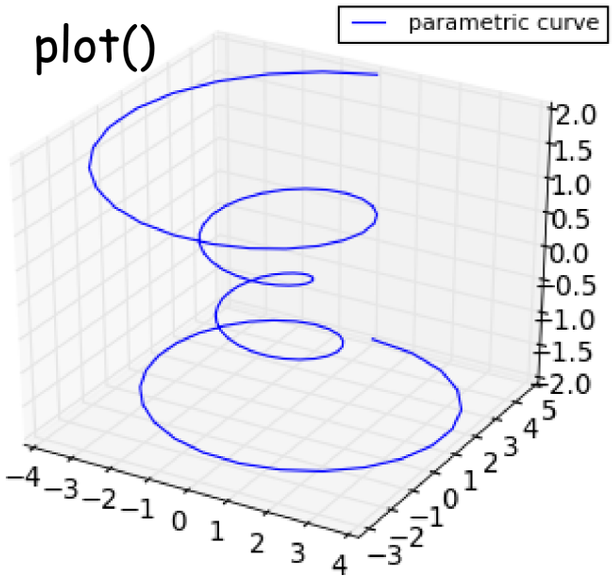




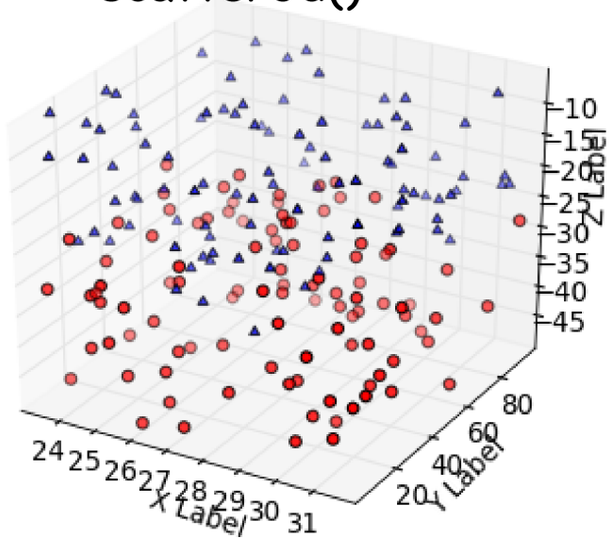




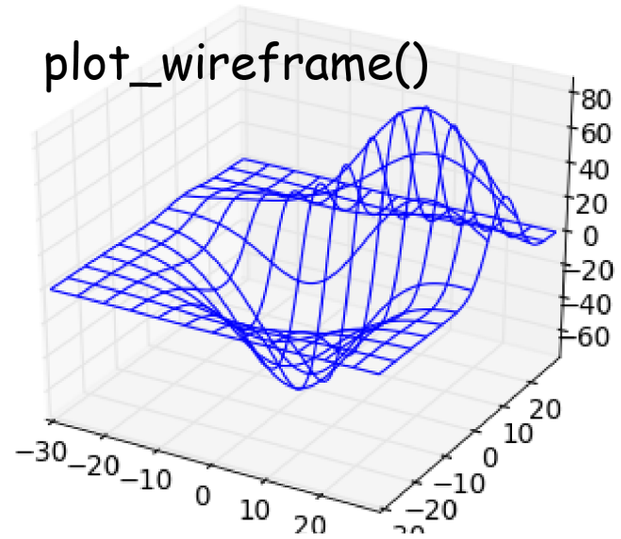
plot()



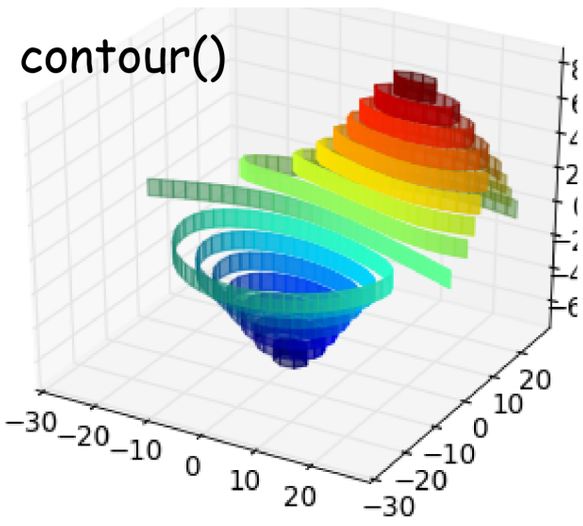
scatter3d()



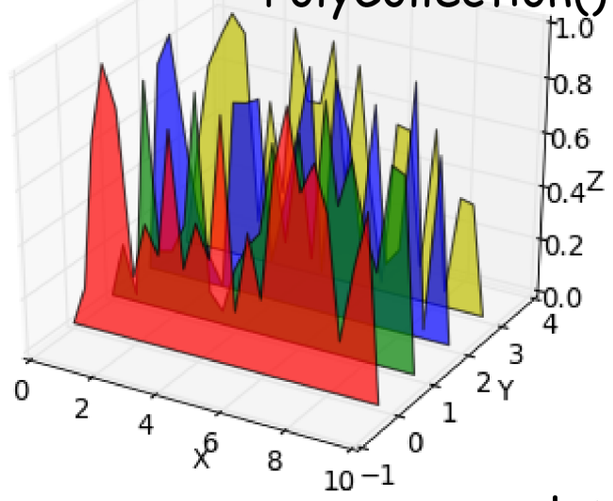
plot_wireframe()



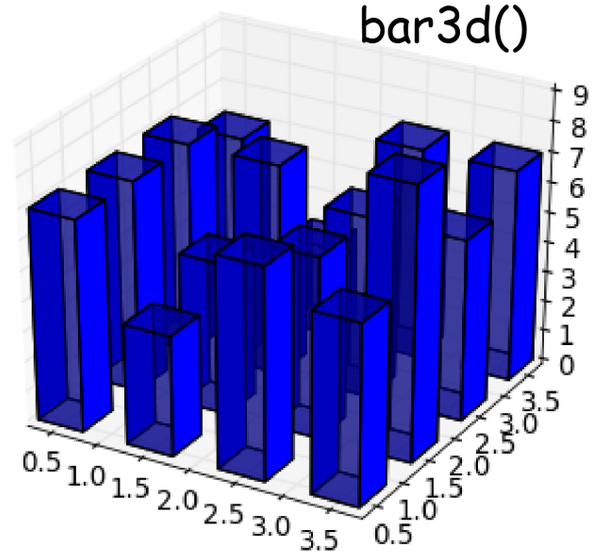
contour()



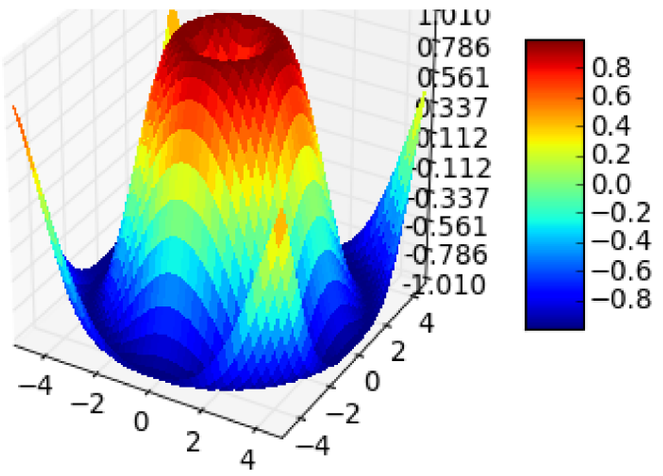
PolyCollection()



bar3d()



plot_surface()



Examples from mplot3d.Axes3D examples
http://matplotlib.org/1.3.0/gallery.html

Have fun!

Let’s make some 3-D color plots using matplotlib!