Introduction to Programming Lecture 11

- Instructor : Andrew Yarmola andrew.yarmola@uni.lu
- Course Schedule : Wednesday 14h00 - 15h30 Campus Kirchberg B21
- Course Website : sites.google.com/site/andrewyarmola/itp-uni-lux
- Office Hours : Thursday 16h00 - 17h00 Campus Kirchberg G103 and by appointment.

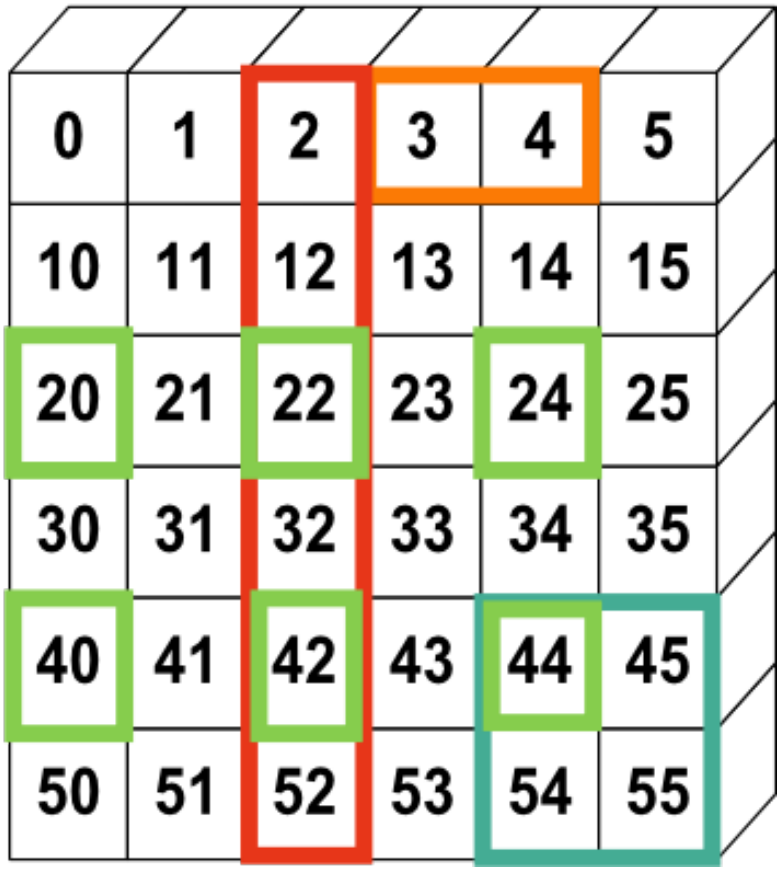
No class Wed Dec 7

Slicing and views

One can **slice** arrays in a very similar manner to python lists by using

`A[start_0 : end_0 : step_0, start_1 : end_1, step_1, ... ].`

```
>>> a[0,3:5]
array([3,4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```



For example :

In [43]:

```
import numpy as np

a = np.arange(4*4).reshape(4,4)
print(repr(a))
# just like with lists, we don't
# need to include start, end, or step
b = a[0:2,2:]
print(repr(b))
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
array([[2, 3],
       [6, 7]])
```

In [44]:

```
b = a[::2,::3]
print(repr(b))
```

```
array([[ 0,  3],
       [ 8, 11]])
```

Important : The arrays returned by a slice are **views** into the **data of the original**. Thus, if you **modify the data in a view you modify the original data!**

In [45]:

```
b[0,0] = 17
print(repr(b))
print(repr(a))
```

```
array([[17,  3],
       [ 8, 11]])
array([[17,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Important : A lot of other operations (such as broadcast) also share the data with the original! If you need to modify something, but you aren't sure if data is shared you can check with `numpy.may_share_memory(a,b)` and `numpy.shares_memory`

In [46]:

```
np.may_share_memory(a,b)
```

Out[46]:

```
True
```

In [47]:

```
np.shares_memory(a,b)
```

Out[47]:

```
True
```

or you can always just **make a copy!** (though you may not want to for efficiency reasons.)

In [48]:

```
c = b.copy()
np.may_share_memory(a,c)
```

Out[48]:

```
False
```

Assigning several values at once

You can use a view to assign a bunch of values at once :

In [49]:

```
print(repr(a))
b = a[::2,::3]
# assign all the same
b[...] = -75
print(repr(a))
```

```
array([[17,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
array([[-75,   1,   2, -75],
       [  4,   5,   6,   7],
       [-75,   9,  10, -75],
       [ 12,  13,  14,  15]])
```

In [50]:

```
# assign from another array
b[...] = np.zeros((2,2))
print(repr(a))
```

```
array([[ 0,  1,  2,  0],
       [ 4,  5,  6,  7],
       [ 0,  9, 10,  0],
       [12, 13, 14, 15]])
```

In [51]:

```
# assign using broadcasing
b[...] = np.array([57,180])
print(repr(a))
```

```
array([[ 57,   1,   2, 180],
       [  4,   5,   6,   7],
       [ 57,   9,  10, 180],
       [ 12,  13,  14,  15]])
```

Remark We will talk about the differnce between `...`, `:`, and others shortly

Fancy indexing

You can also do some fancy indexing with `numpy arrays` that goes beyond slicing. However, **fancy indexing returns copies not views!**

masks or boolean arrays

You can also specify a **boolean array** as the indexes you want as you can see in the image above. Another good example is :

In [52]:

```
# this is called a boolean mask
t = (a % 3 == 2)
print(repr(t))
```

```
array([[False, False,  True, False],
       [False,  True, False, False],
       [False, False, False, False],
       [False, False,  True, False]])
```

In [53]:

```
print(repr(a))
b = a[t]
print(repr(b))
# you can also use
# a[a % 3 == 2] = 7700
a[t] = 7700
print(repr(a))
```

```
array([[ 57,   1,   2, 180],
       [  4,   5,   6,   7],
       [ 57,   9,  10, 180],
       [ 12,  13,  14,  15]])
array([ 2,  5, 14])
array([[  57,    1, 7700,  180],
       [   4, 7700,    6,    7],
       [  57,    9,   10,  180],
       [  12,   13, 7700,   15]])
```

using tuples or lists

We can give coordinates for each axis as tuples or lists.

For example,

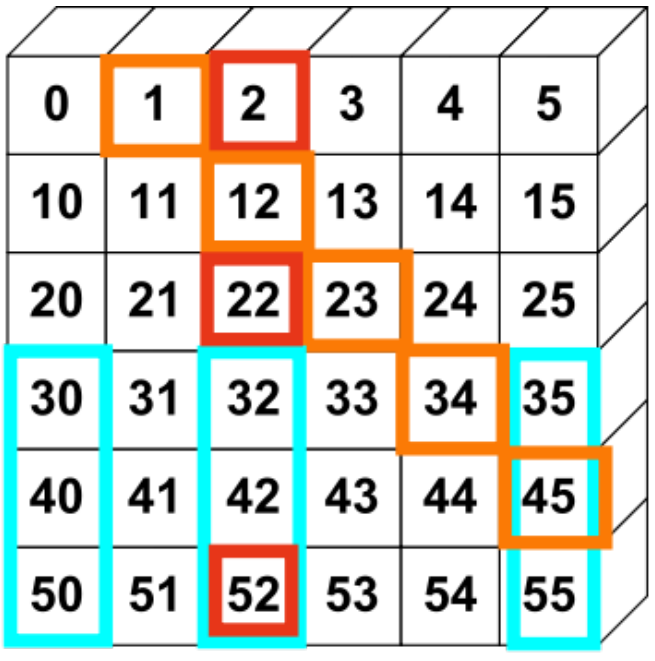
In [54]:

```
print(repr(a))
b = a[(1,3),[0,2]]
print(repr(b))
a[(1,3),(0,2)] = 8900
print(repr(a))
```

```
array([[  57,    1, 7700,  180],
       [   4, 7700,    6,    7],
       [  57,    9,   10,  180],
       [  12,   13, 7700,   15]])
array([   4, 7700])
array([[  57,    1, 7700,  180],
       [8900, 7700,    6,    7],
       [  57,    9,   10,  180],
       [  12,   13, 8900,   15]])
```

Here is a nice image from our text

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45]])
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
array([2,22,52])
```



Remark Note that `a[(1,2)]` and `a[(1,2),]` are different!

In [55]:

```
# same as a[1,2]
a[(1,2)]
```

Out[55]:

```
6
```

In [56]:

```
print(repr(a))
# same as a[(1,2),...]
a[(1,3),]
```

```
array([[  57,    1, 7700,  180],
       [8900, 7700,    6,    7],
       [  57,    9,   10,  180],
       [  12,   13, 8900,   15]])
```

Out[56]:

```
array([[8900, 7700,    6,    7],
       [  12,   13, 8900,   15]])
```

mixing fancy and normal sclicing

Notice that we can do the following

In [57]:

```
print(repr(a))
a[2:,(1,3)]
```

```
array([[  57,    1, 7700,  180],
       [8900, 7700,    6,    7],
       [  57,    9,   10,  180],
       [  12,   13, 8900,   15]])
```

Out[57]:

```
array([[  9, 180],
       [ 13,  15]])
```

Controlling the shape of array returned by fancy indexing

For fancy slicing, you can acutally give **arrays** as your indexing set :

In [58]:

```
print(repr(a))

# a (3,1) array
rows = np.array([[0],[1],[3]])
# a (1,3 array)
cols = np.array([2,3])

b = a[rows, cols]
print(repr(b))
```

```
array([[  57,    1, 7700,  180],
       [8900, 7700,    6,    7],
       [  57,    9,   10,  180],
       [  12,   13, 8900,   15]])
array([[7700,  180],
       [   6,    7],
       [8900,   15]])
```

In [59]:

```
np.may_share_memory(a,b)
```

Out[59]:

```
False
```

In essense, `numpy` will boradcast your indexing specification.

Named slices and special symbols

Since slicing can get complicated, we can separate slicing construction and actually taking a slice of an array.

The syntax `start:stop:step` is equivalnet to `slice(start,stop,step)`

In [60]:

```
x_slice = slice(None,3,2)
y_slice = slice(3,1,-1)

print(repr(a))

b = a[x_slice, y_slice]

print(repr(b))
```

```
array([[  57,    1, 7700,  180],
       [8900, 7700,    6,    7],
       [  57,    9,   10,  180],
       [  12,   13, 8900,   15]])
array([[ 180, 7700],
       [ 180,   10]])
```

The symbol `:` is equivalent to `slice(None, None, None)`. So, for example, to get a "layer" of a 3 dimensional array we can do :

In [61]:

```
a = np.arange(4*3*2).reshape(2,3,4)
all_slice = slice(None, None, None)

print(a)

b = a[:,1,:]

t = (all_slice, 1, all_slice)

c = a[t]

print(repr(b))
print(repr(c))
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
array([[ 4,  5,  6,  7],
       [16, 17, 18, 19]])
array([[ 4,  5,  6,  7],
       [16, 17, 18, 19]])
```

The advantage of this, is that `slice` can be used to consturct things **dynamically**. For example, if I want a layer form a specific axis :

In [62]:

```
def layer(array, axis = 0, idx = 0) :
    """ Returns array[:,:,...., idx, :, :, ....] where idx is in
    axis position specified. """
    all_slice = slice(None, None, None)
    t = [all_slice]*(axis) + [idx] + [all_slice]*(array.ndim - axis - 1)
    return array[t]
```

In [63]:

```
layer(a,1,2)
```

Out[63]:

```
array([[ 8,  9, 10, 11],
       [20, 21, 22, 23]])
```

special symbol ...

The **ellipsis** (`...`) can be used **once** when slicing an array to **fill** the appropirate number of `:`. For example :

In [64]:

```
b = a[...,2]

c = a[:,:,2]

print(repr(b))
print(repr(c))
```

```
array([[ 2,  6, 10],
       [14, 18, 22]])
array([[ 2,  6, 10],
       [14, 18, 22]])
```

new axis

Sometimes it is very useful to insert a new axis into your array. This can be accomplished using `numpy.newaxis`

In [65]:

```
a = np.arange(4)
print(repr(a))
print(a.shape)
```

```
array([0, 1, 2, 3])
(4,)
```

In [66]:

```
b = a[:, np.newaxis]

print(repr(b))
print(b.shape)
```

```
array([[0],
       [1],
       [2],
       [3]])
(4, 1)
```

In [67]:

```
a = np.arange(4*3*2).reshape(2,3,4)
print(a.shape)
b = a[:,np.newaxis,...]
print(b.shape)
```

```
(2, 3, 4)
(2, 1, 3, 4)
```

Takeaway : if you think you should be able to slice an array in a given way, you probably can.

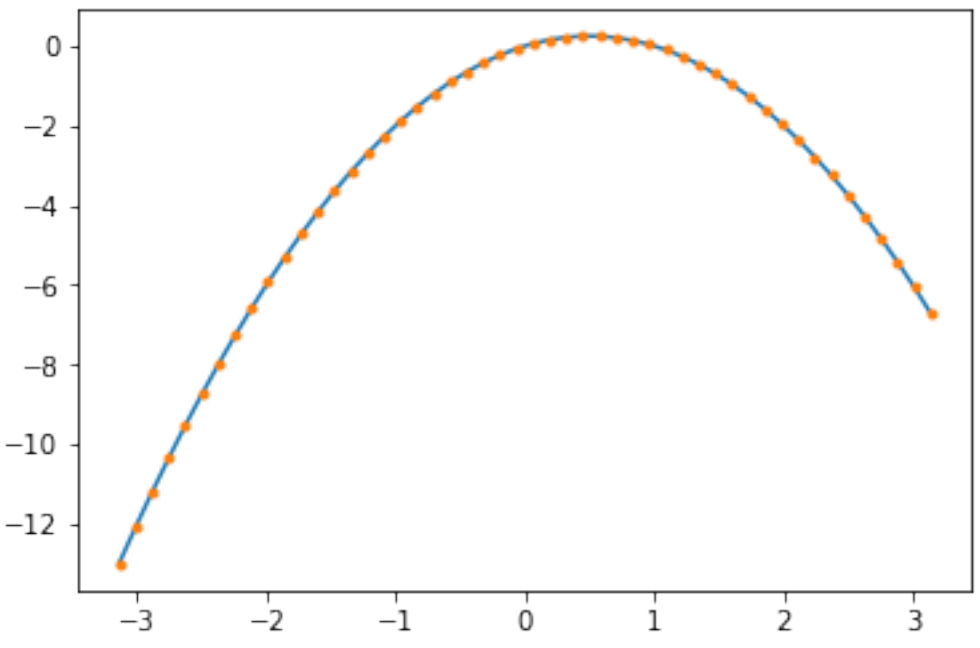
Plotting with `matplotlib`

Now that we are familiar with `numpy arrays`, we can use them to plot some very basic data.

We will use the `matplotlib` module for much of you plotting of data.

In [68]:

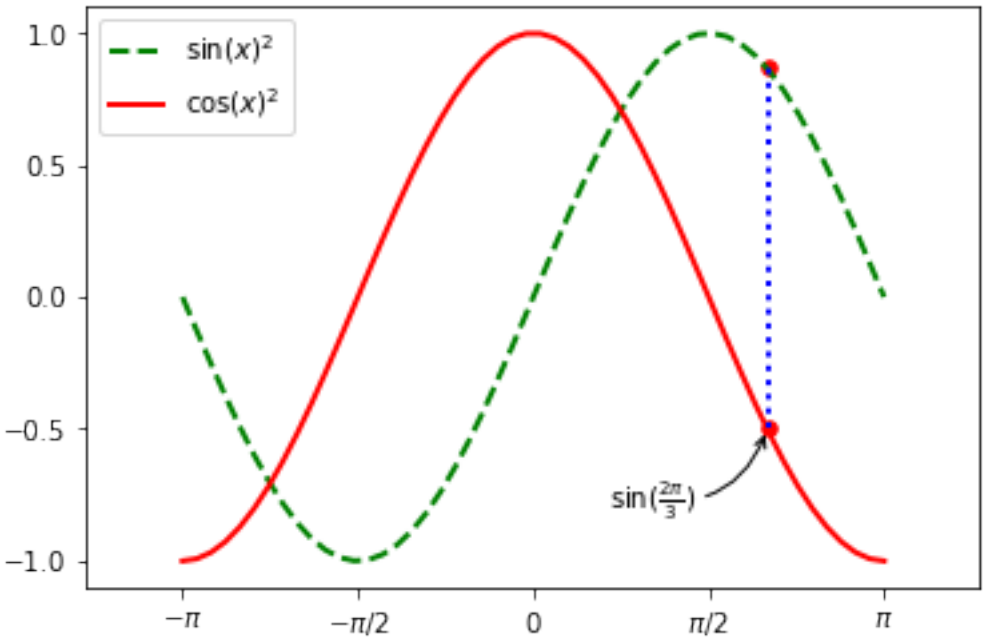
```
from matplotlib import pyplot as plt  # the tidy way
%matplotlib inline

x = np.linspace(-np.pi, np.pi, 50, endpoint=True)
y = x-x**2

plt.plot(x, y) # line plot
plt.plot(x, y, '.') # dot plot

plt.show()
```



In [69]:

```
f,g = np.sin, np.cos
y = f(x)
z = g(x)

# customized!
plt.plot(x, y, color="green",
         linewidth=2.0,
         linestyle="--",
         label = r'$\sin(x)^2$') # line plot

plt.plot(x, z, color="red",
         linewidth=2.0,
         linestyle="-",
         label = r'$\cos(x)^2$') # line plot

# set the x and y limits
plt.xlim(-4.0, 4.0)
plt.ylim(-1.1, 1.1)

# change the tick marks
plt.yticks([-1, -1/2, 0,1/2,1])

# chane ticks with labels
plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
    [r'$-\pi$',r'$-\pi/2$',r'$0$',r'$\pi/2$',r'$\pi$'])
```

```
# add a vertical line
t = 2 * np.pi / 3
plt.plot([t, t], [g(t), f(t)],
         color='blue', linewidth=2.0,
         linestyle=":")

# add some points using a scatter plot
plt.scatter([t,t],[f(t),g(t)], 30, color='red')

# add annotation
plt.annotate(r'$\sin(\frac{2\pi}{3})$',
             xy=(t, g(t)), xycoords='data',
             xytext=(-60,-30), textcoords='offset points',
             arrowprops = dict(arrowstyle="->",
                               connectionstyle="arc3,rad=.4"))

# add a legend
plt.legend(loc='upper left')

plt.show()
```

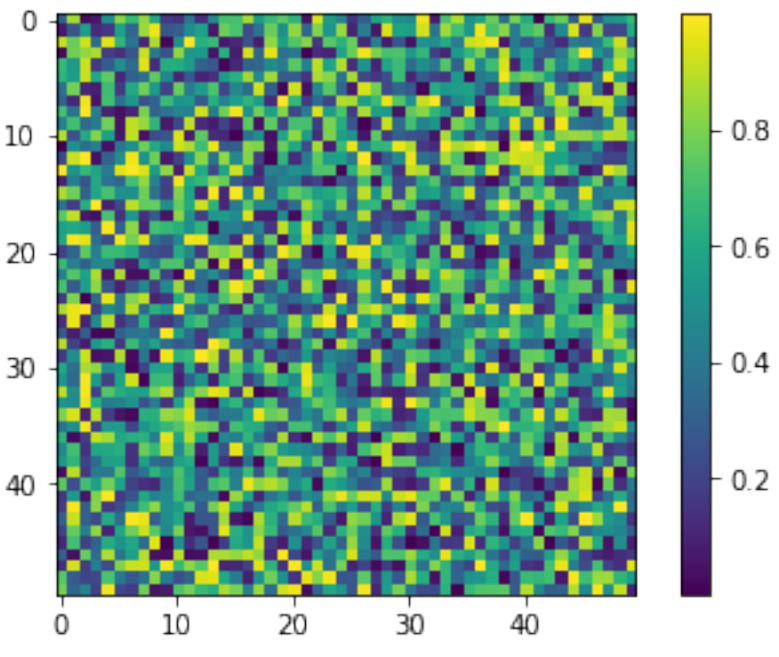


Remark the strings of the form `r'stuff'` are called **raw strings**, they exists so that `\` and other special characters are treated as just those characters an not like a LaTeX-style **escape** character.

Remark for LaTeX users You can always **annotate** points and mark things on plot, however, I suggest doing all LaTeX labels using **overpic** or **pinlabel** so that your document and image label fonts are the same.

In [70]:

```
# we can do images too!
image = np.random.rand(50, 50)
plt.imshow(image, interpolation='nearest')
plt.colorbar()
# we can save our work!
# format can be 'svg', 'pdf', 'png', etc
plt.savefig('random_color.eps', format='eps')
plt.show()
```



Remark Note that `pyplot.show()` will clear your drawing "canvas", so call `save` before calling `show`. There are also many other ways to save images.

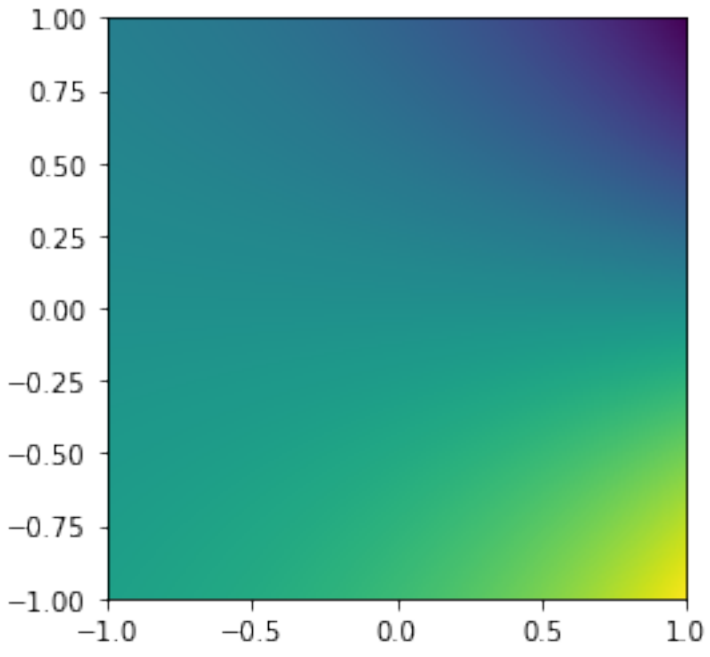
In [71]:

```
X,Y = np.meshgrid(np.array([0,1]),np.array([2,3]))

print(X)
print(Y)
```

```
[[0 1]
 [0 1]]
[[2 2]
 [3 3]]
```

In [72]:

```
# manually clear canvas
plt.clf()
N = 1000
axis_x = np.linspace(-1,1,N)
axis_y = np.linspace(-1,1,N)

X,Y = np.meshgrid(axis_x,axis_y)

def f(x,y) :
    return (np.exp(x + 1j * y)).imag

image = f(X,Y)

plt.imshow(image, extent=(-1,1,-1,1))
plt.show()
```



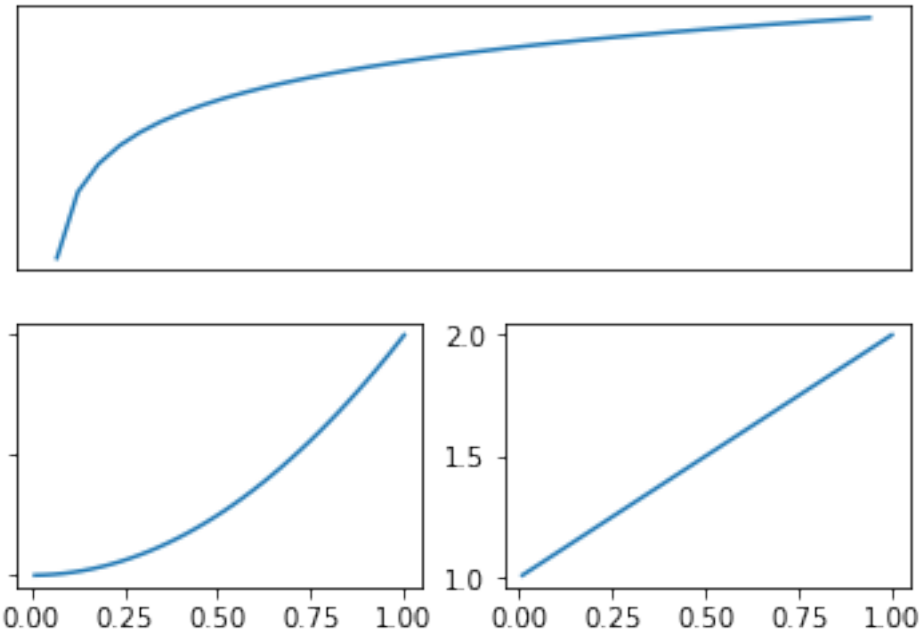
Remark there are plenty of libraries for plotting complex functions, such as `mpmath`. Above, the color is just a third dimension. Other libraries use color for the imaginary part and brightness for magnitude.

Subfigures

If you need to arrange several different charts in one place, you can use subfigures. Here is an example.

In [73]:

```
# the 2 indicates take have the height,
# the next 1 indicates take full width
# the last 1 indicates piece to choose
one = plt.subplot(2, 1, 1)
two = plt.subplot(2, 2, 3)
three = plt.subplot(2, 2, 4)
one.xaxis.set_major_locator(plt.NullLocator())
one.yaxis.set_major_locator(plt.NullLocator())

x = np.linspace(0.01,1,40)
one.plot(x,np.log(x))
two.plot(x, x**2)
three.plot(x,1+x)

plt.show()
```

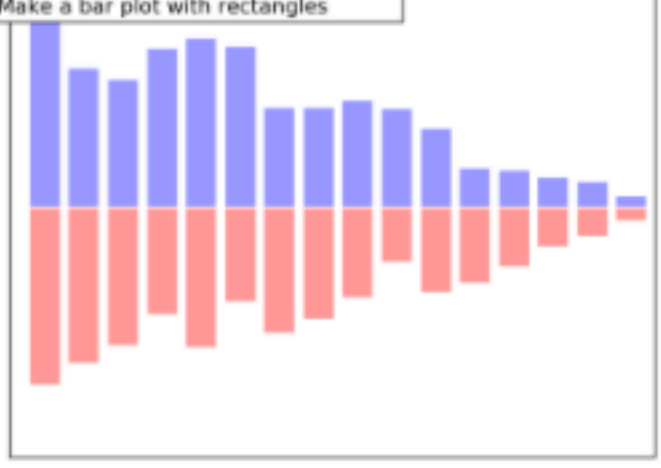


Other types of plots

`matplotlib` has many other different kinds of plots. You will explore these in the homework.

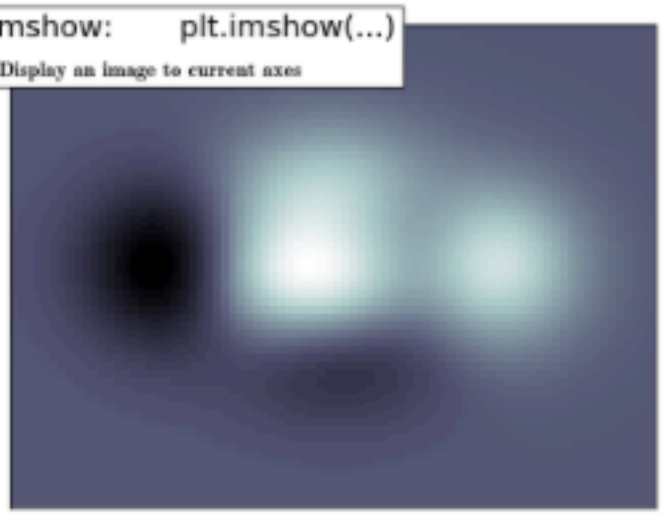
Bar Plot: plt.bar(...)

Make a bar plot with rectangles



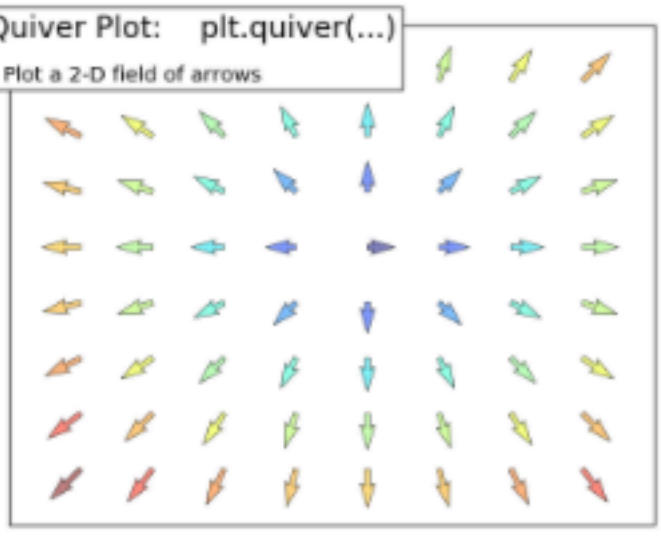
Imshow: plt.imshow(...)

Display an image to current axes



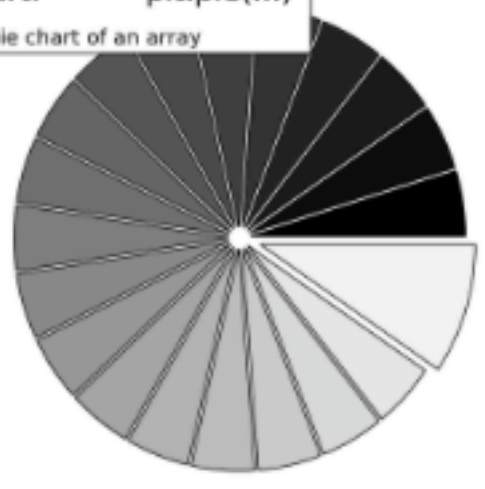
Quiver Plot: plt.quiver(...)

Plot a 2-D field of arrows



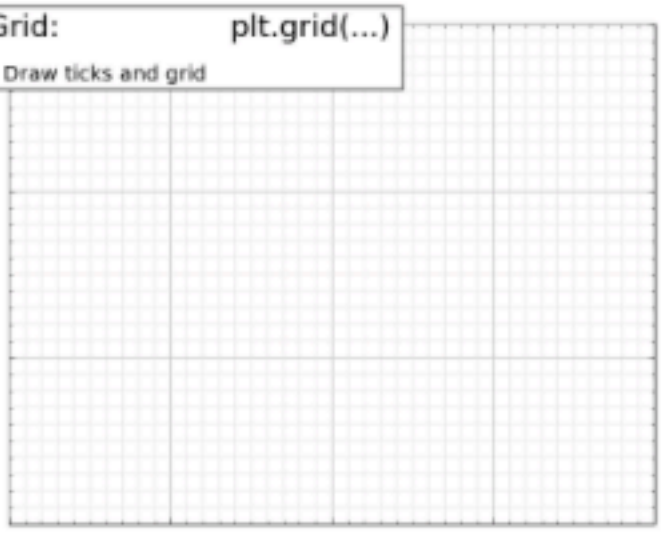
Pie Chart: plt.pie(...)

Make a pie chart of an array



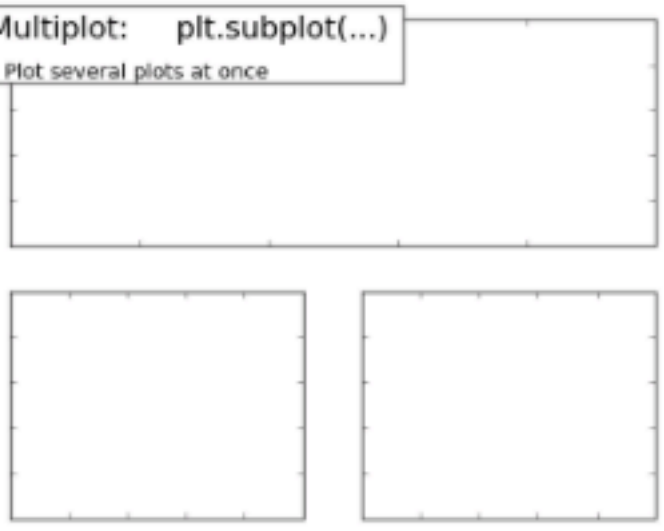
Grid: plt.grid(...)

Draw ticks and grid



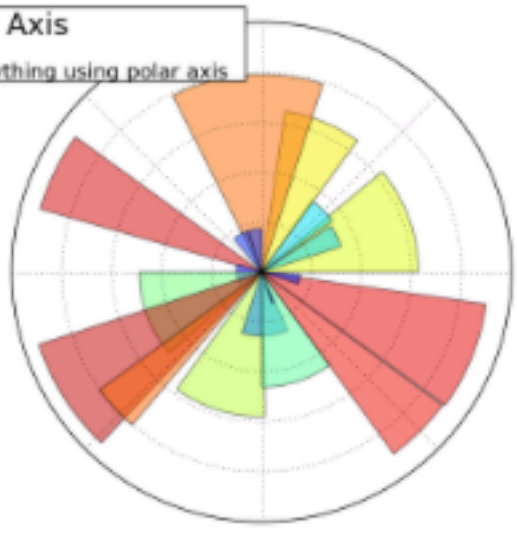
Multiplot: plt.subplot(...)

Plot several plots at once



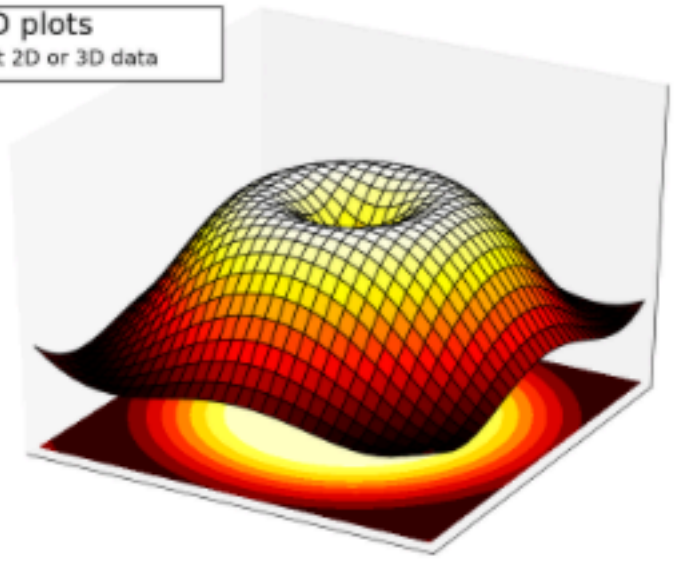
Polar Axis

Plot anything using polar axis



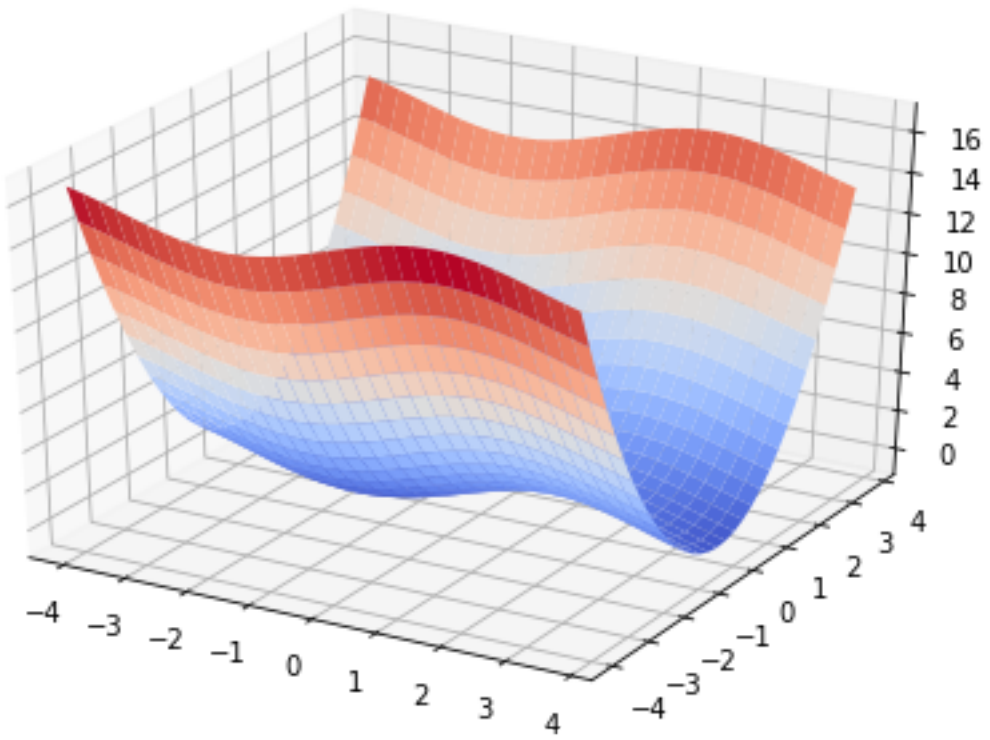
3D plots

Plot 2D or 3D data



In [74]:

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = Axes3D(fig)
x = np.arange(-4, 4, 0.25)
y = np.arange(-4, 4, 0.25)
X, Y = np.meshgrid(x, y)
Z = (np.sin(X) + Y**2)

ax.plot_surface(X, Y, Z,
                rstride=1, cstride=1,
                cmap='coolwarm', linewidth=0.1)
plt.show()
```



Plotting (3D) with Mayavi

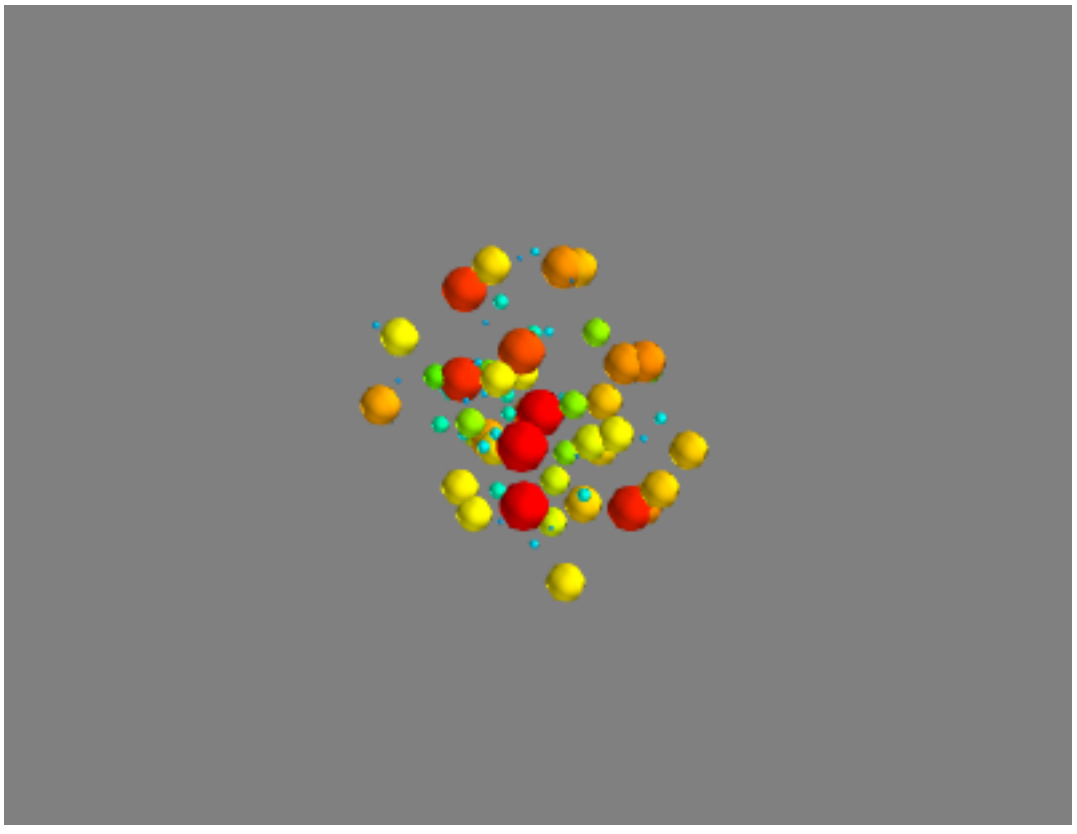
You can use `matplotlib` tools for 3D plots, but (from what I am told) Mayavi is a much better alternative, however **there might be some difficulties installing it**. I will provide instructions in the homework.

In [ ]:

```
import numpy as np
from mayavi import mlab
```

In [ ]:

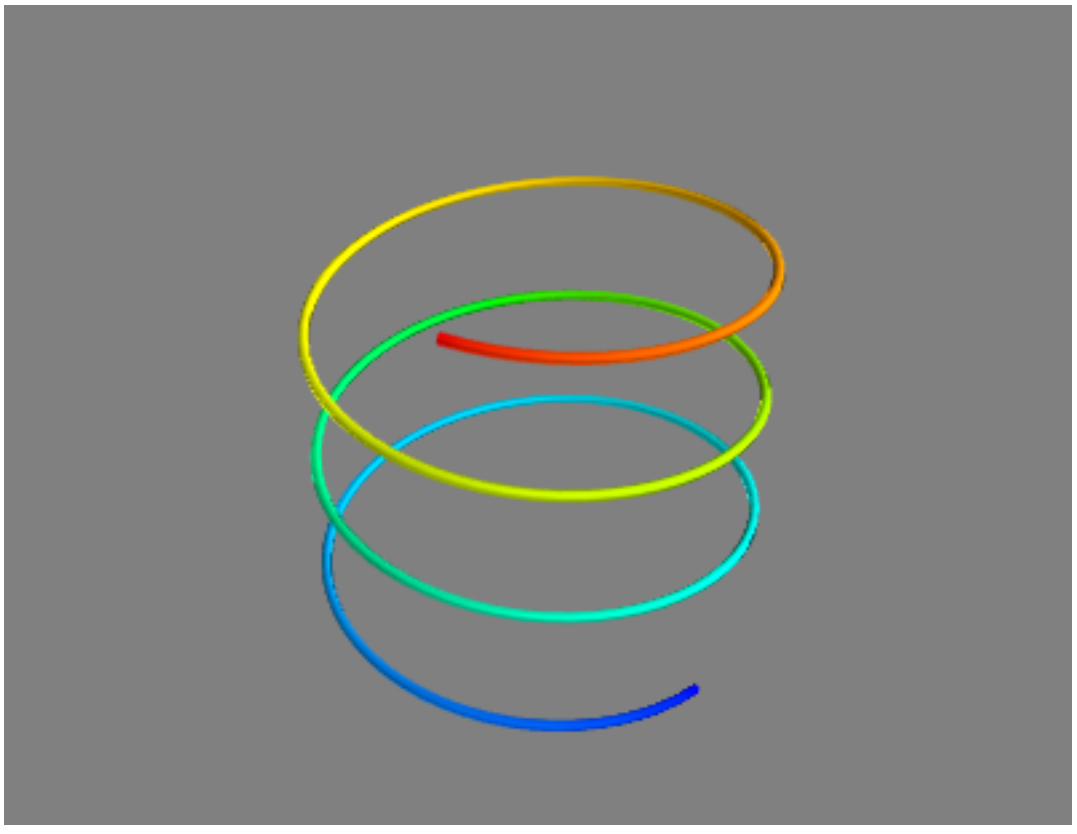
```
x, y, z, value = np.random.random((4, 40))
mlab.points3d(x, y, z, value)
mlab.savefig('rand_points.png')
```



Remark In the lecure notes I am just showing you static images, but in reality, these are interactive.

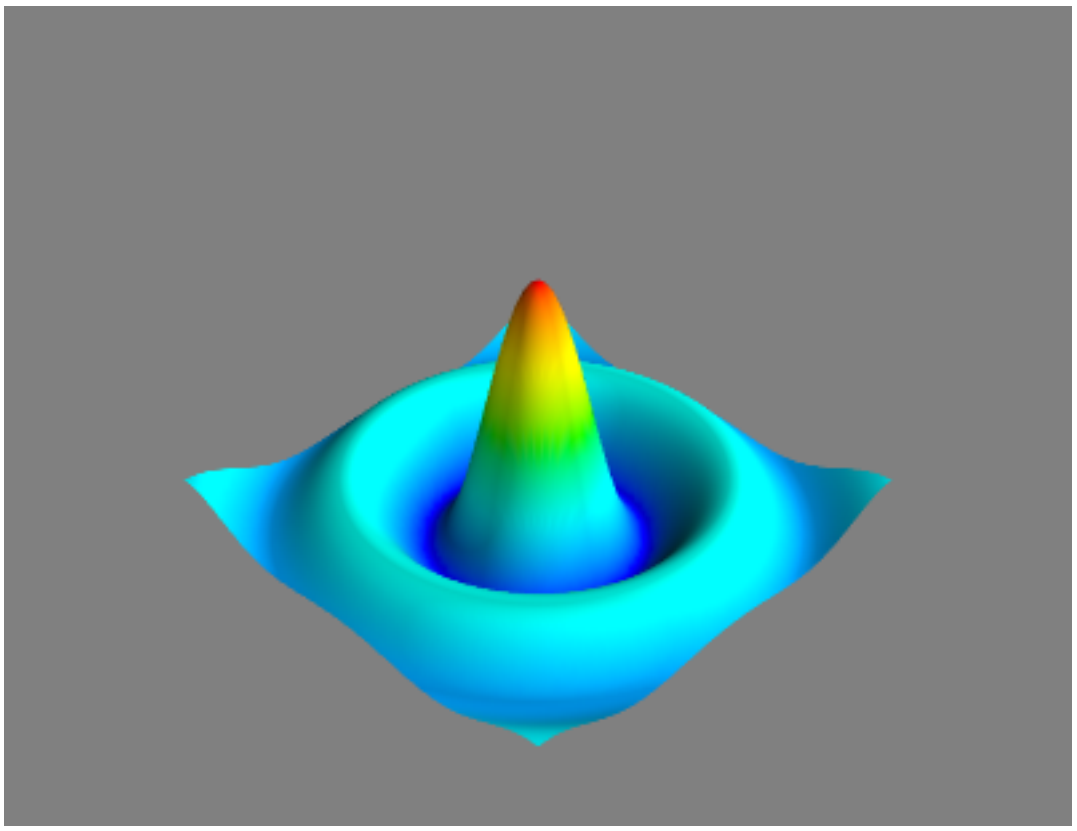
In [ ]:

```
mlab.clf()  # Clear the figure
t = np.linspace(0, 20, 200)
mlab.plot3d(np.sin(t), np.cos(t), 0.1*t, t)
mlab.savefig('spiral.png')
```



In [ ]:

```
mlab.clf()
# mgrid is like mesh grid but in one line
# then 100j means divide in 100 steps insead
# of taking steps of size 100
X, Y = np.mgrid[-10:10:100j, -10:10:100j]
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)/R
mlab.surf(Z, warp_scale='auto')
mlab.savefig('surface.png')
```



Animations and GIFs

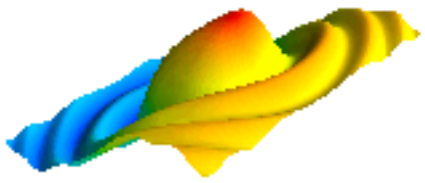
There are several tools for making animations, but the easiest one to use might be `moviepy`. However, this is the first time I am using it, so maybe there are better options.

In [ ]:

```
import numpy as np
import  moviepy.editor as mpy
```

In [ ]:

```
duration= 2 # duration of the animation in seconds (it will loop)

fig_myv = mlab.figure(size=(220,220), bgcolor=(1,1,1))

X, Y = np.mgrid[-2:2:200j, -2:2:200j]

Z = lambda p : np.sinc(X**2 + Y**2) + np.sin(X + p)

def make_frame(t):
    mlab.clf() # clear the figure (to reset the colors)
    mlab.mesh(Y, X, Z(2 * np.pi * t / duration), figure=fig_myv)
    return mlab.screenshot(antialiased=True)

animation = mpy.VideoClip(make_frame, duration=duration)
animation.write_gif("sinc.gif", fps=20)
mlab.close()
```



In [ ]:

```
# import the game of life step evolution function
# from homework code
from gof import evolution

board = np.random.randint(2, size=(30,30))
num_frames = 30
frames = []

for _ in range(num_frames) :
    board = evolution(board)
    frame = np.kron((1-board)*255.0, np.ones((10,10)))
    frames.append(frame)

animation = mpy.ImageSequenceClip(frames, fps=2)
animation.write_gif('game_of_life.gif')
```