Making animations with Python

The following page contains a few examples of animations you can make with Python:

http://matplotlib.org/1.5.3/examples/animation/index.html

We will look at two types of animations: a preexisting sequence of images using the method **`ArtistAnimation`** and animating an object calculated in a function using the method **`FuncAnimation`**. Both are methods of the **`matplotlib.animation`** package. Their respective documentation can be found there:

http://matplotlib.org/api/animation_api.html

Note that for each example below, you can either display your animation with **`plt.show()`** or save it in some movie format (I use mp4).
The following script generates a sequence of 50 images containing a normal random distribution of values on a grid 30x30:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig = plt.figure()

ims = []
for add in np.arange(50):
    base = np.random.randn(30,30)
#    ims.append((plt.imshow(base),))
#    ims.append((plt.pcolor(base),))
    ims.append((plt.pcolormesh(base),))

imani = animation.ArtistAnimation(fig, ims, interval=50,
repeat=False)
imani.save('random.mp4')
```

Commenting out the last statement and replacing it with a **`plt.show()`** will play the animation live. The method **`ArtistAnimation`** plays the sequence **`ims`**, called also an *Artist*. The sequence **`ims`** contains the individual images as a Python list, note that you can build an Artist sequence from different plotting calls (**`pcolor`**, **`imshow`** and **`pcolormesh`** in the examples above).

The method **`FuncAnimation`** allows you to create an animation from a function that can generate and update a plot. The plot could be any graphic object in Python. In the

examples below I will show you how it works for a histogram, a simple line and a scatter plot.

The script below shows an animation of 20 frames, each frame is a histogram of 100 random numbers generated from a normal distribution:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

number_of_frames = 20
n_data_perframe = 100
data = np.random.randn(number_of_frames,n_data_perframe)

def update_hist(num, data):
#    plt.cla()
    print(num)
    plt.hist(data[num,:])

fig = plt.figure()
hist = plt.hist(data[0,:])

animation = animation.FuncAnimation(fig, update_hist,
frames=number_of_frames, repeat=False,fargs=(data, ) )
```

Inside **`FuncAnimation`**, the iteration over the number of frames is passed in **`frames`**. The function **`update_hist()`** generates a new histogram based on a new set of data passed with **`fargs`**. Check online what the other arguments are/can be. A new histogram is generated at each function call, but it is possible to freeze the plot environment for all calls as shown in other examples below.

The following script shows you how to animate a line with **`FuncAnimation`**. The line is a **`np.sin()`** function updated inside **`update_line()`** every time the function is called by **`FuncAnimation`**. The update consists in a horizontal shift set by variable **`num`**.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

number_of_frames = 100
data = np.arange(0, 2*np.pi, 0.01)

fig = plt.figure()

def update_line(num):
    plt.cla()
```

```
    plt.plot(np.sin(data + num/10.0))

animation = animation.FuncAnimation(fig, update_line, num-
ber_of_frames,interval=1,repeat=False)
```

The problem with the approach above is that the function **update_line()** generates a brand new plot (axis, frame, lines, etc…) at every function call. This could result in slowing down your animation. One way around this is to update only the elements of a plot which need to be updated. The example below shows the same line animation where only the line us updated, while everything else stays the same. In order to do this we have to start making changes in Python *classes* for plotting operations, i.e. something we haven't discussed yet. But the example below should be clear enough: every Python object contains sub objects which can be altered individually. The example below shows how to alter the line in a plot using an instance of a plot:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

number_of_frames = 100
data = np.arange(0, 2*np.pi, 0.01)

fig, ax = plt.subplots()
line, = ax.plot(data, np.sin(data))

def update_line(num):
    line.set_ydata(np.sin(data + num/10.0))  # update the data
    return line,

animation = animation.FuncAnimation(fig, update_line, num-
ber_of_frames,interval=1,repeat=False)
```

It should be noted that the function **update_line()** now only returns the modified line, this is only for your own clarity because the whole script would work equally well without the **return** statement.

The following example shows how you can use **FuncAnimation** to generate animations which involve calculations in a separate function for each iteration. Similar to the line example above, we will use, and update an instance of **scatter()**, i.e. a scatter plot. The initial plot is a dot located at **xpos, ypos**, and its position is updated using the method **set_offsets** of the scatter plot instance **im**. The function **newypos()** updates the position of the dot, shifting vertically by an amount **dypos**:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

number_of_frames = 49
fig, ax = plt.subplots()

xpos,ypos=0.5,0.
dypos=1./50.

plt.xlim(0,1)
plt.ylim(0,1)
im = ax.scatter(xpos, ypos)

def newypos(i):
    global ypos
    ypos=ypos+dypos

def update_point(num):
    newypos(num)
    im.set_offsets((xpos,ypos))
    return im,

animation = animation.FuncAnimation(fig, update_point, num-
ber_of_frames,interval=1,repeat=False)
```

Project 3 (Due date Thursday November 10th 12 a.m.)

For this project you will study a simple model of burning forest with regeneration using the concept of **cellular automaton**. A cellular automaton is an autonomous system that lives on a lattice, which could be of any dimension (usually it is a regular lattice, but not necessarily), which evolution with time is controlled by a simple set of rules. The rules are often microscopic (i.e. they apply at the grid cell level and its immediate neighbors) but the system often develops macroscopic behaviors and patterns which makes it resemble realistic behavior in realistic systems. For that reason, cellular automata are often used to describe natural phenomena whose complex structure are very difficult to approach with quantitative analysis. They can be seen as a simple discrete mathematical idealization of natural systems.

You can read more on this concept there:

http://natureofcode.com/book/chapter-7-cellular-automata/

and the wikipedia page:

https://en.wikipedia.org/wiki/Cellular_automaton

A well known example of cellular automaton is the game of life (https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life), but this concept is used in many areas such as biology, sociology, physics (e.g. to model phase transitions), economics, finance, and much much more.

Objective: For project 3 you will write you own cellular automaton code which describes the destruction by fire, and regeneration of a forest. The basic idea is described there:

https://en.wikipedia.org/wiki/Forest-fire_model

The rules of your forest burning cellular automaton are the following:

1. A burning cell turns into an empty cell
2. A tree will burn if at least one neighbor is burning
3. A tree ignites with probability `f` even if no neighbor is burning
4. An empty space fills with a tree with probability `p`

These operations should be done simultaneously for each iteration. Your code should be called **`forestfire.py`** and should use `numpy` arrays. The animation file should be called **`forestfire.mp4`**. The parameters you will use are `f=0.01` and `p=0.001`. Your forest should be on a grid that is at least `100x100`, but it can be bigger, as long it runs within a reasonable time. Your animation should go over `300` iterations. For each iteration, you have to perform simultaneously the set of rules 1-4 above, and at the beginning, the grid points (cells) of your forest should be initially populated with trees with a probability of `0.55` per cell.

The original research paper was published in `Physical Review Letters` in 1992 and is available there:
http://journals.aps.org/prl/abstract/10.1103/PhysRevLett.69.1629

You need to be connected from a UBC server to access it for free (or un the UBC VPN).

Tips: There is a Python solution to this problem which uses `dictionaries` which can be found there:

https://rosettacode.org/wiki/Forest_fire#Python

It is a good exercise to reverse engineer this code first, get some inspiration from it, especially if you have some issues with what the cellular automaton should really do. Then

try to make a movie out of it (to replace this ultra-basic text based output by a nice animation), and then write your own code with numpy arrays. You will have to optimize it so it does not take ages to run through various loops (if you use loops). Think how you could use numpy array filtering (i.e. subarray selection based on conditional statements) for speed optimization. Don't let your code waste time checking the status of cells which you know no operation are to be done in them until they change status.