Today’s lecture

- **Animation** using `matplotlib.animate.FuncAnimation()`
  
  **Examples:**
  
  - A moving sine wave
  - The logistic map
  - Van der Pol oscillator
  - 3D animation: the Lorenz equations
  - Julia sets
  - Particles in a box

- Code on course website:
  
  [http://sam-dolan.staff.shef.ac.uk/mas212/code](http://sam-dolan.staff.shef.ac.uk/mas212/code)
A note on Python syntax

>>> (1,2)  # a tuple
(1, 2)
>>> 1,2  # If I omit the brackets I still get a tuple
(1, 2)
>>> # How do I write a tuple with just one element?
>>> (1)  # This doesn’t work ...
1
>>> (1,) # This is how to write a tuple with just one element
(1, )
>>> # Functions often return lists or tuples
>>> # For example, plot() returns a list of Line2D objects.
>>> x = np.linspace(0, 4, 100)
>>> l1 = plt.plot(x, np.sin(x))  # compare these lines
>>> l2, = plt.plot(x, np.sin(x))  # and note the comma
>>> type(l1)  # l1 is the list itself
<class 'list'>
>>> type(l2)  # l2 is the first element of the list
<class 'matplotlib.lines.Line2D'>
Animation

Why? Animations can be used to . . .

- . . . help us understand systems that evolve in time
- . . . understand solutions of ODEs
- . . . impress people! (use in moderation)
Animation

What will we use?

- The `matplotlib.animation` module
- The `FuncAnimation()` function

Code examples here: 
  [http://matplotlib.org/examples/animation/](http://matplotlib.org/examples/animation/)
Animation

For Spyder:
- Menu: Preferences -> IPython Console -> Graphics -> Backend
- Set Backend: to Automatic

For Jupyter notebook:
- `%matplotlib notebook`
First example: animating a sine wave

- Here is the equation for a sine wave with period 1 moving to the right:
  \[ f(x, t) = \sin(2\pi(x - t)) \]

- We will make an animation from a list of frames, shown in order, like a flip-book.

- In the \( k \)th frame, we plot
  \[ y_k(x) = \sin(2\pi(x - t_k)), \quad t_k = k/100. \]
```python
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation

fig = plt.figure()
ax = plt.axes(xlim=(0, 2), ylim=(-2, 2))
line, = ax.plot([], [], lw=2)  # an empty line

def init():  # Initialize with a blank plot
    line.set_data([], [])  # line is a global variable
    return line,

def animate(i):  # Plot a moving sine wave
    x = np.linspace(0, 2, 1000)
y = np.sin(2 * np.pi * (x - 0.01 * i))
    line.set_data(x, y)
    return line,

anim = animation.FuncAnimation(fig, animate, init_func=init,
                              frames=200, interval=20, blit=True)

plt.show()
```
Import the necessary modules:

```python
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation
```

Set up a Figure, an AxesSubplot (a blank plot inside the figure), and an empty line which can be changed later

```python
fig = plt.figure()
ax = plt.axes(xlim=(0, 2), ylim=(-2, 2))
line, = ax.plot([], [], lw=2)  # an empty line
```

Here line is a Line2D object. This is the only element which will change during the animation; the rest of the plot stays the same.
The initialization function `init()` is called before the animation begins

We set the line data to empty lists: no line will be drawn yet

```python
def init():
    line.set_data([], [])                      # line is a global variable
    return line,
```

This function must return a list of objects: in this case just the Line2D object
**FuncAnimation: step-by-step**

- An animation consists of a sequence of frames.
- This function makes frame $i$ of the animation:

```python
def animate(i):
    # Plot a moving sine wave
    x = np.linspace(0, 2, 1000)
    y = np.sin(2 * np.pi * (x - 0.01 * i))
    line.set_data(x, y)
    return line,
```

- This function is called repeatedly, with $i$ incremented, each time a new frame is to be drawn.
- The function should return a tuple of those elements in the plot which have been changed. Those elements will then be re-drawn.
**FuncAnimation: step-by-step**

- Now we create the FuncAnimation object, passing the figure object, and the `animate()` and `init()` functions that we defined.

```python
anim = animation.FuncAnimation(fig, animate,
     init_func=init, frames=200, interval=20, blit=True)
plt.show()
```

- We also set the total number of frames to 200, and the delay between frames to 20 milliseconds.

- The `FuncAnimation` object must persist, so we assign it to a variable (anim).

- Setting `blit` to True ensures that only those elements of the plot that have changed will be redrawn. This makes the animation fast & smooth.
‘Blitting’

**Bit blit** (bit-boundary block transfer) is a computer graphics operation in which several bitmaps are combined into one using a raster operator.

- e.g., if a line changes, but the background axes stay the same, we don’t want to redraw the entire image.
- ‘Blitting’ optimizes drawing ⇒ animations are smoother & faster.
The bifurcation diagram for the logistic map looks like this:

Let’s increase the parameter $r$ slowly . . .
fig = plt.figure()
ax = plt.axes(xlim=(0,50), ylim=(0.0, 1.0))
line, = ax.plot([], [], 'ro')
def logisticmap(r=2.0, N=100):
    xs = 0.5*np.ones(N)
    for i in np.arange(N-1):
        xs[i+1] = r*xs[i]*(1.0-xs[i])
    return xs
def init():
    line.set_data([],[])
    return line,
def animate(i):
    data = logisticmap(2.0+i*0.01)[50:]
    line.set_data(np.arange(len(data)), data)
    return line,
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=200, interval=100, blit=True)
plt.show()
Animating fractals

- In Lec 3, we plotted **Julia sets**: regions of the Argand diagram that converge under iteration of $z \to z^2 + c$.

- e.g. for $c = -0.123 + 0.745i$:

- How does the shape change as $c$ is varied?
- Make an animation …
Animating the van der Pol oscillator

\[ \ddot{x} - \mu(1 - x^2)\dot{x} + x = 0 \]

All curves tend towards a limit cycle
Animating the van der Pol oscillator

\[ \ddot{x} - \mu (1 - x^2) \dot{x} + x = 0 \]

\[ \dot{x} = y \]
\[ \dot{y} = \mu (1 - x^2) y - x \]

First, solve the ODEs to obtain data:

```python
mu = 1.0  # parameter value

def dx_dt(x, t):
    return [x[1], mu*(1-x[0]**2)*x[1] - x[0]]

def random_ic(scalefac=2.0):  # generate initial condition
    return scalefac*(2.0*np.random.rand(2) - 1.0)

ts = np.linspace(0.0, 40.0, 400)
nlines = 20
linedata = []
for ic in [random_ic() for i in range(nlines)]:
    linedata.append( odeint(dx_dt, ic, ts) )
```
Now create the animation:

```python
fig = plt.figure()
ax = plt.axes(xlim=(-3, 3), ylim=(-3, 3))
line, = ax.plot([], [], 'ro')
npts = len(linedata[0][0])
def init():
    line.set_data([],[])
    return line,
def animate(i):
    line.set_data([l[i,0] for l in linedata],
                   [l[i,1] for l in linedata])
    return line,
anim = animation.FuncAnimation(fig, animate, init_func=init,
                                frames=npts, interval=50, blit=True)
plt.show()
```
The Lorenz equations

In 1963, Edward Lorenz developed a simplified mathematical model for atmospheric convection. The model is a system of three ordinary differential equations now known as the Lorenz equations:

\[
\begin{align*}
\frac{dx}{dt} &= \sigma(y - x) \\
\frac{dy}{dt} &= x(\rho - z) - y \\
\frac{dz}{dt} &= xy - \beta z
\end{align*}
\]

- \( \sigma, \rho, \beta \) are parameters. Set e.g. \( \sigma = 10, \beta = 8/3, \rho = 28 \).
- **Non-linear** because of \( xy \) term
- Displays deterministic chaos
- 3 dimensional system with a **strange attractor**
Animating the Lorenz equations

- Code for animating 3D Lorenz equations on Jake Vanderplas’ Python blog:
  https://jakevdp.github.io/blog/2013/02/16/animating-the-lorentz-system-in-3d/
Saving your animations

- You may wish to save an animation to file, for use on the web.

- Common file formats include .gif and .mp4

- To save, you will first need to install a ‘writer’ such as ImageMagick (for gif) or ffmpeg (for mp4).

- These cannot be installed on the Managed Desktop (access rights).

- You can install these on your own laptop

- However, there is another option to try ...
Saving your animations

- **JupyterLab** is the next-generation web-based interface for Jupyter notebooks.

- To try it out, we can run a session of JupyterLab ‘in the cloud’ (remotely)

- Using Firefox, go to this blog post: https://blog.jupyter.org/jupyterlab-is-ready-for-users-5a6f039b8906

- Now click ‘try it with Binder’
Saving your animations

- Start a new notebook  File -> New -> Notebook

- Copy and paste the code from e.g. [http://sam-dolan.staff.shef.ac.uk/mas212/code/fractal_anim.py](http://sam-dolan.staff.shef.ac.uk/mas212/code/fractal_anim.py)

- Add one more line:

  ```python
  ani.save("fractal.gif", writer='imagemagick', fps=4)
  # fps = frames per second
  ```

- N.B. imagemagick is installed on the remote host.

- After some processing time, the file fractal.gif should appear in the left panel. You can then download it to your local folder.