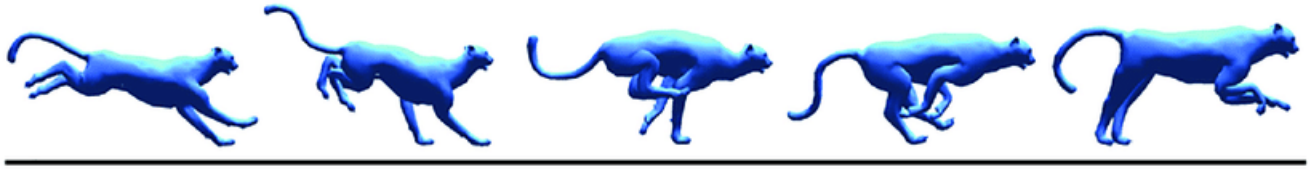
Animation
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm

https://people.sc.fsu.edu/~jburkardt/classes/python_2025/animation/animation.pdf

---



A sequence of still images results in an animation.

"Animation"

- *Animation creates the illusion of action by the display of a sequence of still images;*
- *The animation might be displayed directly to the screen;*
- *It might be saved to a movie file and display later;*
- *One approach uses Python to create the images, and then an external program to display them as a sequence or combine them into a movie.*
- *The more ambitious approach uses* `matplotlib`*'s* `FuncAnimation()` *to create, display, and combine the images in a single Python program.*

1 The idea behind animation

The human eye has a refresh rate of about $\frac{1}{30}$ of a second, so things that happen faster than that are not perceived. Toy makers in the 1800's created simple devices that flashed a series of still images quickly enough that the eye perceived continuous motion. Photography and video graphics continued this development, tricking our visual sense into perceiving moving objects.

A single graphic image. a map, a portrait, a histogram, by itself is a tremendous help to understanding. We can see things in an image that are not obvious in a table of numbers. By adding the illusion of motion, we gain an extra sense of how a given system behaves, changes or interacts.

Creating animations on the computer is often tedious work, but the results can be very rewarding. Even if you don't plan to make any animations yourself, it is a good idea to understand how they can be created, modified, and used.

For our discussion, we will go back to the Arenstorf orbit discussed previously, which I crudely animated by displaying images one at a time and hitting RETURN repeatedly. We want to see how to make such a sequence of images, and how to combine them into a more professional style animation.

2 Python animations with helper applications

It is possible to create an animation from scratch using a single Python program to create the data, arrange it into a series of plots, and combine the plots into a movie. This approach requires a lot of care and can easily break down for beginners.

We will present animation using a less ambitious approach. We will use a multi-step process, in which the later steps depend on accessing special helper programs.

1. The user uses `matplotlib` to creates a sequence of image files with numerically increasing names, such as *pic01.jpg, pic02.jpg, ..., pic99.jpg*:
2. The user invokes a special program, such as Blender, FFmpeg, ImageMagic convert(), mencoder, MovieMaker, Photoshop, or QuickTime that combines the image files into a single animation file.
3. The user invokes a movie player such as `mplayer`, `QuickTime`, or `vlc` to display the animation.

The great advantage of this approach is that the user can concentrate on designing the separate image frames using `matplotlib`. Once that is done, there are a number of choices for how to combine the images and display them. Those choices can be explored separately, and might involve installing new software. Moreover, various animation parameters such as the frames per second, image quality and so on, have to be experimented with. It's good to be able to worry about those things after you are sure you've got your image sequence prepared.

3 Basic animation using ImageMagick convert()

Consider the function $y(x,t) = \cos(x \cdot t)$. Plotted over a finite spatial interval, and starting from $t = 0$, the curve will gradually included more and more wiggles. Using the fixed spatial range $0 \leq x \leq 3$, we want to create 51 snapshots at equally spaced times in $0 \leq t \leq 5$. If we can create an animation from these snapshots, and the curve wiggles when we view it, we have some confidence that we can make an animation.

```
def cos_animation ( ):
  import matplotlib.pyplot as plt
  import numpy as np

  x = np.linspace ( 0.0, 10.0, 101 )

  for k in range ( 0, 51 ):

    t = k / 10.0
    y = np.cos ( x * t )
    plt.clf ( )
    plt.plot ( x, y, linewidth = 3 )
    plt.grid ( True )
    plt.xlabel ( "x" )
    plt.ylabel ( "y" )
    s = 'y=cos( x *' + str ( t ) + ' )'
    plt.title ( s )
    filename = 'frame_%05d.png' % ( k )
    plt.savefig (  filename )

  return
```

After we run this code, we should have a set of 51 snapshot files, named *frame_00000.png* through *frame_00050.png*. If we have ImageMagick available, we can convert these frames into a `gif` animation by a command like

```
convert -delay 10 -loop 1 *.png cos_animation.gif
```

It is even possible to embed this command inside the plotting program, by importing the `os()` library, which allows a Python program to execute external system commands.

You probably also want to delete all the `.png` snapshot files. You might be able to automate this as well, using the `os()` library to issue the appropriate command, which in a Unix system would be `rm *.png`.

4 Basic animation using matplotlib FuncAnimation

I hesitate to present animation using the `FuncAnimation()` routine because it is awkward, unfriendly, and hard to get working the first time. If you are interested in this approach, the best choice is to find a working example that is similar to what you want, and then try to understand it enough to modify it for your needs.

Having warned you, let's just say that the idea is that you define an initial image, and then construct a function typically called `update()` which modifies the previous image to create the next one. In order to do this, you need to save the previous image information in variables, and then modify it in ways that `matplotlib()` understands. Having said all this mysterious stuff, here is an example in which we plot the path of two balls thrown with different valocities, using a scatter plot for one, and a regular plot for the other:

```
def curve_animation ( ):

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

  fig, ax = plt.subplots()
  t = np.linspace ( 0.0, 3.0, 31 )

  g = -9.81
  v0 = 12.0
  z = g * t**2 / 2 + v0 * t

  v02 = 5.0
  z2 = g * t**2 / 2 + v02 * t

  scat = ax.scatter ( t[0], z[0], c = "b", s = 5, \
    label = 'v0 = ' + str ( v0 ) +'m/s' )
  line2 = ax.plot ( t[0], z2[0], \
    label = f'v0 = ' + str ( v02 ) + 'm/s')[0]

  ax.set ( xlim = [ 0.0, 3.0 ], \
           ylim = [ -4.0, 10.0 ], \
           xlabel = 'Time [s]', \
           ylabel = 'Z [m]')
  ax.legend ( )
  ax.grid ( True )

  def update ( frame ):

    x = t[:frame]
    y = z[:frame]
    y2 = z2[:frame]

    data = np.stack ( [ x, y ] ).T
    scat.set_offsets ( data )

    line2.set_xdata ( x )
    line2.set_ydata ( y2 )

    return ( scat, line2 )
```

```
ani = animation.FuncAnimation ( \
  fig = fig, \
  func = update, \
  frames = 31, \
  interval = 30, \
  repeat = False )

filename = 'curve_animation.mp4'
ani.save ( filename )

plt.show ( )
```

Notice that the `update()` function is inside the main function. Also notice that in this case, updating the plot is very simple because we simply show one more point of each of the precomputed curves. The peculiar way in which we signal to `scatter()` and `plot()` that their data has been updated is simply a mystery to me.

5 Converting an existing code to make images

We will suppose that we have used `solve_ivp()` to compute the solution `sol`, containing the sequence of times, positions, and velocities for the satellite that orbits the moon that orbits the earth. We have data at `n=71` equally spaced times, and want to create snapshots.

We create the images one at a time, using `matplotlib`. So we start a `for()` loop with index `i`. Our first plot is at time 0, and uses the 0-th set of y data. So we simply do the usual plotting actions, clearing the screen, drawing lines and markers, specifying grids and labels. When the plot is complete, we need to save it into its own file, with a unique numbered name, before we can move on to the next time step. Commonly used file format options include *jpg*, *png*.

Once the plotting loop is complete, we should have 71 files that form a sequence of still images of our solution. At that point, we can look around for a suitable problem to turn them into a movie.

But now we need to look more closely at some of the details of the filename construction and how we set the plot dimensions.

6 Framing your images

Usually, when you make a sequence of images, you don't want the coordinate system to change as you go along. But `matplotlib` doesn't see the whole movie, only a single image at a time. If decides how to "frame" the image, that is, the horizontal and vertical limits, based only on the current data. In order to get a stable coordinate system, you may need to override this default behavior.

When I made the simple display of the Arenstorf orbit, the first image included the earth at the origin, and the moon at (1,0), and nothing else. As the moon orbited the earth, `matplotlib` gradually realized that there was more to show, and so the coordinate system shifted in a surprising way. But we can insist at the beginning that all images share a common coordinate frame, by using the `xlim()` and `ylim()` commands, which have the form:

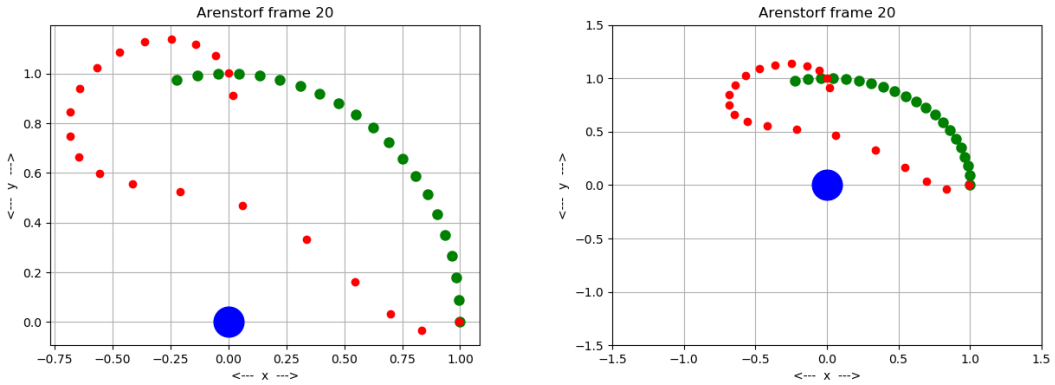
```
plt.xlim ( xmin, xmax )
plt.ylim ( ymin, ymax )
```

or you can use the command

```
plt.axis ( xmin, xmax, ymin, ymax )
```

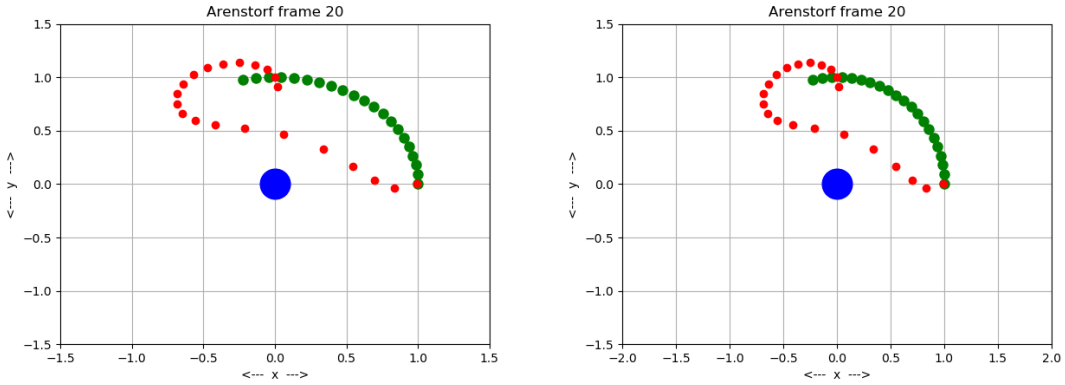
For the Arenstorf orbit, it might at first seem that -1 and +1 are the appropriate limits, but recall that the satellite goes outside the moon's orbit several times. We could guess initially that the limits might be -1.5 and +1.5, and after making the first set of plots, tighten these limits if it seems appropriate.

Here is the difference between the first frame of the Arenstorf orbit using default limits, and the limits we propose:



By forcing all the frames to share a common coordinate system, your animation won't seem to be changing its point of view as it proceeds.

Another issue you may run into is that `matplotlib` by default uses a display area that is 4 units wide and 3 units high. If you want the coordinate system to use the same measurements in both directions, then you have to adjust your `axis()` or `xlim()`, `ylim()` commands. For the Arenstorf orbit, we want the moon's orbit to appear circular. To do that, we should make the x axis run from -2 to 2, leaving the y axis at -1.5 to 1.5.



Note that, as far as I can tell, you can't use multiple `axis()` commands with `matplotlib`; it only carries out the final one. If you use `xlim()` and `ylim()`, then you shouldn't call `axis()` afterwards. If you call `axis(xmin,xmax,ymin,ymax)` then you can't afterwards also call `axis('equal'`, or else `matplotlib` will "forget" your previous command. This is a silly situation for which there must be a remedy, but I haven't found it yet.

7 How to Name a Lot of Files

To make a sequence of images, you presumably have a `for()` loop which increments a counter i as each frame is created. We will assume the frames are to be numbered from 0 through n, and that frame i is to be stored in a file with a common name including the frame index.

For each loop iteration i, we want to construct a correspondingly numbered filename, and then save the current image with that filename, using the command `plt.savefig(filename)`. If n is 101, we might create a file sequence

```
frame000.png
frame001.png
frame002.png
...
frame009.png
frame010.png
...
frame099.png
frame100.png
```

Note that the extra zeros in the initial set of frame names are important. If, instead, the file sequence began with

```
frame0.png
frame1.png
frame2.png
...
frame9.png
frame10.png
...
frame99.png
frame100.png
```

then when the images are combined into a movie, they will probably be combined in the following order:

```
frame0.png
frame1.png
frame10.png
frame100.png
frame2.png
...
frame9.png
frame99.png
```

To create names with the necessary number of leading zeros, we can use the `zfill()` function. In our example, the numeric field in the file names is 3 characters. So if our index i is too short, when printed as a character, we need to left-fill the string with enough zeros to use 3 positions.

Here is a demonstration of a few sample filenames if we require a 3-digit identification field:

```
for i in [ 0, 1, 2, 99, 999 ]:
  filename = 'frame' + str ( i ).zfill(3) + '.png'
  print ( filename )
  plt.savefig ( filename )
```

The resulting output is:

```
frame000.png
frame001.png
frame002.png
frame099.png
frame999.png
```

In other words, if the maximum frame index n uses k digits, we just have to use `zfill(k)` to make sure that every frame index is printed out with enough leading zeros so that the file names are in proper order. That way, the software that assembles the individual files into a movie will collect them in the proper order.

8 Converting the Arenstorf code

The code that I posted to the Canvas website for the Arenstorf orbit already had some graphics commands in it, so that I could make that initial cheap animation for you. Let's back up and consider what the code looked like originally, which was to solve the ODE's that defined the orbit, resulting in a sequence of values for `x,y,x',y'` over the time period $0 \leq t \leq 17$.

```
def arenstorf_solve_ivp ( ):

  from scipy.integrate import solve_ivp
  import matplotlib.pyplot as plt
  import numpy as np

  global mu1, mu2

  mu1 = 0.012277471
  mu2 = 1.0 - mu1
  tmin = 0.0
  tmax = 17.065216560157962558891720624 9
  n = 71

  tspan = np.array ( [ tmin, tmax ] )
  t = np.linspace ( tmin, tmax, n )
  y0 = np.array ( [ 0.994, 0.0, 0.0, -2.00158510637908252240537862224 ] )

  sol = solve_ivp ( arenstorf_dydt, tspan, y0, t_eval = t )
```

Although it was not computed as part of the ODE solution process, we can define the location of the moon as it goes around one orbit as well:

```
  theta = np.linspace ( 0.0, 2.0 * np.pi, n )
  moon_x = np.cos ( theta )
  moon_y = np.sin ( theta )
```

Now we are ready to consider the plot loop. Recall that the x and y coordinates of the satellite at the i-th time are stored in `sol.y[0,i]` and `sol.y[1,i]`. Since we want to indicate all the preceding locations as well, we will be plotting `sol.y[0,0:i+1]` and `sol.y[1,o:i+1]`:

```
  for i in range ( 0, n ):

    plt.clf ( )
    plt.figure ( figsize = ( 10.0, 7.5 ) )

    plt.xlim ( -2.0, 2.0 )
    plt.ylim ( -1.5, 1.5 )
#
#  Plot the earth as a blue circle.
#
    plt.plot ( 0.0, 0.0, 'bo', markersize = 25 )
#
```

```
#  Plot the moon as a green circle.
#
    plt.plot ( moon_x[0:i+1], moon_y[0:i+1], 'go', markersize = 15 )
#
#  Plot satellite as a red circle.
#
    plt.plot ( sol.y[0,0:i+1], sol.y[1,0:i+1], 'ro', markersize = 10 )

    plt.grid ( True )
    plt.title ( 'Arenstorf frame ' + str ( i ) )
    filename = 'arenstorf' + str ( i ).zfill(3) + '.png'
    plt.savefig ( filename )
    plt.close ( )
```

9 Creating a movie file

For a proper animation, we don't want to used the `plt.show()` command and have to hit return over and over. We want a movie file that we can play, or send to friends who don't have Python. To do that, we need to find a convenient application that accepts a sequence of still images in files, and creates a movie.

The program I found is `convert`, part of the freely available ImageMagick graphics library. Information and downloads are available at:

`https://imagemagick.org/index.php`

On my Linux system, my image files were called *arenstorf000.png* through *arenstorf070.png*. To create an `mp4` movie, I had to specify the delay between images, in hundreds of a second, and the desired image quality between 0 and 100. So my command was

```
convert -delay 10 -quality 05 arenstorf*.png arenstorf_animation.mp4
```

To create a `gif` version, a reasonable command would be

```
convert -delay 10 -loop 1 arenstorf*.png arenstorf_animation.gif
```