2 Introduction to Matplotlib: 3D Plotting and Animations

Lab Objective: *3D plots and animations are useful in visualizing solutions to ODEs and PDEs found in many dynamics and control problems. In this lab we explore the functionality contained in the 3D plotting and animation libraries in Matplotlib.*

Introduction

Matplotlib is a Python library that contains tools for creating plots in multiple dimensions. The library contains important classes that are needed to create plots. The most important objects to understand in this lab are figure objects, axes objects, and line objects. These three objects are created using the following code.

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()                  # Create figure object.
>>> ax = fig.add_subplot(111)           # Create axes object.
>>> line2d, = plt.plot([],[])           # Create empty 2D Line object
>>> line3d, = plt.plot([],[],[])        # Create empty 3D line object
```

Recall that `plt.figure()` creates a `matplotlib.figure.Figure` object, which is the window that is displayed when `plt.show()` is called. 3D plotting and animation both require explicitly defining the `Figure` object, as shown above. This allows for the object to be updated and modified, as will be explained later in the lab.

`Figure` objects contain `matplotlib.axes._subplots.AxesSubplot` objects, called *axes*. Axes are spaces to plot on, and are created by the `add_subplot()` method of a `Figure` object. Figures can have multiple axes.

Calling `plt.plot()` returns a list of line objects. For example, supposing `x1`, `y1`, `x2`, and `y2` are arrays containing data for two separate curves, then calling `plt.plot(x1, y1, x2, y2)` will return a list with two elements. Each element of the list is a `matplotlib.lines.Line2D` object. If the axes is three-dimensional, then the returned list will contain `matplotlib.lines.Line3D` objects. Because this function call returns a list, if only one line is plotted, adding a trailing comma to the variable name will assign the name to the first element of the returned list. You can alternatively reference the zero index of the returned list, but using a trailing comma is standard.

Animation Background

The animation library in Matplotlib contains a class called `FuncAnimation`. We will use this class throughout this lab. `FuncAnimation` requires a user-defined *update* function that controls the plot for each frame of the animation. This grants the user wide flexibility and control of the resulting animation. The following steps describe the process of creating a simple animated plot using the `FuncAnimation` class.

1. Compute all data to be plotted.
2. Explicitly define figure object.
3. Define line objects to be altered dynamically.
4. Create function to update line objects.
5. Create `FuncAnimation` object.
6. Display using `plt.show()`.

These steps will be explained by way of an example. The arrays `x` and `y` contain data giving the location of a particle moving in the plane. To visualize this motion, one could animate the particle as well as display the trajectory that the particle has traveled. For this animation, two separate `Line2D` objects must be created on an axes object. The first, `particle` will be for the position of the particle itself, and the second, `traj` will be for the trajectory that the particle has traveled. Note that these objects are created with empty lists of data. The update function will be used to dynamically set the data to be plotted in these line objects.

```
>>> import matplotlib.animation as animation
>>> import numpy as np
>>> t = np.linspace(0,2*np.pi,100)
>>> x = np.sin(t)
>>> y = t**2
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.set_xlim((-1.1,1.1))
>>> ax.set_ylim((0,40))
>>> particle, = plt.plot([],[], marker='o', color='r')
>>> traj, = plt.plot([],[], color='r', alpha=0.5)
```

The update function must be defined a specific way in order to interact properly with the `matplotlib.animation.FuncAnimation` object. The update function must accept the current frame index as its first input parameter and it must return a list or tuple of line objects. The current frame index is used to access the data to be plotted in the current frame. Both 2D and 3D line objects have the built-in method `.set_data()`. This function takes in two one-dimensional arrays representing x and y values to plot. This allows a single line object to display different data for each frame. Inside the update function, `.set_data()` is called on the line objects with the relevant data as inputs.

```
>>> def update(i):
>>>     particle.set_data(x[i],y[i])
>>>     traj.set_data(x[:i+1],y[:i+1])
>>>     return particle,traj
```

Next, the `FuncAnimation` object is created. The argument `frames` specifies the iterable representing the frame indices. If `frames` is an integer, it is treated as the iterable `range(frames)`. After the `FuncAnimation` object is created, `plt.show()` displays the animation.

```
>>> ani = FuncAnimation(fig, update, frames=range(100), interval=25)
>>> plt.show()
```

The following table shows more parameters that can be passed into FuncAnimation.

Parameter	Description
`fargs` (tuple)	Additional arguements to pass update function
`interval` (float)	Delay between frames in milliseconds
`repeat` (bool)	Determines whether animation repeats (Default True)
`blit` (bool)	Determines whether blitting is used (Default False)

> **Note**
>
> When using FuncAnimation, it is essential that a reference is kept to the instance of the class. The animation is advanced by a timer and if a reference is not held for the object, `Python` will automatically garbage collect and the animation will stop.

Embedding Animations

Some systems may struggle with using `plt.show()` to display an animation. In this case, it may be easier to embed the animation using the HTML5 API. Jupyter notebooks use HTML to display their contents, so we can leverage this and use HTML5's video capabilities to insert video directly into a notebook.

To embed the video directly into a notebook using HTML5 you must use the `IPython.display` module. This module will be able to interpret an encoded HTML5 video, which `matplotlib.animation` can create. This method tends to be much more simple than rendering the animation to an `.mp4` file and then embedding that file into a notebook, as it does not require an outside encoder, and it tends to be encounter fewer bugs than using `plt.show()`. Here is a snippet you may reference to embed an animation using `IPyton.display`

```
# required import statements
from IPython.display import HTML
import matplotlib.pyplot as plt
from matplotlib import animation

# disable interactive mode
plt.ioff()
'''
Here we would insert whatever code needed to create the animation
such as instantiating the fig object and defining the update function
'''
# create animation
ani = animation.FuncAnimation(fig, update, frames=N, interval=i)
```

```
# render as html5 and embed
HTML(ani.to_html5_video())
```

Problem 1. Use the FuncAnimation class to animate the function $y = \sin(x + 0.1t)$ where $x \in [0, 2\pi]$, and t ranges from 0 to 100 seconds.

3D Plotting Introduction

3D plotting is very similar to 2D plotting. The main difference is that a set of 3D axes must be created within the figure object. A 3D axes object is created using the additional keyword argument `projection='3d'`, as shown below. Note that the Axes3D submodule must first be imported in order to create the 3D axis.

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>>
>>> # Create figure object.
>>> fig = plt.figure()
>>>
>>> # Create 3D axis object using add_subplot().
>>> ax = fig.add_subplot(111, projection='3d')
```

3D Static Plotting

When the axes object is explicitly defined, plots are generated by calling the chosen plot function (such as `ax.plot()` on the axes object. Additional information on the use of axes objects can be found here: https://matplotlib.org/api/axes_api.html.

Problem 2. The orbits for Mercury, Venus, Earth, and Mars are stored in the file `orbits.npz`. The file contains four NumPy arrays: `mercury`, `venus`, `earth`, and `mars`. The first column of each array contains the x-coordinates, the second column contains the y-coordinates, and the third column contians the z-coordinates of each planet, all relative to the Sun, and expressed in AU (astronomical units, the average distance between Earth and the Sun, approximately 150 million kilometers).

Use `np.load('orbits.npz')` to load the data for the four planets' orbits. Create a 3D plot of the orbits, and compare your results with Figure 2.1.

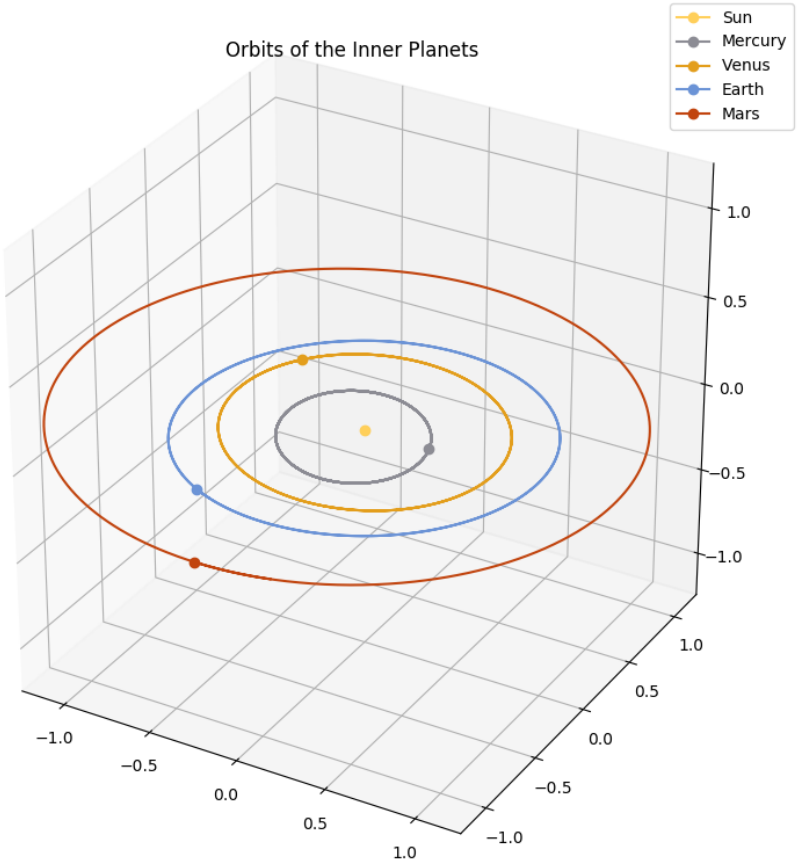

Figure 2.1: The solution to Problem 2.

3D Animations

The key difference between 2D and 3D animations is that the `.set_data()` method does not support setting the `z` values. Instead, set the `x` and `y` values with `.set_data()` as before, and then set the `z` values with `.set_3d_properties()`. The `.set_3d_properties()` function call is also made inside the update function.

Animation in 3D requires more careful consideration than in the 2D case. When `matplotlib` displays a 3D plot, it does so in an interactive figure that allows the user to change the camera angle and position. Since 3D rendering is more computationally expensive than 2D rendering, interactive views of 3D animations often have poor framerates and choppy rendering. The solution is to use the tools in the `matplotlib.animation` module to render the animation and then embed that rendered animation within a Jupyter Notebook. This can be done in two ways: embedding the video directly using an HTML5 API, or encoding the animation to an `.mp4` video and embedding that video. Embedding the video directly tends to be easier, as `matplotlib` does not come with an `.mp4` video encoder, while it can encode video for HTML5.

Saving Animations

Saving the animation by encoding it to a `.mp4` file will allow you to display the video inline inside a Jupyter Notebook, or view it using any video player supporting the chosen filetype.

Unfortunately, Matplotlib does not come with a built-in video encoder. The `matplotlib.animation` module supports several third-party encoders. FFmpeg is a lightweight solution which can be obtained from: https://www.ffmpeg.org/download.html.

To prevent the animation from displaying while it is being rendered as video, use `plt.ioff()`. This turns off matplotlib's interactive mode until `plt.ion()` is called. After creating the animation object, use its `.save()` method with the desired filename to render and save the video. The following code is given for reference:

```
animation.writer = animation.writers['ffmpeg']
plt.ioff()    # Turn off interactive mode to hide rendering animations

# Code to create figure, axes, update function
ani = animation.FuncAnimation(fig,update,frames,interval)
ani.save('my_animation.mp4')
```

To display the `.mp4` video in a Jupyter Notebook, run the following code in a separate markdown cell:

```
<video src="planet_ani.mp4" controls>
```

Problem 3. Each row of the arrays in `orbits.npz` gives the position of the planets at evenly spaced time points. The arrays correspond to 1400 points in time over a 700 day period (beginning on 2018-5-30). Create a 3D animation of the planet orbits. Display lines for the trajectories of the orbits and points for the current positions of the planets at each point in time. Your `update()` function will need to return a list of Line3D objects, one for each orbit trajectory and one for each planet position marker. Embed your animated plot.

Surface Plotting

3D surface plotting is very similar to regular 3D plotting discussed earlier. The difference with surface plots is that they require first creating a *meshgrid* for X and Y. Meshgrids are created using the NumPy command `np.meshgrid(x, y)` where `x` and `y` are 1D arrays representing the x and y coordinates of the grid. This function creates 2D arrays `X` and `Y` that combined give cartesian cordinates for every point made from the `x` and `y` arrays.

Once a meshgrid is defined, a surface plot is generated by calling `ax.plot_surface(X, Y, Z)`, where `Z` is a 2D array of height values that is the same shape as `X` and `Y`.

Problem 4. Make a surface plot of the bivariate normal density function given by:

$$f(\mathbf{x}) = \frac{1}{\sqrt{det(2\pi\Sigma)}} \exp\left[-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right]$$

where $\mathbf{x} = [x, y]^T$, $\mu = [0, 0]^T$ is the mean vector, and

$$\Sigma = \begin{bmatrix} 1 & 3/5 \\ 3/5 & 2 \end{bmatrix}$$

is the covariance matrix. Compare your results with Figure 2.2.

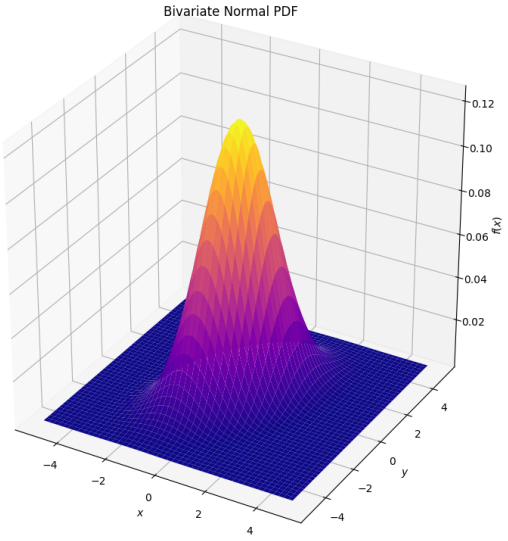

Figure 2.2: The solution to Problem 4.

Surface Animations

Animating a 3D surface is slightly different from animating a parametric curve in 3D. The object created by `.plot_surface()` does not have a `.set_data()` method. Instead, use `ax.clear()` to empty the axes at each frame, followed by a new call to `ax.plot_surface()`. Note that the axes limits must be reset after `ax.clear()` is called.

Problem 5. Use the data in `vibration.npz` to produce a surface animation of the solution to the wave equation for an elastic rectangular membrane. The file contains three NumPy arrays: `X`, `Y`, `Z`. `X` and `Y` are meshgrids of shape `(300,200)` corresponding to 300 points in the y-direction and 200 points in the x-direction, giving a 2x3 rectangle with one corner at the origin. `Z` is of shape `(150,300,200)`, giving the height of the vibrating membrane at each (x,y) point for 150 values of time. In the language of partial differential equations, this is the solution to the following initial/boundary value problem:

$$u_{tt} = 6^2(u_{xx} + u_{yy})$$
$$(x, y) \in [0, 2] \times [0, 3], t \in [0, 5]$$
$$u(t, 0, y) = u(t, 2, y) = u(t, x, 0) = u(t, x, 3) = 0$$
$$u(0, x, y) = xy(2 - x)(3 - y)$$