Programming for Medical Physics

Neil Kirby, Ph.D.

Assistant Professor

The University of Texas Health Science Center San Antonio

Adapted for Python by Noah Bice

Lecture 4: Data Visualization

4.1 Importance of Data Visualization.
4.2 Output Factors.
4.3 Plotting Data.
4.4 Image Visualization.

4.1 Importance of Data Visualization

Data visualization is a crucial tool for scientists, as converting numeric data into visual plots is usually the quickest way to understand and internalize the information. For this reason, it is important to be proficient at various plotting tools in Python. Data processing usually involves several data conversion steps. My only reliable way to make sure this has been done correctly is to visualize data at each step. Beyond this, one of our other goals as scientists is to publish our findings. Python is a powerful tool for creating beautiful figures for publication.

4.2 Output Factors

Figure 1 displays a diagram of a collimator head for a linear accelerator, which converts MeV-scale energy electrons to MeV-scale energy photons. These electrons strike a target in this head and the resulting deceleration creates bremsstrahlung. The produced photons then diverge from point (as point source) and are shaped by the primary collimator. This collimator is a static component in this head. The bremsstrahlung photons are forward peaked, so a flattening filter is next utilized to simply flatten the photon fluence. After the flattening filter, the photons pass through the monitor chamber, which is simply an ion chamber. The charge collected from this chamber serves as the internal dose units for the linear accelerator, referred to as monitor units (MUs). After passing through the monitor chamber, the photons pass through the secondary collimator, which can be moved to change the width of the radiation fluence impinging upon a patient.

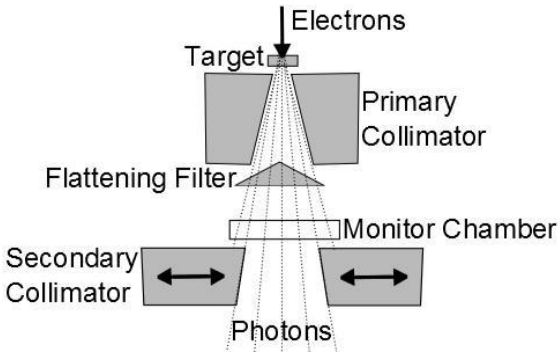

FIG. 1. The components of a collimator head for a linear accelerator.

As an example, when a 6 MV photon beam (photons from 6 MeV electron-target collision) is properly calibrated at our institution, 1 MU delivers the dose of 1 cGy at a depth of 1.5 cm along the central radiation axis, for a source to surface distance of 100 cm and a field size of 10 cm (created by the secondary collimator). These are referred to as reference conditions. For a fixed number of delivered MUs, the field size of the secondary collimators changes the amount of dose delivered at the reference measurement point. An output factor is the ratio of this dose to that at reference conditions.

For every installed treatment machine, these output factors must be measured and included in the physics model for the machine. This is a critical point for modeling any machine, as an error in these measurements would cause a dose delivery error equal to the error of the determined output factor. The film measurements from the last homework assignment were output factor measurements.

4.3 Plotting Data

There are plenty of libraries for visualizing data with Python, but MatPlotLib is by far the most popular. MatPlotLib has a module called PyPlot which wraps MatPlotLib's lower-level objects for convenient use. When I want to plot data, I usually begin with an import statement of matplotlib.pyplot (and NumPy). In this section, I'll plot some results from the previous homework assignment. I start by importing the necessary libraries and manually entering data for each of the students.

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

>>> collsize = np.array([5, 7.5, 10, 12.5, 15, 20, 25, 30, 35, 40, 50, 60])
>>> student_data = np.zeros((4,12))
>>> student_data[0] = np.array([2.4, 3.86, 4.54, 4.74, 4.606, 5.88, 5.684, 6.566, 6.034, 6.16, 6.286,
6.174])/6.174
>>> student_data[1] = np.array([3.6058, 4.6313, 5.2841, 5.5168, 5.6156, 6.1053, 6.0814, 6.2311, 6.0951,
6.1324, 6.1986, 6.3929])/6.3929
```

```
>>> student_data[2] = np.array([3.7536, 4.6017, 5.2172, 5.4776, 5.5632, 5.9846, 5.9947, 6.0347, 6.0084, 6.0211, 6.0371, 6.2556])/6.2556
>>> student_data[3] = np.array([3.6667, 4.3279, 5.27, 5.4494, 5.4651, 6.0738, 6.1444, 6.5299, 6.3340, 6.1915, 6.2716, 6.633])/6.633
```

We now have a list of 12 collimator sizes and a 4x12 NumPy array with the students' results. We'll come back to these data in a bit.

. . .

The function for plotting lines is called plt.plot(). Like many other functions, this function can operate with a various number of inputs, but at least one is required. The simplest use of plt.plot() is to give this function a list of values (see Fig. 2). plt.plot() requires that the x (and y) data that you feed it are "array-like"; this just means that you should be able to call np.array() on them without throwing an error.

```
>>> plt.plot([1, 2, 3, 2, 1])
>>> plt.show()
```

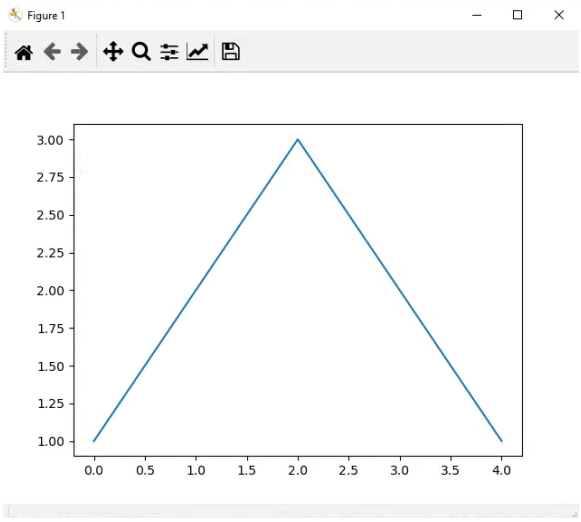

FIG. 2. Using PyPlot's plot() function.

This plots the values of this array along the y direction and then assumes x values that run from 0 to the number of elements (0:4 in this case). Alternatively, you can plot this array of values with corresponding x values. You can also use the plt.scatter() function to plot scatter data.

```
>>> xvals = np.linspace(0, 2*np.pi)
>>> yvals = np.cos(xvals)
>>> plt.plot(xvals, yvals)
>>> plt.show()
```

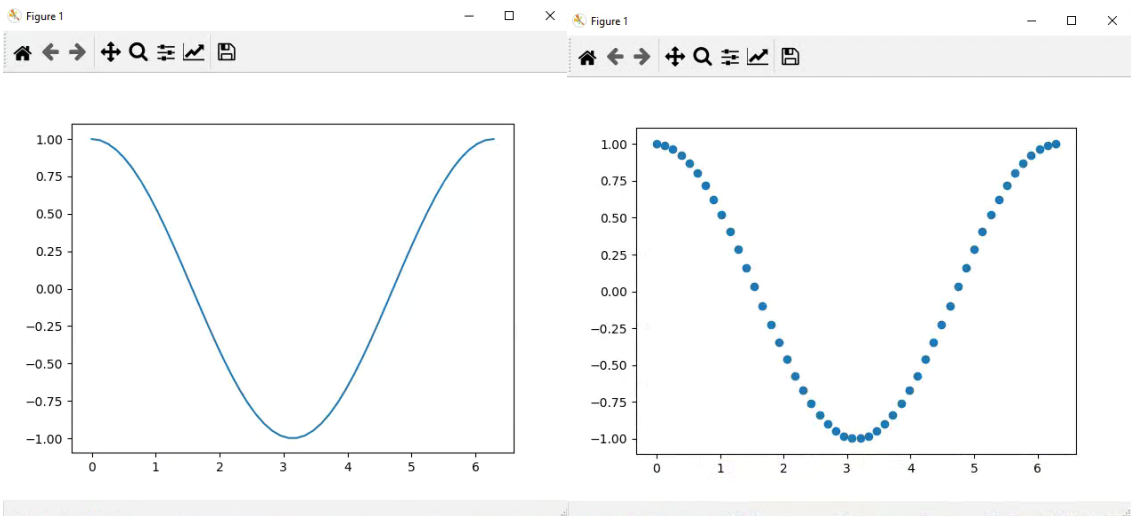

FIG. 3. Plotting cosine on the interval [0, 2*pi] with plt.plot() and plt.scatter().

These are the basics for using the function, which by default generates blue lines. If you create more objects on the same plot, however, PyPlot will automatically use a different color to avoid confusion.

```
>>> plt.plot(xvals, np.cos(xvals))
>>> plt.plot(xvals, np.sin(xvals))
>>> plt.show()
```

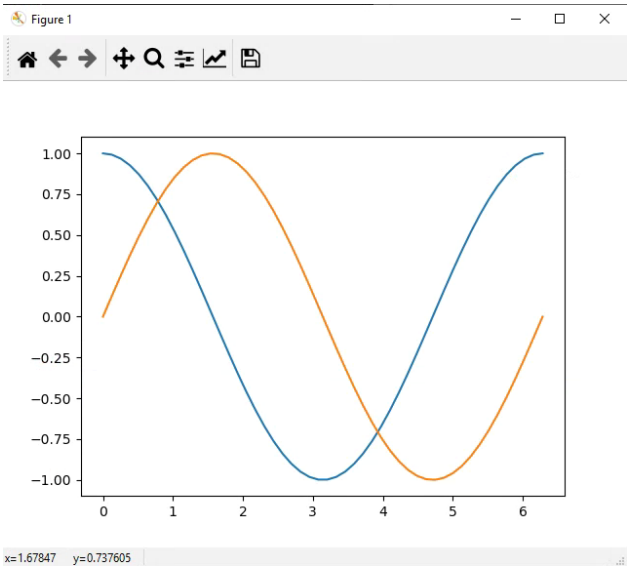

FIG 4. Drawing multiple lines on the same plot will automatically change their color.

We can specify the way our lines want to look by adding several keyword arguments. You can give a two character string as a third argument, where the first represents color and the second represents the line or marker style. For example, plt.plot(xvals, yvals, 'r--') plots a red dotted line. The letters r, g, b, c, m, y, k, and w represent, red, green, blue, cyan, magenta, yellow, black, and white, respectively. The line style specifiers of -, --, :, -., represent a solid line, a dashed line, a dotted line, and a dash-dot line,

respectively. The various marker types are +, o, *, ., x, square, diamond, ^, v, >, <, pentagram, and hexagram.

. . .

We can now continue the walkthrough of plotting student data.

```
>>> plt.plot(collsize, student_data[0], 'b^', markersize=15, label='student 1')
>>> plt.plot(collsize,student_data[1], 'r--', linewidth=2, label='student 2')
>>> plt.plot(collsize,student_data[2], 'o', markersize=10, color=[0,1,1], markerfacecolor=[1,0,0], label='student 3')
>>> plt.plot(collsize,student_data[3], '-' ,color=[0.5,0.5,0.5], linewidth=2, marker='s', markersize=17, label='student 4')
```

With the above code I've created four four lines on the same axes. I have specified the marker and line types with strings such as 'b^'. plt.plot() also takes several keyword arguments, where you can specify the style of your plots, like markersize=15. I've included a list of some of plt.plot()'s keyword arguments below, from the PyPlot documentation.

label	object
linestyle or ls	{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
linewidth or lw	float
marker	marker style
markeredgecolor or mec	color
markeredgewidth or mew	float
markerfacecolor or mfc	color
markerfacecoloralt or mfcalt	color
markersize or ms	float
markevery	None or int or (int, int) or slice or List[int] or float or (float, float)
path_effects	AbstractPathEffect
picker	float or callable[[Artist, Event], Tuple[bool, dict]]
pickradius	float
rasterized	bool or None
sketch_params	(scale: float, length: float, randomness: float)
snap	bool or None
solid_capstyle	{'butt', 'round', 'projecting'}
solid_joinstyle	{'miter', 'round', 'bevel'}
transform	matplotlib.transforms.Transform

FIG 5. A list of some keyword argmuents to plt.plot(). The full documenation can be found on the PyPlot website. PyPlot is one of the most commonly used packages, and it is very well documented and explained in the tutorials.

Notice I've included label='Student X' each time I created a line. This makes it easy to add a legend to the plot. I can simply call plt.legend(), and the lines will be displayed in the legend according to their label.

```
>>> plt.legend()
```

I can re-label my figures axes and title by simply calling plt.xlabel(), plt.ylabel(), and plt.title(). These functions also use keyword arguments to specify style. To use a LaTex interpreter for mathematical expressions, you can enclose the string in dollar signs ($). If you want to use a backslash to declare a special character like a letter from the Greek alphabet, remember to use two backslashes as opposed to the usual single backslash, e.g. $\\alpha$.

```
>>> plt.xlabel('Collimator Size[mm]')
>>> plt.ylabel('Output Factor', fontsize=16)
>>> plt.title('This is a LaTex example: $\\alpha$', fontsize=18)
```

Finally, we can add text at some location in the plot and add grid lines. To add text, you can call plt.text() with the x and y positions of the beginning of your string. To add grid lines, simple call plt.grid().

```
>>> plt.text(20,0.6,'This is how you enter text', fontname='fantasy', fontsize=15)
>>> plt.grid()
```

We can save the figure with plt.savefig() or we can show it with plt.show().

```
>>> plt.savefig('./my_figure.png')
>>> plt.show()
```

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

collsize = [5,7.5,10,12.5,15,20,25,30,35,40,50,60]
student_data = np.load('pyth_lect4.npy')

plt.plot(collsize, student_data[0], 'b^', markersize=15, label='student 1')
plt.plot(collsize,student_data[1],'r--',linewidth=2, label='student 2')
plt.plot(collsize,student_data[2],'o',markersize=10,color=[0,1,1],
    markerfacecolor=[1,0,0], label='student 3')
plt.plot(collsize,student_data[3],'-',color=[0.5,0.5,0.5],
    linewidth=2,marker='s',markersize=17, label='student 4')
plt.legend()
plt.xlabel('Collimator Size[mm]')
plt.ylabel('Output Factor', fontsize=16)
plt.title('This is a LaTex example: $\\alpha$', fontsize=18)
plt.text(20,0.6,'This is how you enter text', fontname='fantasy', fontsize=15)
plt.grid()
plt.savefig('my_figure.png')
plt.show()
```

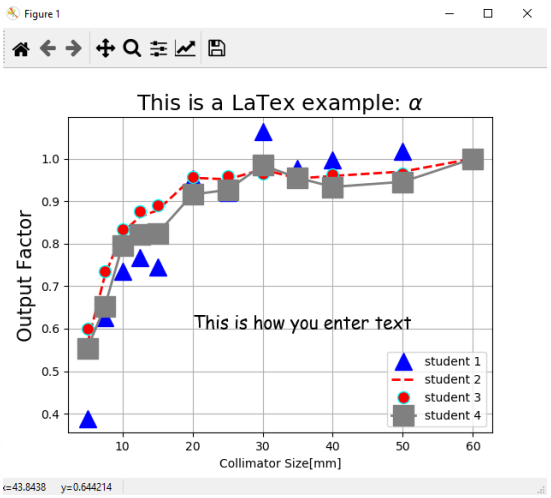

FIG. 6. The final sum of code and the figure it creates.

4.4 Figures and Subplots

You will find in many Python codes that the user will declare an instance of a plot with plt.subplots(). This function outputs a figure and a list of axes. The axes objects are what you'd conventionally think of as plots and the figure object keeps track of all the individual axes. Creating figures in this way can be useful for keeping track of several plots, creating animations, and slicing 3D CT data. The syntax for creating subplots is

```
>>> fig, axs = plt.subplots(2, 2)
```

Here I've create a figure called "fig" and axes called "axs" in a 2x2 grid. There are 4 individual subplots in the figure, which I access by their indices. I can plot lines on the individual subplots very much like plotting on a single figure.

```
>>> axs[0,0].plot(collsize, student_data[0])
>>> axs[0,0].set_title('Student 1')
>>> axs[0,1].plot(collsize, student_data[1])
>>> axs[0,1].set_title('Student 2')
>>> axs[1,0].plot(collsize, student_data[2])
>>> axs[1,0].set_title('Student 3')
>>> axs[1,1].plot(collsize, student_data[3])
>>> axs[1,1].set_title('Student 4')
>>> plt.show()
```

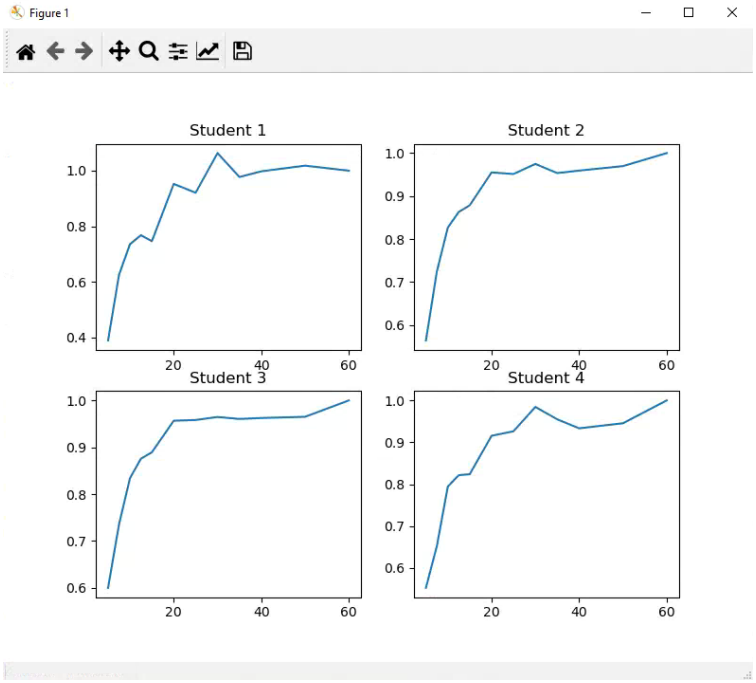

FIG 7. Creating a 2x2 grid of subplots.

Notice that the syntax for changing the properties of axes is slightly different from just creating a single figure. Instead of using plt.title(), plt.xlabel(), we use ax.set_title(), ax.set_xlabel(), etc. Being able to access axes by their index is convenient for creating figures in loops. See the following example.

```
>>> f, a = plt.subplots(4)
>>> for i in range(4):
...         a[i].plot(collsize, student_data[i])
```

```
...         a[i].set_xlabel('coll size')
...         a[i].set_ylabel('output ratio')
...         a[i].set_title('Student {}'.format(i+1))
>>> plt.show()
```

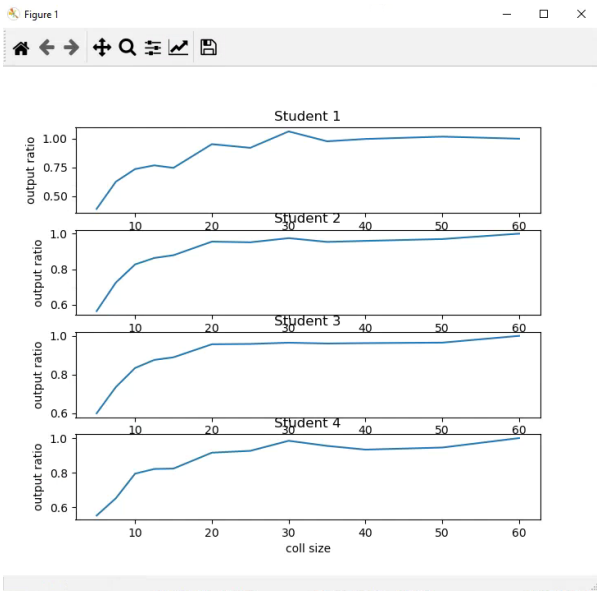

FIG 8. Output from creating subplots with a loop. Notice that the x-axis labels for students 1-3 are covered by the titles for students 2-4.

In the figure we've created, the titles of some subplots overlap with x-axis labels on other plots. Hiccups like this sometimes happen, and they can be addressed in a couple ways. We could go back to our code and change the margins between subplots with f.tight_layout(pad=XXX), or we can use the convenient interface provided by PyPlot. At the top of the figure window, there are a few tools to help us edit our plots without needing to know every possible keyword argument for every function.

FIG 9. The zoom, edit subplots, and edit axes tools in the PyPlot figure toolbar.

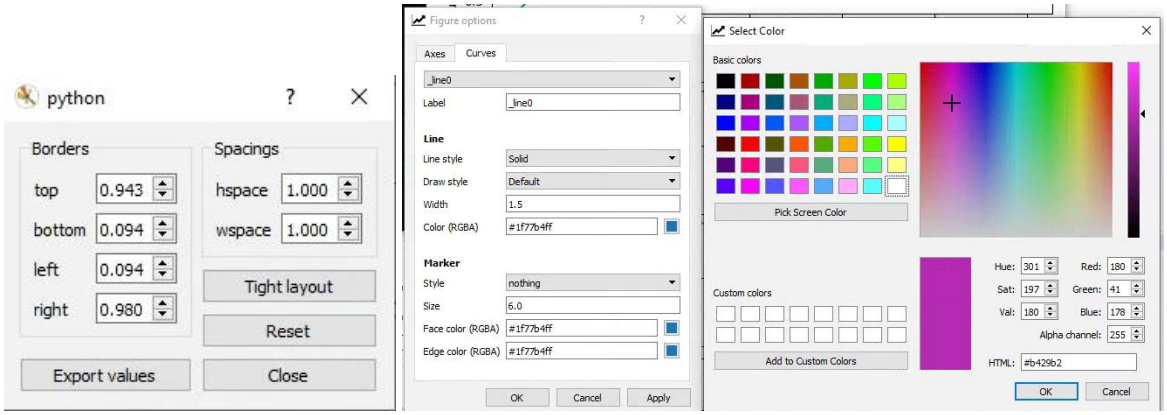

FIG 10. You can use the PyPlot tools to conveniently adjust the appearance of a plot.

We can fix the margins of Figure 8 by changing hspace and wspace in the edit subplots interface. I've also changed the line color for Student 2 to a unique shade of magenta. We can also save the figure from this window after we've applied our edits.

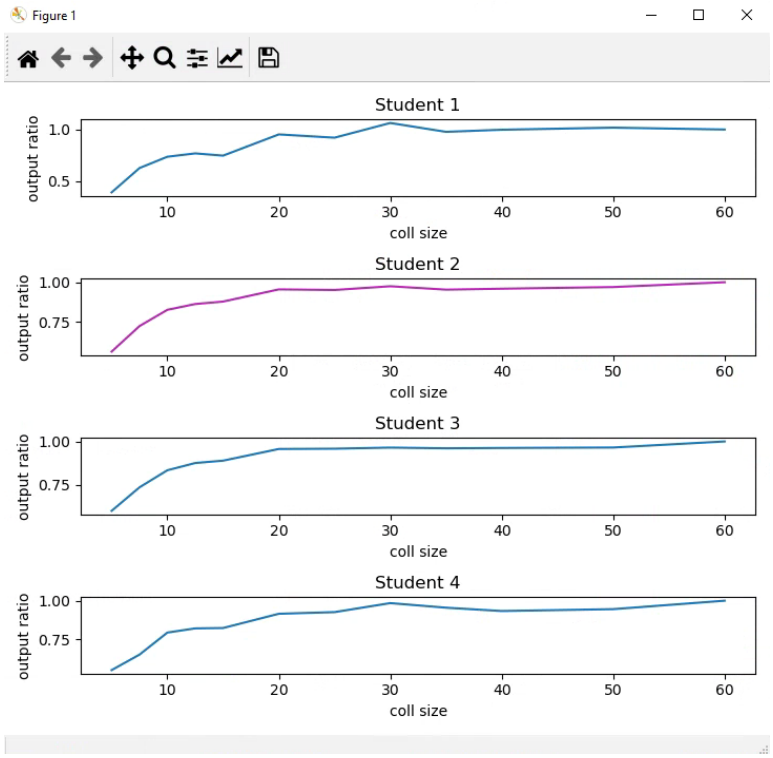

FIG 11. Figure 8 with corrected margins.

4.5 CT Visualization

One of the most useful scripts I've written is for visualizing CT datasets. In the script, I create an interactive figure with MatPlotLib that allows the user to parse through and visualize slices in a CT image set. The script is based around a method of PyPlot figures called plt.figure().canvas.mpl_connect(). This

function allows us to execute code when we receive an input from the user like “button_press_event”, “key_press_event”, or in our case, “scroll_event”. The way we use the function is something like

1. Create a figure and axes
2. Write a function to update the axes following an event
3. Call figure.canvas.mpl_connect(‘event’, function)
4. plt.show()

In the following script, these 4 steps are illustrated in the function show_CT(). I’ve implemented step 2 by creating a class called Tracker. The Tracker class takes the axes of our figure and the CT data and manages them. Objects of this class will store the CT data and update the axes with the method Tracker.scroll(). See the following figure.

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

class Tracker(object):
    def __init__(self, ax, CT):
        self.ax = ax
        ax.set_title('CT Visualization')
        self.CT = CT
        self.slices = CT.shape[0]
        self.ind = self.slices//2
        self.im = ax.imshow(self.CT[self.ind, :, :], cmap='gray')
        self.update() #upon instatiation, draw the image at self.ind on the axes' canvas

    def scroll(self, event):
        if event.button == 'up':
            self.ind = (self.ind + 1) % self.slices #modulus, so when we reach the end of the patient, we start over
        else:
            self.ind = (self.ind - 1) % self.slices
        self.update()

    def update(self):
        self.im.set_data(self.CT[self.ind, :, :]) #update image data
        self.ax.set_ylabel('Slice {}'.format(self.ind)) #update slice label
        self.im.axes.figure.canvas.draw() #draw new data on axes

def show_CT(ct_file):
    ct = np.load(ct_file)
    fig, axs = plt.subplots()
    tracker = Tracker(axs, ct)
    fig.canvas.mpl_connect('scroll_event', tracker.scroll)
    plt.show()

if __name__ == '__main__':
    show_CT('ct.npy')
```

FIG 12. The function show_CT() uses a Tracker to display CT slices when the user scrolls. Notice I’ve used “if __name__ == ‘__main__’”, to keep the code at the bottom from being executed if I want to import show_CT() in another script.

See how Tracker.update() is used to draw an image to the figure’s axes, and Tracker.scroll() uses Tracker.update(). I have a file saved in this directory called ‘ct.npy’, which I can use for testing my code. When I run this script, a PyPlot figure opens at the center CT slice, and I can navigate from slice to slice with my mouse’s scroll wheel.

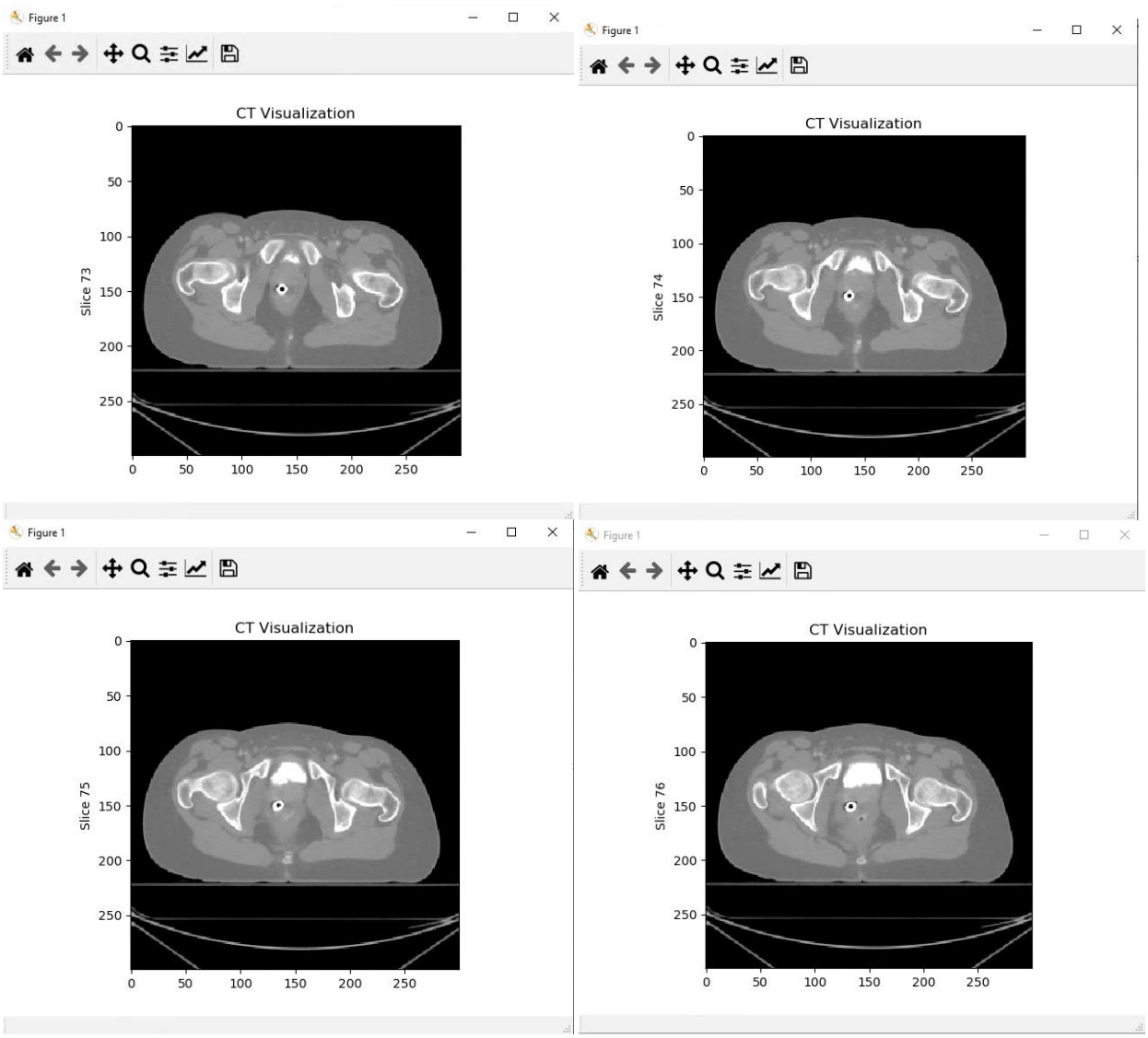

FIG 13. Some slices of ‘ct.npy’ visualized with the tool in Figure 12.