
Math 4610 Fundamentals of Computational Mathematics - Topic 10.

It is very important as a computational mathematician that we use computational resources to visualize data or output from simulations. This means producing graphics codes that can display output from data sets in real world problems or from computer codes that simulate real world problems. We could go back to basic principles and write software from scratch, say using Open Graphics Library (OpenGL) or some other set of libraries. This is time consuming and is basically a digital exercise in reinventing the wheel.

At least initially, we should take advantage of the many software packages and codes that produce graphics. The idea in this section of the course is to come up with a way to display 2-d graphical output. For example, we may want to graph a set of ordered pairs that represent data or output of a simulation. We will have plenty of examples and problems in this course that will be aided by having a code that will do some simple graphs of functions and/or data.

Note that there are a lot of other graphical representations of data that we could consider. For example, surfaces, barcharts, contours and level sets, and beyond. In this first shot at displaying graphics, we will stick to the simplest ideas and produce a Python module/script that will produce 2D graphics. The code will become useful in all kinds of applications, from plotting the error in approximations to solutions of differential equations. We will reuse the code in some form or another throughout the semester.

Visualizing Results: 2D Plotting

We can start in several places, but the ideas are all centered around plotting points in the plane and connecting the plotted points with curves of some kind to display the behavior of an underlying function. It should be noted that no matter how accurate the graph is, there will always be some error due to screen resolution and finite precision in the representation of numbers. So, suppose that we have generated a list of ordered pairs and want to plot these values.

As an example, the content in Topic 09 includes a couple of 2-d plots displaying the behavior of the error in approximating a derivative with a difference quotient. In that example, a source of some bad behavior in the derivative approximation was instantly manifested in the graph.

In a general setting, suppose that data in the form of a set of ordered pairs is given. That is,

$$f = \{(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$$

This is a set of $n + 1$ data points. There is a reason for starting at zero with our indices. That will be described when we fit a polynomial through given data points - more on this later in the semester.

Another thing to note is that we could also take a function, say

$$P(t) = \frac{400 e^{0.2t}}{1 + 3 e^{0.2t}}$$

and sample the function at a set of input values, t_i , for $i = 0, 1, \dots, n$. The function under consideration is a solution of the logistic differential equation for population growth. Using the formula above, we can generate a set of ordered pairs by choosing values of the input (time) and computing values of $P(t)$ (population density). That is,

$$P = \{(t_0, P(t_0)), (t_1, P(t_1)), (t_2, P(t_2)), \dots, (t_n, P(t_n))\}$$

Now, let's figure out a way to get a simple graph of this function. A 2-d plot of the population density above is given in the following graph.

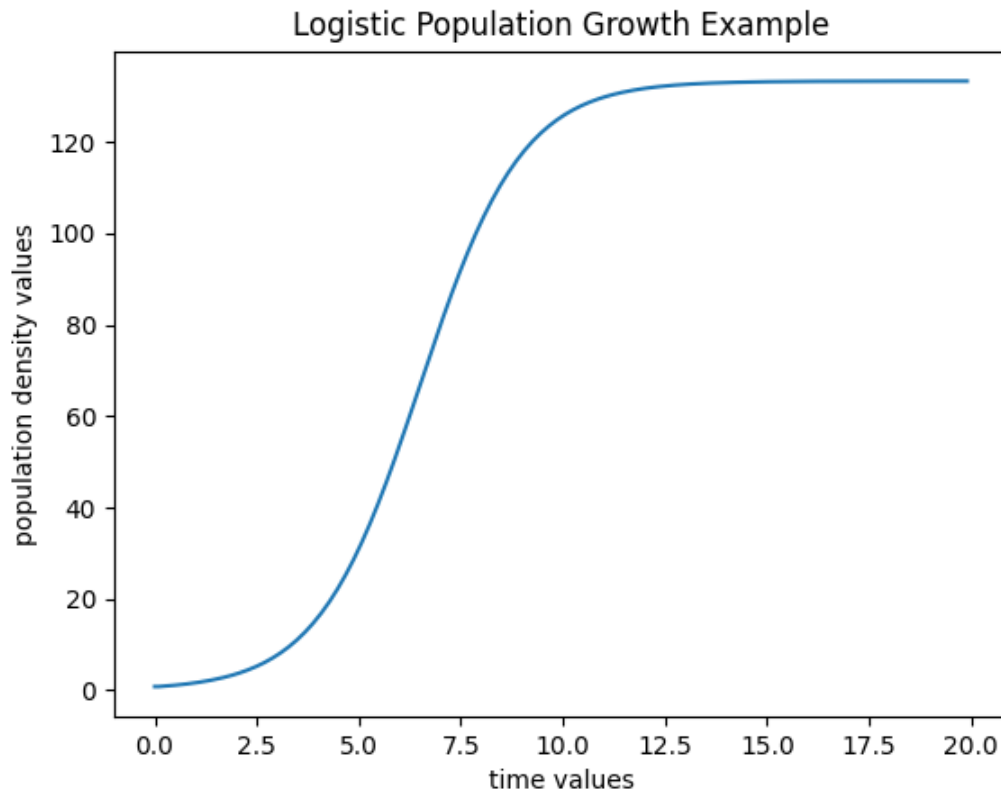


Figure 1: A 2-d plot of a population density function. This is just a simple example of a plot that shows the results of sampling a function.

Visualizing Results: 2D Plotting

To create the graph shown above, a module/script was written in Python. In addition, the development was done in an Integrated Development Environment (IDE) named Integrated Development and Learning Environment (IDLE) that can be used to develop bits of code into useful applications. If you have installed a fairly recent version of Python on your computer, you should have a version of IDLE that will come with the installation bundle. To start IDLE, you can type

```
koebbe% idle&
```

Note that there is an ampersand at the end of the command. This tells the operating system to run the command in the background. This is a Unix feature that allows you to run multiple processes in a single terminal. In a terminal emulator, invoking the IDLE application looks like the window displayed in the following figure.

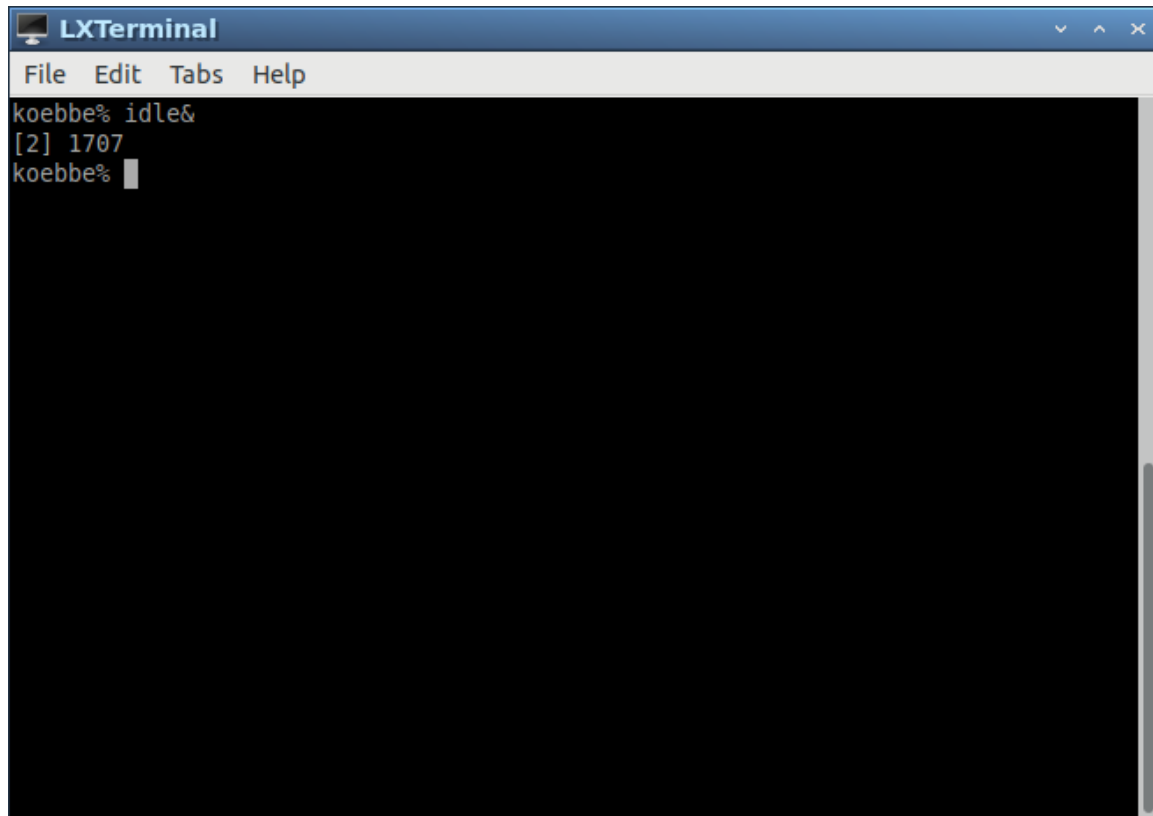


Figure 2: Starting IDLE in a terminal emulator.

Opening a New File to Write Code

To get the editor up and running, you can click on the file menu in the IDLE window and click on the New File item in the drop down menu. The window containing IDLE will look like the next figure.

The Final Version of the Plotting Code in Python

The next step is to type in Python code that will provide the 2-d plot that we want. The figure below shows the final code in the IDLE new file window. The code can be run using the Run menu at the top of the window. When the module is running you will be prompted to see if you want to save any changes to the code. It is usually a good thing to save the changes.

```
#
# import some stuff
# -----
#
from matplotlib import pyplot as plt
import numpy as np
#
# this is hardwired for a logistic function - so give initial time and final
# time
# ----
#
start = 0.0
end = 20.0
#
# set an array of input/time values
# -----
#
t = []
#
# initialize a variable to keep track of the time at each iteration and append
# the initial value to the array
# -----
#
x = 0.0
t.append(x)
#
# compute the time increment between samples
# -----
#
dx = ( end - start ) / 201.
#
# set the functional form for the logistic solution
# -----
#
expression = '(400.0 * np.exp(0.8*x)) / (500.0 + 3.0 * np.exp(0.8*x))'
#
# initialize an array for the output/population density values
# -----
#
p = []
p.append(eval(expression))
#
# set the loop iterator and start the while loop
# -----
#
```

```

l = 0
while l < 200:
    #
    # compute the current value of the expression
    # -----
    #
    p.append(eval(expression))
    #
    # move on to the next value of the input
    # -----
    #
    x = x + dx
    #
    # append the new value to the input array
    # -----
    #
    t.append(x)
    #
    # plus one the iterator
    # -----
    #
    l += 1

#
# do the plot thing in matplotlib
# -----
#
plt.xlabel('time values')
plt.ylabel('population density values')
plt.title('Logistic Population Growth Example')
plt.plot(t, p)
plt.show()

```

The code can be copied from these lecture notes and saved into a file. The Python code is a hardwired version to graph the single expression

```
expression = '(400.0 * np.exp(0.8*x)) / (500.0 + 3.0 * np.exp(0.8*x))'
```

in python. It might be a good idea to write a general code. This will be done in the next topic.

The Final Result

When the Run menu item is selected, the result will be a new window popping up that displays the graph in the first figure of this section of the notes. The nice thing is that you can save a copy of the figure in several formats. Your instructor usually likes a Portable Network Graphics (png) format.

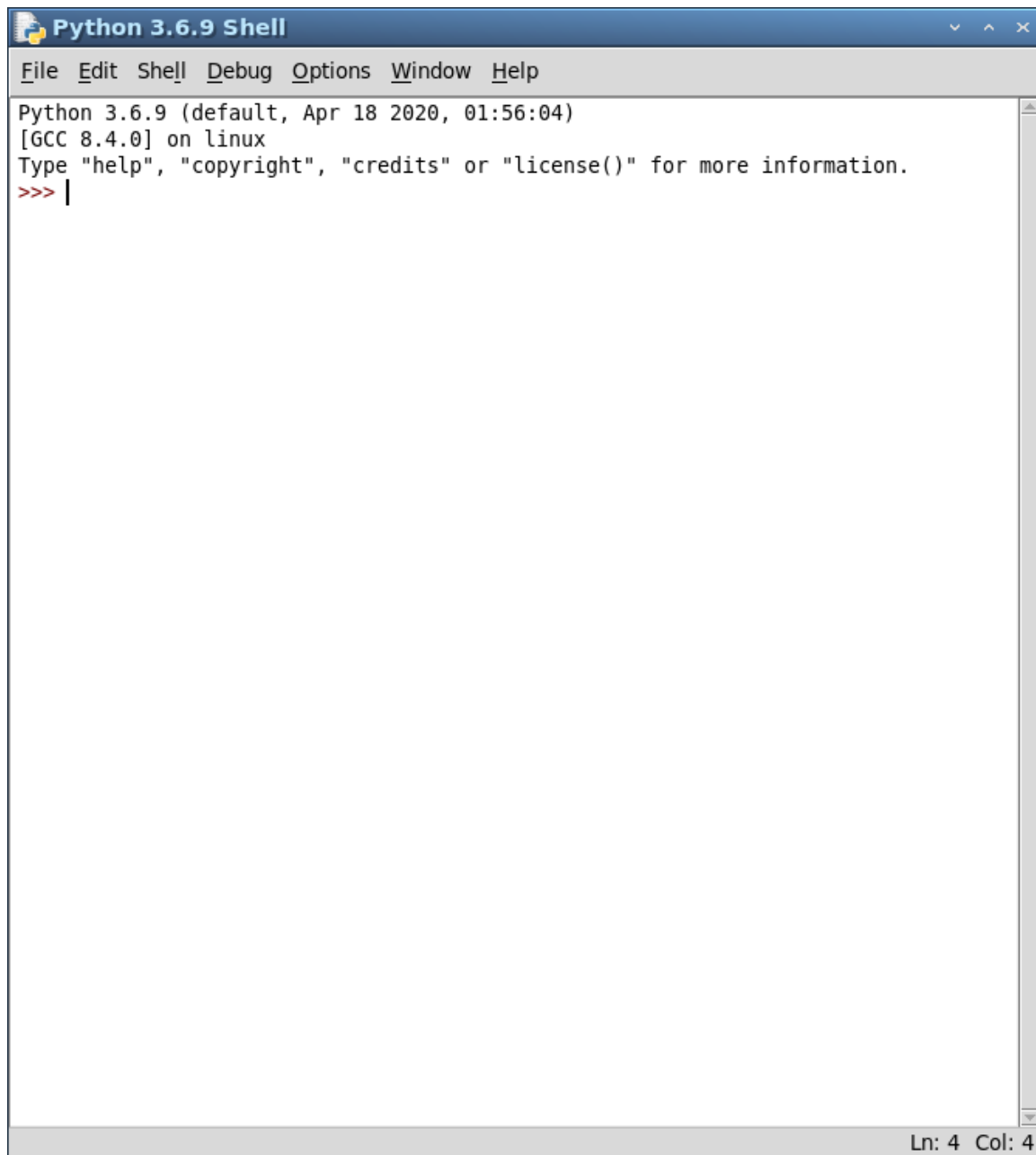
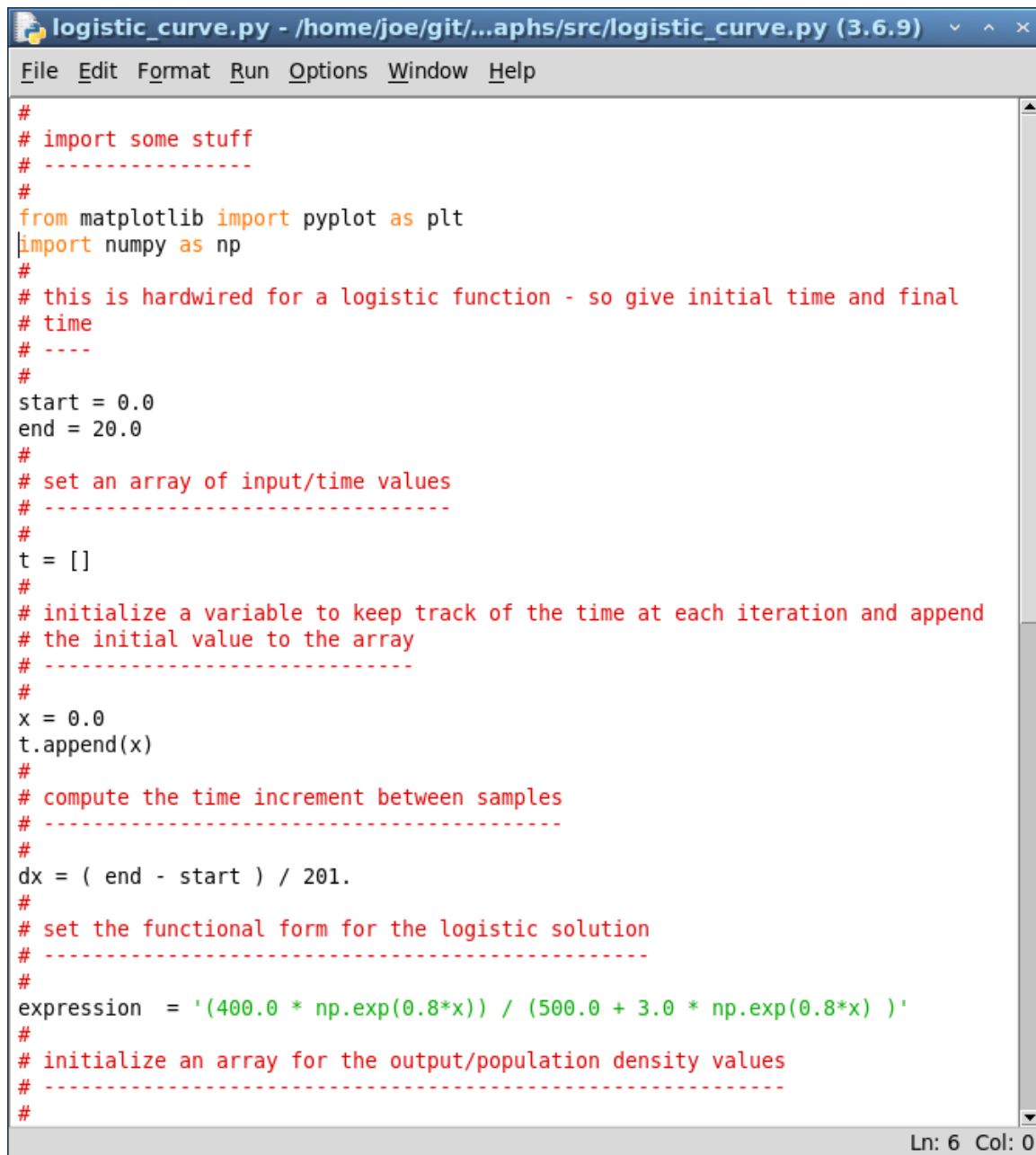


Figure 3: Opening a New File to write code in Python.



```
logistic_curve.py - /home/joe/git/...aphs/src/logistic_curve.py (3.6.9)
File Edit Format Run Options Window Help
#
# import some stuff
# -----
#
from matplotlib import pyplot as plt
import numpy as np
#
# this is hardwired for a logistic function - so give initial time and final
# time
# ----
#
start = 0.0
end = 20.0
#
# set an array of input/time values
# -----
#
t = []
#
# initialize a variable to keep track of the time at each iteration and append
# the initial value to the array
# -----
#
x = 0.0
t.append(x)
#
# compute the time increment between samples
# -----
#
dx = ( end - start ) / 201.
#
# set the functional form for the logistic solution
# -----
#
expression = '(400.0 * np.exp(0.8*x)) / (500.0 + 3.0 * np.exp(0.8*x))'
#
# initialize an array for the output/population density values
# -----
#
Ln: 6 Col: 0
```

Figure 4: The final code in the window to produce a 2d-plot of a logistic population model.

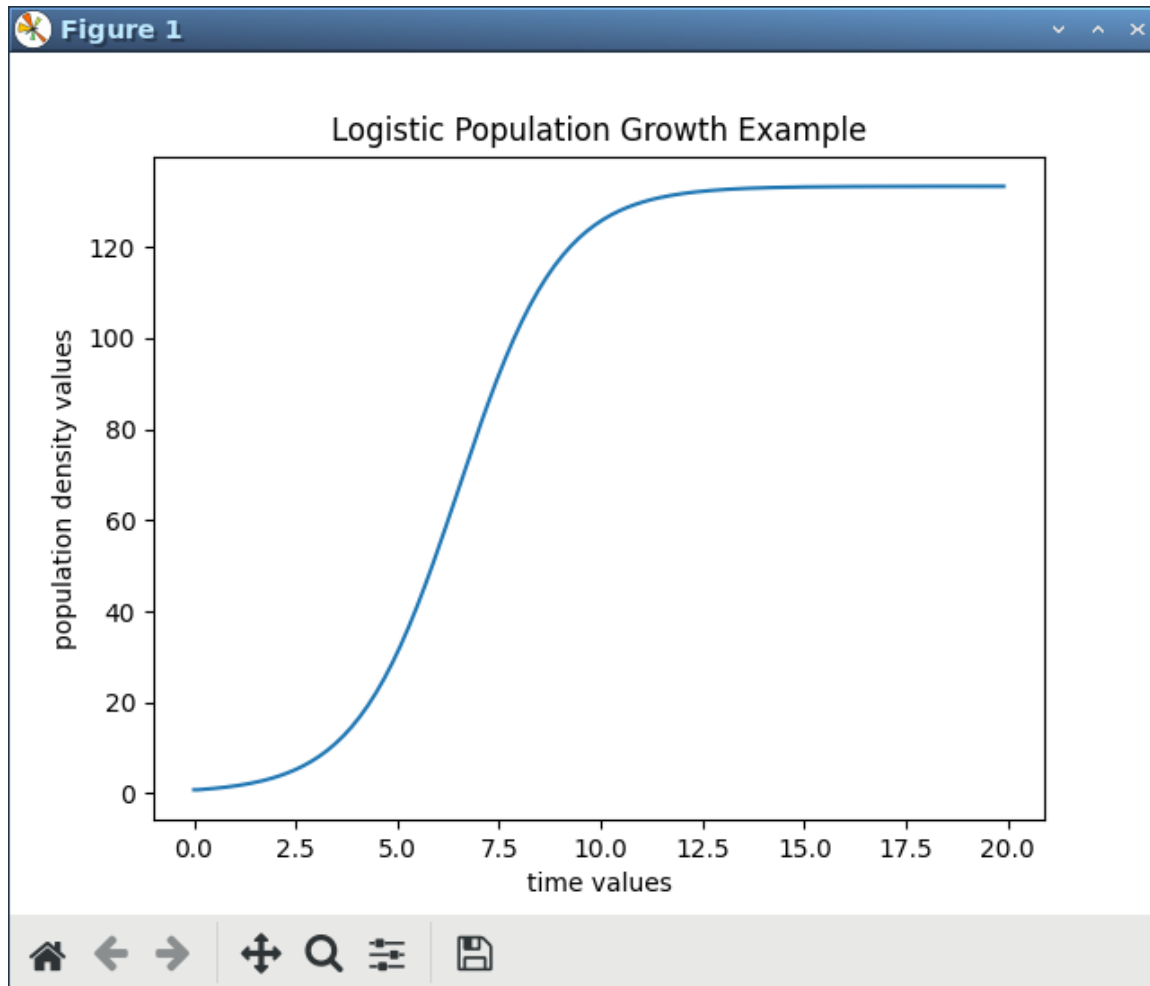


Figure 5: The output from the 2d plotting script is a window that shows the plot with a toolbar at the bottom of the window. The controls on the toolbar can be used to save a copy among other options.