Chapter 5

Lab 5: Arrays, Plotting and Chaos

5.1 Introduction

This lab will introduce a fundamental element of scientific python, the numpy array, and use them to produce plots. We will also consider the non-linear logistics map and examine bifurcation in the approach to chaos.

If you can complete all of the plots in Section 5.6 including the optional plot and *without any instructor help* then you may omit the plots from the preceding sections.

5.2 Preparation

This lab will rely on the material from Sections 1.4.1 to 1.4.2 and 1.5.1 to 1.5.2 of the Scientific Python Lecture notes.
For this lab you will need the "pandas" library, which we did not install when we created the environment. If you have not already done so, after activating the environment, type

```
pip install pandas
```

on the command line.

This is the first lab that relies on inline plotting, so make sure you are starting your notebook with the "line magic" and install the required libraries:

```
%matplotlib inline
import matplotlib.pyplot as plt.
import numpy as np
import pandas as pd
```

A Numpy array is a grid of values. Unlike Python lists, the elements of a numpy array all have the same data type, which makes them much more computationally efficient. Choices for the data type include the built-in python integer, float, and bool types. The numpy library provides a wide range of analysis tools that are mostly centered on the numpy array type.

Numpy arrays can be constructed easily from a Python list:

```
a = np.array([1.3,7.2,4.1,0.0])
b = np.array([[1,2],[3,4]])
print(a)
```

```
print(b)
print(np.shape(a))
print(np.shape(b))
```

This is conveninent when you have specific values you want to define by hand. Another possibility is to construct the numpy array by calling a function designed specifically for the purpose:

```
a = np.linspace(0,1,11)
print(a)
b = np.arange(0,5,1)
print(b)
```

Both `linspace` and `arange` allow you to specify the range of values you want, but with `linspace` you specify the number of points you want whereas with `arange` you specify the step size.

One of the great joys of using numpy arrays comes from the fact that most operators are applied elementwise automatically, without the need to explictly write a for loop:

```
a = np.arange(0,5,1)
b = 10
print(a)
print(b)
print(a+b)
```

Notice how the value of b (10) is added to *every element* of a, without the need to explicitly loop over every element. Try modify the example code to multiply every element of a by b. Then try raising each element of a to the power of 2.

△ **Jupyter Notebook Exercise 5.1:** Use numpy arange and elementwise operations to implement a function `def powers(a, n)` which returns a numpy array containing the first powers of a from 1 to a^n. So for example `print(powers(2,4)` should output `[ 1 2 4 8 16]`
vskip 1cm

Numpy arrays can also be built up element by element using the append function:

```
a = np.array([])
print(a)
a = np.append(a, 1)
print(a)
a = np.append(a, 3)
print(a)
a = np.append(a, 4)
print(a)
```

This example creates an empty numpy array and then adds one element at a time.

△ **Jupyter Notebook Exercise 5.2:** Run the snippet above and observe the output.

△ **Jupyter Notebook Exercise 5.3:** Implement a function `def recur(n,x)` which returns an array containining the first n values of the recurrance relationship $x_{i+1} = 2\,x_i + 1$ starting from $x_0 = x$. Use the following algorithm:

```
1  Parameter x # starting value
2  Parameter n # number of iterations
3  Create empty array a
4  Repeat n times:
5     append x to array a
6     x := 2x+1
7  print a
```

Test your code for $x = 1$ and $n = 5$ and ensure that it produces the correct output: `[ 1. 3. 7. 15. 31.]`.

5.3 Plotting with Scientific Python

Basic plotting in Python requires two numpy arrives: one for the x coordinates and one for the y coordinates. Consider the following very simple plot:

```
x = np.array([0.0, 1.0, 2.0, 3.0, 4.0,  5.0])
y = np.array([0.3, 3.2, 5.8, 9.0, 12.4, 14.7])
plot(x,y,"bo")
```

Here, the "bo" options specifies blue circles. Now consider:

```
x = np.linspace(0, 1, 100)
y = np.sin(np.pi*x)
plt.plot(x,y,"r-")
```

Here the "r-" option specifies red line. Including 100 points (as done here) results produces a smooth looking curve.

Now promise me that you will never make another plot without labeling the x and y axes! Here's another example will all the bells and whistles you need to make a professional looking plot:

```
UPPER = 2
LOWER = 0
tau   = 2*np.pi
x = np.linspace(LOWER,UPPER,100)
s = np.sin(tau*x)
c = np.cos(tau*x)
plt.plot(x,s,"b-",label="sin")
plt.plot(x,c,"r-",label="cos")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Two Periods of a Sine and Cosine")
plt.legend(frameon=False)
plt.show()
```

Make sure you understand all of the features demonstrated here:

- Variables `UPPER` and `LOWER` located at the top of the snippet, allowing for easy adjustment of parameters that affect the plot.
- Use of `np.linspace` to define an array of x values, with plenty of them (100) to produce nice smooth curves.
- Creation of two different arrays of y values, one for sin and one for cos.
- Plotting the arrays of x and y values with `plt.plot` using the "-" option for a line and color blue("b") for sin and red("r") for cos.
- Defining appropriate axis labels with `plt.xlabel` and `plt.ylabel`.
- Adding a title with `plt.title`

- Creation of a legend using the `label` optional argument to `plt.plot` and the plot.legend() command. Removing the frame with option `frameon=False`

It is written so concisely and intuitively, you might not even notice what is going on with the line:

```
s = np.sin(tau*x)
```

Remember that x here is a numpy array of 100 elements. The `tau*x` multiples every element of x by our value tau. The `np.sin(tau*x)` then takes the sine of each element. The resulting numpy array, also of 100 elements, is referenced by variable s. Each element of s contains $\sin(\tau x)$ for the corresponding element of the array x. It takes some getting used to for programmers used to explicitly writing for loops for things like this, but ultimately, the fact that python handles so much of this bookkeeping for us is what makes it a very fun language to work with.

△ **Jupyter Notebook Exercise 5.4:** Plot the $sinc(x)$ function as a smooth line in the x range from -5 to 5. Add appropriate labels to each axes. Include a legend identifying the sinc function. For the line color, use any color other than red or blue.

5.4 Plotting with Pandas

Pandas is a powerful tool for manipulating data. It's essentially an internal spreadsheet. Download the file "lab5.csv" from "Files/labs/lab5" at the Canvas site. This is a comma separated file with 3 columns: "x", "f1", and "f2".

△ **Jupyter Notebook Exercise 5.5:** Read this file in with

```
    df = pd.read_csv('lab5.csv')
```

Create a new column, which is the product of the first two, by typing

```
    df['f3'] = df['f1'] * df['f2']
```

Now plot all three functions on the same plot by typing

```
    plt.plot(df['x'],df['f1'],label='f1')
```

etc. Include a legend and a label on the x axis.

5.5 Plot and Distributions

Use `numpy.random.normal` to create an array with 10000 elements. This array of number will follow a Gaussian distribution (use $\mu = 10$ and $\sigma = 3$).

△ **Jupyter Notebook Exercise 5.6:** Histogram this distribution using the plt.hist() function. Remember to add labels on the axes and a title for the plot.

△ **Jupyter Notebook Exercise 5.7:** Implement a function `def Gaussian(x, sigma, mu)` which, given sigma and mu returns a Gaussian. Use the following Gaussian formula to create this function. The try to overlap this function to your plot to confirm that the random.normal distribution is a Gaussian distribution. Remember, you will have to scale this to match your histogram.

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$$

5.6 The Logistics Map

The logistics map is the recurrence relation

$$p_{i+1} = r\ p_i\ (1 - p_i)$$

which calculates the value of p for step $i + 1$ from step i. The variable p can represent the ratio of a population to its maximum possible value, and each iteration (n) is a step in time (such as one year). Each year, the population increases due to birth and decreases due to starvation as the population approaches it's maximum value (p near 1). The growth (or decline) of the population is controlled by the growth rate parameter r. We will only consider r in the range from $[0, 4]$ which keeps p in the range $[0, 1]$. This is a simple non-linear model which illustrates chaotic behavior.

△ **Jupyter Notebook Exercise 5.8:** Implement a function `def logmap(p, r)` which returns the next iteration (p_{i+1}) of the logistic map for parameter r and $p_i = p$. Test your code by showing that for $r = 3.0$ and $p_0 = 0.1$ the next five iterations are: 0.27, 0.5913, 0.725, 0.5981 and 0.7211.

△ **Jupyter Notebook Exercise 5.9:** Use your `logmap` function to build an array containing the first n entries in the logistics map starting from $p_0 = 0.1$. Check that the output is correct by comparing with the results from the previous exercise for $n = 6$.

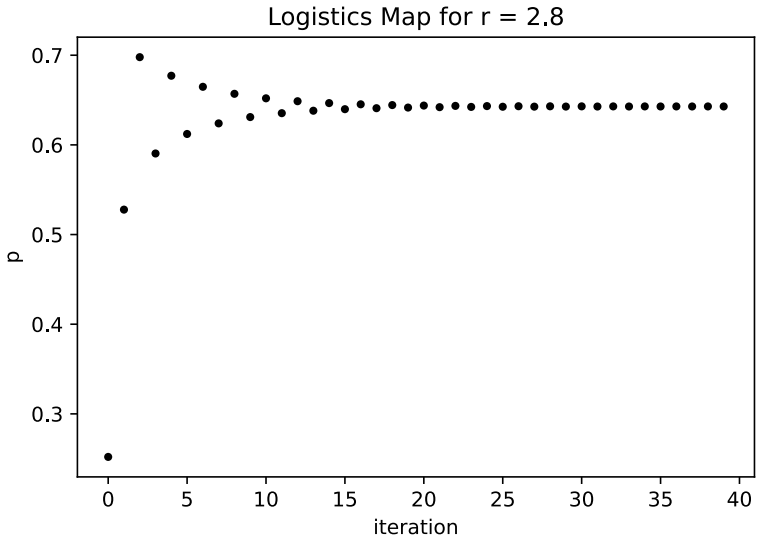

Figure 5.1: Convergence of the logistic map for $r = 2.8$

△ **Jupyter Notebook Exercise 5.10:** Plot the time evolution of the logistics map for $r = 2.8$ for 40 iterations, starting from $p_0 = 0.1$ as in Fig. 5.1. To create a plot, you'll need an array containing the p values (these correspond to the y axis in the plot) which you can construct as in the previous

exercise. But what about the x axis values? Since your array of p values contains $[p_0, p_1, p_2 \ldots p_n]$ the corresponding array of indices is simply $[0, 1, 2 \ldots 3]$ which you can construct using `np.arrange`.

Your results from the previous exercise should reproduce Fig. 5.1. This shows that for $r = 2.8$ the logistics map converges to a value of about 0.64. But this non-linear recursion relationship does not always converge to a single value.

△ **Jupyter Notebook Exercise 5.11:** Plot the time evolution of the logistics map for $r = 3.2$ for 40 iterations, starting from $p_0 = 0.1$.

Notice that now the system oscillates between two values (near 0.5 and 0.8).

△ **Jupyter Notebook Exercise 5.12:** Plot the time evolution of the logistics map for $r = 3.5$ for 40 iterations, starting from $p_0 = 0.1$.

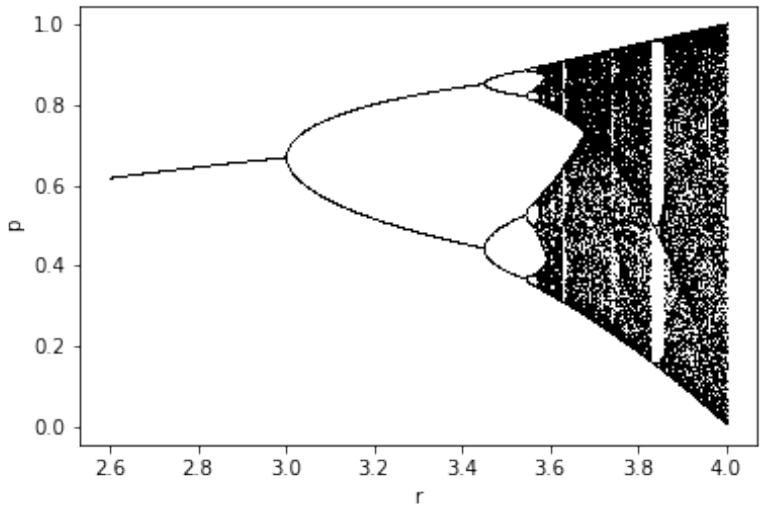

Figure 5.2: Long-term behavior of the logistics map as a function of parameter r.

Notice that now the system oscillates between four values. This is an example of bifurcation in the approach to chaos. To show this more clearly, we'd like to produce a plot as in Fig. 5.2. For each r value, this shows the p values from 100 iterations *after* the first 1000 iterations. This shows the *long term behavior* of the logistics map. We can clearly see that for $r = 2.8$ the p values converge to one single value and for $r = 3.2$ there are two values just as in the previous exercises. These bifurcations continue until the system becomes chaotic (oscillating between many different values) with occassional windows of stability.

To produce this plot for yourself, start by considering this snippet:

```
Nr = 5
r = np.linspace(2.6,4,Nr)
p = logmap(0.1,r)
print(p)
p = logmap(p,r)
```

```
print(p)
```

Here we create a numpy array of r values, and pass that to our `logmap` function instead of a single value. The operations within the function are applied elementwise to the array, and the result is that instead of a single p value, the call to to `logmap(0.1,r)` returns an array of p values, one for each r value. This is just what we need to make the plot in Fig. 5.2.

△ **Jupyter Notebook Exercise 5.13:** Run (and understand!) the snippet and make a plot of p versus r. Understand that you are plotting p_2 as a function r! Use the `\"k,\"` format option (black pixels) when plotting. Comment out the print statements and increase Nr to 100.

△ **Jupyter Notebook Exercise 5.14:** Make a plot that is *almost* like that of Fig. 5.2 by plotting p_{1001} versus r. Hint: in the code above, instead of one call to `p = logmap(p, r)` use a for loop which calls this 1000 times.

You should start to see features of the Fig. 5.2 but you are only plotting one p value for each r. To see the bifurcations and chaos, you'll need to plot about 100 p values at each r value.

△ **Jupyter Notebook Exercise 5.15:** Reproduce the plot in Fig. 5.2. Hint: instead of plotting just p_{1001} as in the previous exercise, plot the next 100 values as well. Increase the number of r values plotted to 1000. Add appropriate labels to each axis.

△ **Jupyter Notebook Exercise 5.16:** (Optional) The bifurcation diagram which you have constructed exhibits self-similarity. If you zoom into an appropriate region of the diagram, you will find a diagram which looks quite similar to the original diagram. Produce a plot that demonstrates this self-similiarity by zooming into a particular region.