MATHEMATICAL COMPUTING WITH PYTHON

Dennis Hall
Angelo State University

LibreTexts™

Angelo State University

Mathematical Computing with Python

Dennis Hall

TABLE OF CONTENTS

Licensing

CHAPTER OVERVIEW

1: Introduction to Python

All examples and resources in this textbook are written in the Python programming language. Python is an open source (under OSI), interpreted, general-purpose programming language that has a large number of users around the world. It is the top-ranked language in January 2023 by the TIOBE programming community index[1], a measure of popularity of programming languages. A more robust tutorial of the language can be found at https://www.python.org/.

This book uses Replit, an online browser-based IDE for Python and several other languages. Replit may be accessed for free at https://www.replit.com/.

Much of the content in this chapter is from "First Semester in Numerical Analysis with Python", available at https://open.umn.edu/opentextbooks/textbooks/925.

Tags recommended by the template: article:topic-guide

[1] https://www.tiobe.com/tiobe-index/

1.1: Replit

Getting Started with Replit

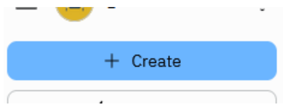
For new and casual users, we recommend using the free online Python environment Replit, available at https://www.replit.com.

After registering an account on Replit and answering (or skipping) the initial questions, you will be presented with the Replit Home page.

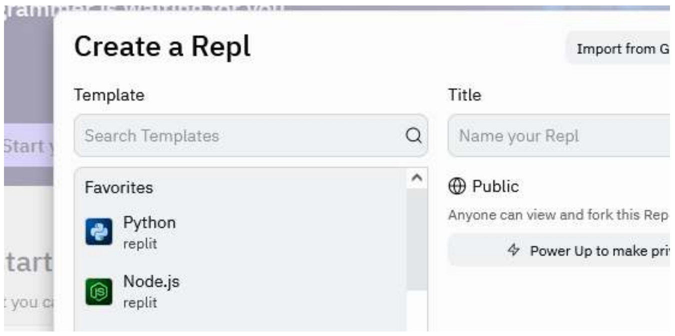
What is a Repl?

Repl stands for "Read Evaluation Print Loop" and, at Replit, is an interactive environment for developers to create small projects. Each of our small programs will take place inside individual Repls.

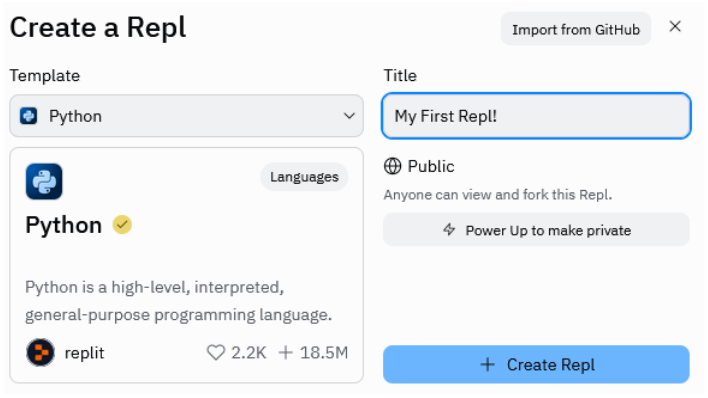
1. To start, click the "+ Create" button.



2. Since we will be using Python, choose the "Python" replit.



3. Type in a Title for your Repl, or leave it as the randomly-generated suggestion.



4. Click "Create Repl".

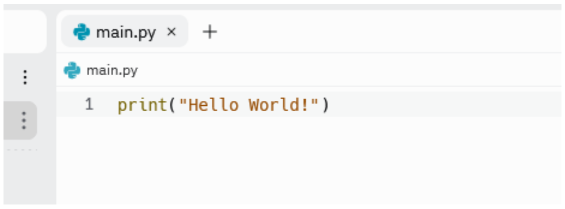
Your first Repl usually has a small tutorial. Feel free to read through each of the prompts to get familiar with the environment.

Hello World!

Traditionally, your first program in a new language is a program that simply prints "Hello World!". This is very easy in python and can be done by typing the following line as line 1 in your newly-created Repl.

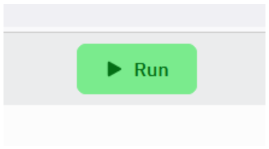
```
1  print("Hello World!")
```

In Replit, it should like this:

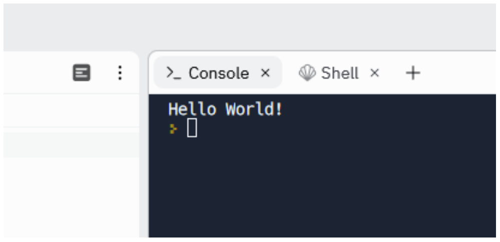


```python
print("Hello World!")
```

To run your program, click the green "Run" button at the top.



Now, on the right side, you will see the output in Python's "console". The console is where text-based programs will display their result.



```
Hello World!
>
```

Congratulations! You've written your first Python program.

1.2: Python Basics

Computations

Let's get started with some basic computations.

The following code will produce no output:

```
2+3
```

```
output:
```

This is because here, you are asking Python to add 2 and 3 together, but you aren't specifying what to do with the output. Instead, we will add a print command, so that Python knows that we want to actually see what the computation yields.

```
print(2+3)
```

```
output: 5
```

Note that, unlike our Hello World! program from the previous section, we do not include quotes here. That's because we do not want to literally print the string "2+3", but instead we want Python to add the numbers together and then display the result.

Most arithmetic is intuitive, though exponents are a bit different in Python than in other languages. For example, to calculate 3^7, we use two asterisks **:

```
1  print(3**7)
```

```
output: 2187
```

Modulo Operator

Modulo in python is performed using the percent symbol **%**. The modulus of two numbers is the remainder after dividing the two numbers. For example,

```
1  print(7 % 2)
```

```
output: 1
```

As can be seen above, 7 mod 2 = 1 since $7 \div 2 =$ 3 with a remainder of 1.

Math Package

There are several mathematical functions that python does not have built in, so we must "import" the package. Here, we will import all functions (the wildcard *) from the **math** package. This gives us access to both the **sin** function and **pi**. We use them here.

```
from math import *
print(sin(pi/4))
```

```
output: 0.7071067811865475
```

One way to learn about a function is to search for it online in the Python documentation https://docs.python.org/3/. For example, the syntax for the logarithm function in the
math package is log(x[; b]) where b is the optional base. If b is not provided, the natural logarithm of x (to base e) is computed.

```
from math import *
print(log(4,2))
```

```
output: 2.0
```

NumPy arrays

NumPy is a useful Python package for array data structure, random number generation, linear algebra algorithms, and so on. A NumPy array is a data structure that can be used to represent vectors and matrices, for which the computations are also made easier. Import the NumPy package and define an alias (np) for it.

```python
import numpy as np
```

Importing a package like this means that you will need to use the prefix **np.** for any functions. For example, here is the basic syntax to create a 1-dimensional NumPy array using the **array** function from the NumPy package.

```python
import numpy as np
x=np.array([10,20,30])
print(x.dtype)
```

```
output: [10 20 30]
```

Note that Python automatically chooses the variable types. For example, we can check that the variable types for the array about are integers of 64 bits.

```python
import numpy as np
x=np.array([10,20,30])
print(x.dtype)
```

```
output: int64
```

If we input a real number that is not an integer, Python will change the type accordingly:

```python
import numpy as np
x=np.array([10,20,30,0.1])
print(x.dtype)
```

```
output: float64
```

Here, float64 means the variable type is a 64-bit floating point number. Floating point numbers are decimal values with an undefined number of digits before and after the decimal point (so the decimal point can "float" as needed).

A 1D NumPy array does not assume a particular row or column arrangement of the data, and hence taking transpose for a 1D NumPy array is not valid. Here is another way to construct a 1D array, and some array operations:

```python
import numpy as np
x = np.array([10*i for i in range(1, 6)])
print(x)
```

```
output: [10 20 30 40 50]
```

If we wish to print a certain element of an array, we use the bracket notation **x[index]**, where the element in the first position is index 0.

```python
import numpy as np
x = np.array([10*i for i in range(1, 6)])
print(x[2])
```

```
output: 30
```

For the last entry, we use the index -1. Note that we omit the first two lines for the following examples.

```
...
print(x[-1])
```

```
output: 50
```

To find the minimum entry:

```
...
print(min(x))
```

```
output: 10
```

To add them together:

```
...
print(np.sum(x))
```

```
output: 150
```

To append a new entry:

```
import numpy as np
x = np.array([10*i for i in range(1, 6)])
print(x)
x=np.append(x,99)
print(x)
```

```
output: [10 20 30 40 50]
[10 20 30 40 50 99]
```

To find the number of entries:

```
import numpy as np
x = np.array([10*i for i in range(1, 6)])
print(x.size)
```

```
output: 6
```

The NumPy package has a wide range of mathematical functions such as sin, log, etc., which can be applied elementwise to an array:

```
import numpy as np
x = np.array([1,2,3])
x = np.sin(x)
print(x)
```

```
output: [0.84147098 0.90929743 0.14112001]
```

1.3: Plotting

There are several packages for plotting functions and we will use the PyPlot package. Similar to the NumPy package, we must import the PyPlot package before we use it. We will use the alias plt. The first time you import PyPlot during a session, it will need to automatically download the package and install the dependencies.

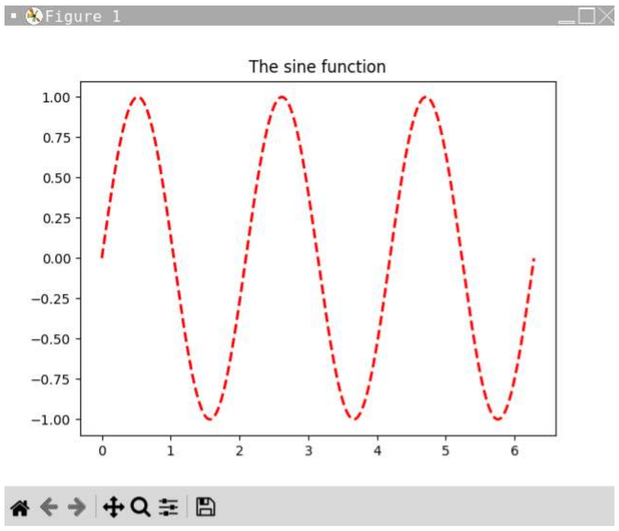
```
import matplotlib.pyplot as plt
```

```
output:
```

To actually plot a graph, we will also call the NumPy package so that we have access to the sin function.

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

x = np.linspace(0, 2*np.pi, 1000)
y = np.sin(3*x)
plt.plot(x, y, color='red', linewidth=2.0, linestyle='--')
plt.title('The sine function');
plt.show()
```

```
output:
```



Observe that, in the code above, we define the x-values we desire to be plotted using NumPy's **linspace** function. This function returns evenly spaced numbers over a specified interval. Here, we called for 1000 evenly-spaced numbers between 0 and $2pi$. Then, we defined the y-values as a function of x. Here, we used $y = \sin(3x)$.

In the following example, we plot two functions, $\sin(3x)$ and $\cos(x)$ simultaneously.

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

x = np.linspace(0, 2*np.pi, 1000)
y = np.sin(3*x)
z = np.cos(x)
plt.plot(x, y, color='red', linewidth=2.0, linestyle='--', label='sin(3x)')
plt.plot(x, z, color='blue', linewidth=1.0, linestyle='-', label='cos(x)')
plt.legend(loc='upper center');
plt.show()
```

```
output:
```

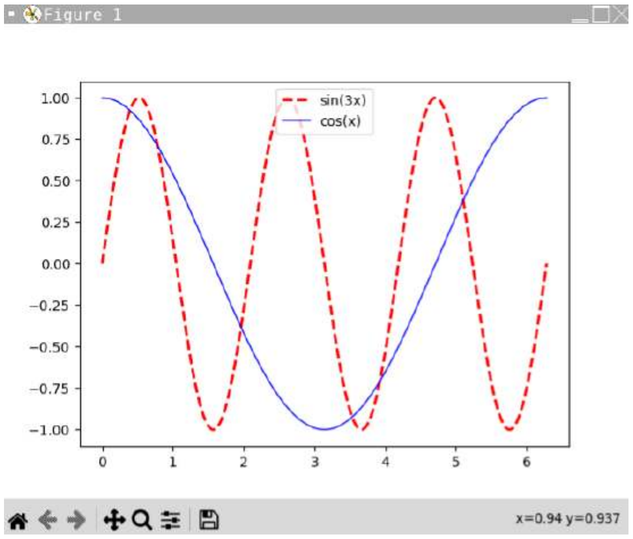


Figure 1
sin(3x)
cos(x)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
1
2
3
4
5
6
x=0.94 y=0.937

1.4: Logic and Loops

Logic Operations

We will often need to determine whether two numbers are the equal, whether one is greater, lesser, etc. To do this, we use logic operations. Here are some basic logic operations.

Equality:

```
print(2 == 3)
```

```
output: False
```

```
print(2 == 2)
```

```
output: True
```

We can check if a number is even by using the modulo operation:

```
print(53 % 2 == 0)
```

```
output: False
```

Inequality:

```
print(2 <= 3)
```

```
output: True
```

```
print(2 > 3)
```

```
output: False
```

An exclamation point with an equals means "Not Equal". It will return True if the quantities are not equal. It will return False if the quantities are equal.

```
print(2 != 3)
```

```
output: True
```

And/Or/Not:

The **and** operation will return true if and only if both statements return true.

```
print((2 == 2) and (2 < 3))
```

```
output: True
```

The following will return False, since it is False that "2 does NOT equal 2" AND "3 equals 3". The statements are not both true.

```
print((2 != 2) and (3 == 3))
```

```
output: False
```

The **or** operation will return true if at least one of the statements return true.

The following will return True, since at least one of them is true.

```
print((2 != 2) or (3 == 3))
```

```
output: True
```

If/Else

Sometimes we wish to take an action only if a certain statement is true. To do this, we use **If** statements. The following snippet of code checks whether the number 37 is even or not. If it is, we print "37 is even". Otherwise, we print "37 is odd".

```
if (37 % 2 == 0):
  print("37 is even")
else:
  print("37 is odd")
```

```
output: odd
```

For Loops

Sometimes we wish to perform the same operations to several numbers or objects in a row. One way to do this is with **for loops**. The following code defines an array, and then prints each number one by one with the message "is a cool number!" after each.

```
import numpy as np

coolNumbers=np.array([2,3,5,8,9,14])
for x in coolNumbers:
  print(x, " is a cool number!")
```

```
output: 2 is a cool number!
3 is a cool number!
5 is a cool number!
8 is a cool number!
9 is a cool number!
14 is a cool number!
```

We can combine the program from the if statement section to check if several numbers are even.

```
import numpy as np

coolNumbers=np.array([2,3,5,8,9,14])
for x in coolNumbers:
  if (x % 2 == 0):
    print(x," is even")
  else:
    print(x," is odd")
```

```
output: 2 is even
3 is odd
5 is odd
8 is even
9 is odd
14 is even
```

If we wish to perform the above on every number from 1 to 7, we can use Python's **range** function. The range function has syntax **range(start,stop,step)**, and returns all integers in $[start, stop)$ and counts the numbers up by *step*. Note that the number entered for *stop* is not included.

```
import numpy as np

for x in range(1,8,1):
  if (x % 2 == 0):
    print(x," is even")
  else:
    print(x," is odd")
```

```
output: 1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
```

While loops

Another way to construct loops is with the **While** construct. Instead of going through a list, While loops continues until certain conditions are met. For example, if we want to keep counting through numbers until we get to 20, we can use the following.

```
n=1
while n<=8:
  print(n)
  n=n+1
```

```
output: 1
2
3
4
5
6
7
8
```

1.5: Random Numbers

Random numbers are part of the NumPy package. For example, if we want to create 5 random numbers in $(0,1)$, we use the following.

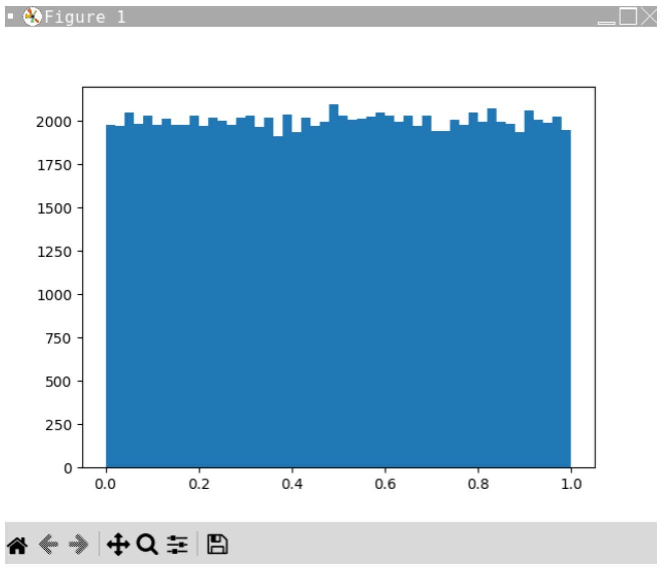
```python
import numpy as np
print(np.random.rand(5))
```

output: [0.99332819 0.98682806 0.32328937 0.70417298 0.59908967]

Using the same function above, together with the matplotlib package, we can create a histogram of 100000 real numbers (with 50 bins) with the following code to see the distribution.

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

y = np.random.rand(10**5)
plt.hist(y, 50);
plt.show()
```

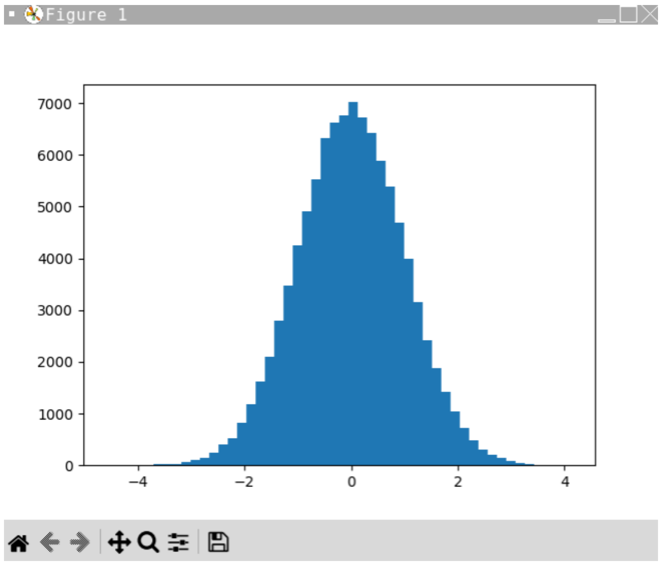
output:



If, instead, we wanted a normal distribution, we can use the **randn** function.

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

y = np.random.randn(10**5)
plt.hist(y, 50);
plt.show()
```

output:



Random Package

For a bit more flexibility with random numbers, we can use Python's **random** package. This allows us to use, for example, the **randint** function to give us a random integer between two numbers, inclusive. We do so here:

```
import random
x=random.randint(1,5)
print(x)
```

```
output:    3
```

CHAPTER OVERVIEW

2: Monte-Carlo Simulation

2.1: Introduction to Stochastic Simulation

A **stochastic simulation** is a simulation of a system that has variables that can change stochastically (randomly) with individual probabilities. These random variables are generated, inserted into a model of the system, their outputs are recorded, and then the process is repeated using a new set of random variables.

Suppose you have a coin and don't know the probability of it landing on heads versus tails. One way to approximate this probability is by flipping the coin over and over, recording the outcome each time, and then analyzing the results. Suppose your record the following.

Heads	Tails	Total
498	502	1000

Then you would approximate that your coin will land on heads 498/1000 = 49.8% of the time and heads 502/1000 = 50.2% of the time.

We, of course, will be using Python to do the work for us. Below, we will calculate the probability of landing on heads. We use the **random** package to find a random integer between 0 and 1, inclusive. We decide that 0 will represent heads and keep track of the number of "successes", that is the number of flips that land heads.

```
import random

tries = 10000
success = 0
for a in range(1,tries+1):
  flip = random.randrange(0,2)
  if flip == 0:
    success += 1
print(success/tries)
```

```
output:     0.4958
```

Of course, if you run the above program, you will likely get a slightly different probability. Our successive runs of the above program returned 0.5061, 0.4869, and 0.4994. Try increasing or decreasing the number of "tries" above to see how they affect the predictability of the probability returned.

2.2: Monte Carlo Simulation

Suppose that we make N simulations, each time drawing the needed random parameters x_i from a random number "black box" (for our purposes, that black box will be the random package's random number generator). We define the high-level output, that is the result of our simulation, of our system S to be $g(\vec{x})$. For simplicity, we say that $g(\vec{x})$ is a scalar. $g(\vec{x})$ can be virtually any output of interest, for example: the value of one state at a given time after an impulsive input, or the integral over time of the trajectory of one of the outputs, with a given input. In what follows, we will drop the vector notation on x for clarity.

Let the **estimator** G of $g(x)$ be defined as

$$G=\frac{1}{N}\sum_{j=1}^{N}g(x_j). \tag{2.2.1}$$

You may recognize this as a straight average. Indeed, taking the expectation on both sides,

$$E(G)=\frac{1}{N}\sum_{j=1}^{N}E(g(x_i)), \tag{2.2.2}$$

it is clear that $E(G)=E(g)$. At the same time, however, we do not know $E(g)$; we calculate G understanding that with a very large number of trials, G should approach $E(g)$.

Example: Dice

In the previous section, we used a Monte-Carlo simulation to approximate the probability that a flipped coin will land on heads. We can do a similar simulation to approximate the probability that a die will land on a 3:

```
import random

tries = 10000
success = 0
for a in range(1,tries+1):
  flip = random.randrange(1,7)
  if flip == 3:
    success += 1
print(success/tries)
```

```
output:     0.1604
```

This is relatively close to our expected probability of 1/6. Suppose instead that we want to roll two dice and add them up. We wish to find the probability that the sum of the two dice is 5. In this case, we call for two random numbers from 1 to 6, add them up, and call it a success if the sum is 5.

```
import random

tries = 10000
success = 0
for a in range(1,tries+1):
  flip1 = random.randrange(1,7)
  flip2 = random.randrange(1,7)
  if flip1 + flip2 == 5:
    success += 1
print(success/tries)
```

```
output:     0.1108
```

Again, this is close to our expected value of 4/36.

Example: Cards

Finding the probability of a certain hand of cards is a bit trickier, mainly because we have to find a way to generate the entire deck, and then draw cards from the deck. Using arrays, though, this task can be completed with relative ease.

```python
import numpy as np

card_values = np.array(['A',2,3,4,5,6,7,8,9,10,'J','Q','K'])
deck = np.array([])

for i in card_values:
    for j in range(1,5):
        deck=np.append(deck,i)
print(deck)
```

output:

```
['A', 'A', 'A', 'A', 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5,
5, 5, 5, 6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 8, 9, 9, 9, 9, 10,
10, 10, 10, 'J', 'J', 'J', 'J', 'Q', 'Q', 'Q', 'Q', 'K', 'K',
'K', 'K']
```

The above does *not* keep track of suits, however this will be good enough for our purposes. Suppose now that we wish to draw a card from the deck, and determine the probability that the card is an Ace.

```python
import random
import numpy as np

card_values = np.array(['A',2,3,4,5,6,7,8,9,10,'J','Q','K'])

deck = np.array([])
#This iterates all 52 cards into a deck
for i in card_values:
    for j in range(1,5):
        deck=np.append(deck,i)

tries = 10000
success = 0
for a in range(1,tries+1):
  card1 = deck[random.randint(0,51)]
  if card1=="A":
    success += 1
print(success/tries)
```

output: `0.0785`

This is approximately equal to the expected value of 4/52.

What is the probability that we draw two cards and both cards are aces? Since the second draw is affected by the first draw, we have to do a few things differently. First, we need to delete the first card from the deck before drawing the second (line 9 below). Next, we need to change the range of the second random index to account for the fact that we only have 51 cards remaining (line 10 below). Finally, we need to start with a new deck each time (line 6 below). All of this is done here, where we omit the deck generation:

```
01 #deck generation omitted: see previous example.
02
03 tries = 10000
04 success = 0
05 for a in range(1,tries+1):
06   testdeck = deck
07   index1=random.randint(0,51)
08   card1 = testdeck[index1]
```

```
09   testdeck=np.delete(testdeck,index1)
10   index2=random.randint(0,50)
11   card2 = testdeck[index2]
12   if (card1=="J" or card1=="Q" or card1=="K") and \
13       (card2=="J" or card2=="Q" or card2=="K"):
14     success += 1
15 print(success/tries)
```

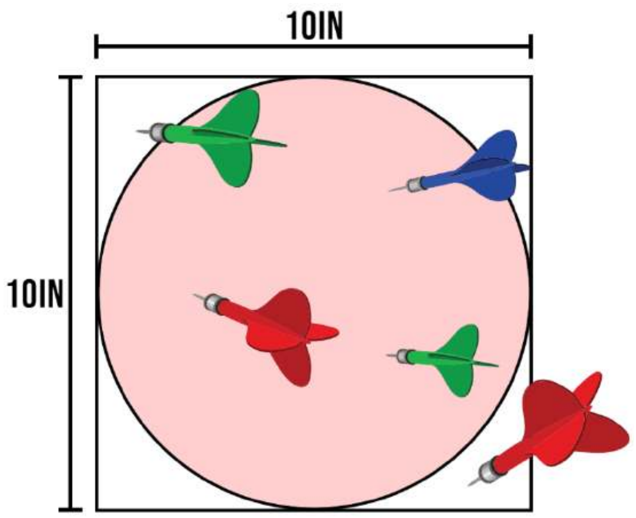
```
output:     0.049
```

This is close to our expected value of (12/52)*(11/52). Note that in the code above, we use a backslash one line 12 to tell python we are continuing the current code on the next line.

2.3: Integration

Finding Area

Let's use a Monte-Carlo type method to calculate area. Suppose we have the following circular dart board and want to know its area.



We enclose the dart board with a square of known area, in this case 100in^2, and perform following the following procedure:

1. Throw darts at the square, being careful not to favor any particular spot.
2. Record the percentage of darts that land inside the circle.
3. Multiply this percentage by the known area of the square (100in^2)
4. That's our approximation!

Of course, doing this by hand would take a lot of time. Instead, we can use Python. In our program, we will randomly choose an x-value and y-value using the **random.uniform(start,stop)** function. This function returns a random float in [start,stop). We will center our circle at the origin, so we will use **random.uniform(-5,5)** for both the x and y values.

Next, we need to have a way to determine whether the point (x,y) lies inside the circle or not. We know that the equation of the circle under discussion will be

$$x^2+y^2=5^2 \tag{2.3.1}$$

Solving for y, we get

$$y=\pm\sqrt{25-x^2}. \tag{2.3.2}$$

Thus, in our python code, we will check that

$$y\leq\sqrt{25-x^2} \quad \text{and} \quad y\geq-\sqrt{25-x^2}. \tag{2.3.3}$$

That is, that it is below the top half of the circle and above the bottom half of the circle. Here we go!

```python
import random

tries=10000
success=0
for a in range(1,tries+1):
  x=random.uniform(-5,5)
  y=random.uniform(-5,5)
  if (y<=(25-x**2)**(1/2)) and (y>=-(25-x**2)**(1/2)):
    success+=1
print(100*success/tries)
```

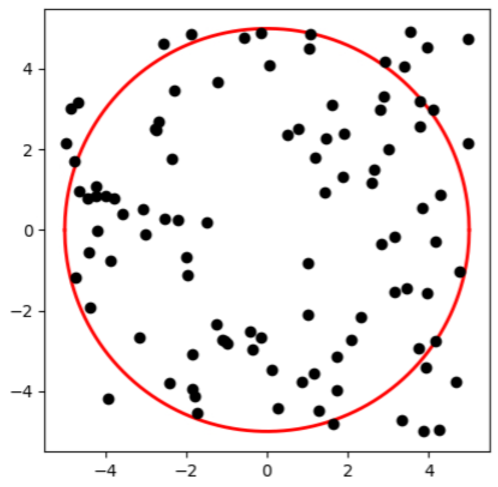
```
output:     78.52
```

Above, the percentage of "darts" that hit the circle is given by success/tries. We then multiplied that by the area of the square (100) to get the approximate area of the circle, which is given as 78.52, very close to our expected value of 25π.

For fun, we could plot each of the dots to see what our board looks like. Here is an example with 100 dots:

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

xc = np.linspace(-5, 5, 1000)
yc = (25-xc**2)**(1/2)
zc = -(25-xc**2)**(1/2)
plt.plot(xc, yc, color='red', linewidth=2.0, linestyle='-')
plt.plot(xc, zc, color='red', linewidth=2.0, linestyle='-')

#we need to make the axes have the same scale so it looks like a circle.
#these two lines do that.
ax = plt.gca()
ax.set_aspect('equal', adjustable='box')

tries=100
success=0
for a in range(1,tries+1):
  x=random.uniform(-5,5)
  y=random.uniform(-5,5)
  if (y<=(25-x**2)**(1/2)) and (y>=-(25-x**2)**(1/2)):
    success+=1
  plt.plot(x,y,'o',color='black')
print(100*success/tries)
plt.show()
```

```
output:     80
```



Integrals

Now that we've found area using a Monte-Carlo type simulation, you might have an idea of how to calculate integrals: just find the area under the curve!

Example: Use a Monte-Carlo type simulation to find the area under the curve over $[0, 4]$

$$y = 0.5x^2 \tag{2.3.4}$$

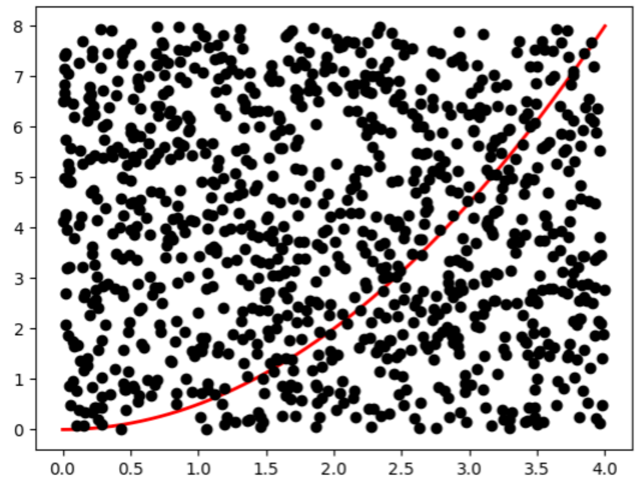
Answer: To start, we need to find a box that will fit the entire curve we are calculating. The x-values are easy: just use the given interval $[0,4]$. For the y-values, we need to find the least and greatest y-values that this curve achieves. Since this is a parabola, we can see that the least y-value is 0 and the greatest is $0.5*(4)^2=8$. So our box will be $[0,4]x[0,8]$ That means our random points will have x-values in [0,4] and y-values in [0,8]. We will determine whether the point is above or below the curve similar to the circle example above. We will consider a "success" - that is, consider our point to be under the curve - if:

$$y \leq 0.5x^2. \tag{2.3.5}$$

Note that we don't need to determine whether the point is above the x-axis because our y-values are only chosen from [0,8]. The area of our box is 4*8=32. Let's modify the above program:

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

xc = np.linspace(0, 4, 1000)
yc = 0.5*xc**2
plt.plot(xc, yc, color='red', linewidth=2.0, linestyle='-')

tries=1000
success=0
for a in range(1,tries+1):
  x=random.uniform(0,4)
  y=random.uniform(0,8)
  if (y<=(0.5*x**2)):
    success+=1
  plt.plot(x,y,'o',color='black')
print(32*success/tries)
plt.show()
```

```
output:     10.208
```



This is relatively close to our expected value (using integration) of 10.6667. If we increase the "tries" variable, we can get even closer. However, this will make our picture look like a blob!

Exercise

Use the techniques above to approximate the integral

$$\int_{\pi/2}^{\pi} \tag{2.3.6}$$

a. 11

b. 3
c. -4

Answer

a. $\frac{11}{1}$
b. $\frac{3}{1}$
c. $-\frac{4}{1}$

CHAPTER OVERVIEW

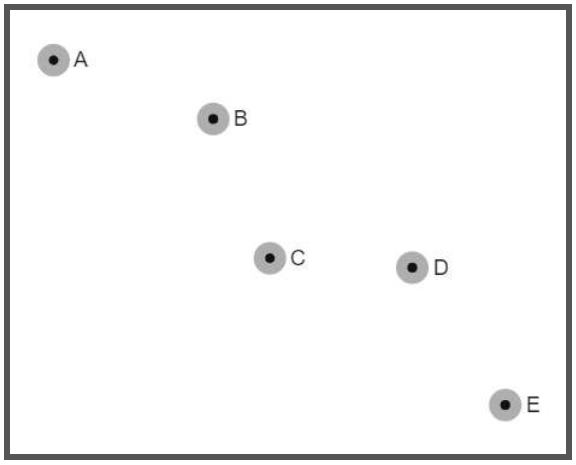
3: Interpolation and Curve Fitting

Discrete data sets or lists of information are commonly involved in modeling real-world scenarios. The data may come from observations in the field, experimental results, or analytical calculations. In this chapter, we discuss the differences between interpolation and curve fitting and find ways to numerically apply these techniques.

3.1: Introduction to Interpolation and Curve Fitting

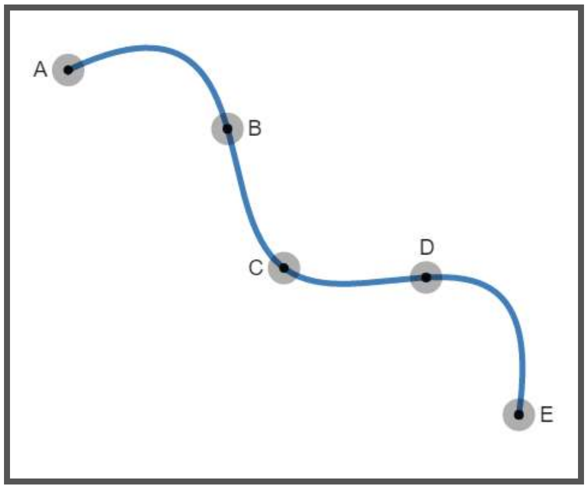
Given a set of data, we might want to approximate the value in between, after, or before what is given. To do this, we can generate a curve by using either interpolation or curve fitting. **Interpolation** is a method of estimating the value of a function at a given point within the range of a set of known data points. It involves finding a polynomial that fits a set of data points exactly, rather than just approximating them. **Curve fitting**, on the other hand, is the process of finding the best-fitting curve, where the goal is to find a model that captures the underlying trends in the data, rather than fitting the data points exactly.

For example, suppose we are given the following data points. Perhaps this data represents the temperature over several days, or the location of a board over the past several hours. We are wanting to estimate what the temperature was - or where the boat was - in between data points D and E.



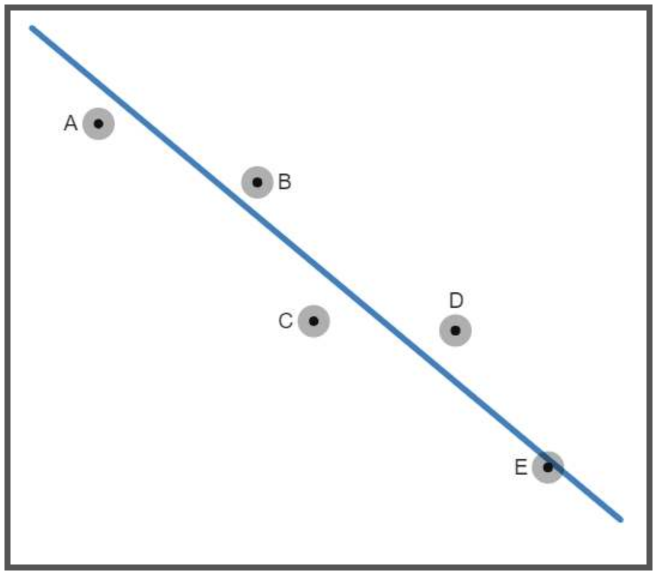
A discrete data set

To do this, we first use interpolation, that is, generate a polynomial that exactly matches the data set. In general, if we are using interpolation on n points, we can use a polynomial of degree $n-1$. In this case, we will use a polynomial of degree 4.



Example of Interpolation

Your intuition might say that this is an unusual path for temperature to follow, though maybe it makes sense for the drifting of a boat. Either way, let's next fit a line to the data set.



Example of Curve Fitting

In the sections that follow, we find techniques to generate these curves in the best possible way.

3.2: Polynomial Interpolation

Given a set of data, **polynomial interpolation** is a method of finding a polynomial function that fits a set of data points exactly. Though there are several methods for finding this polynomial, the polynomial itself is unique, which we will prove later.

3.2.1: Lagrange Polynomial

One of the most common ways to perform polynomial interpolation is by using the Lagrange polynomial. To motivate this method, we begin by constructing a polynomial that goes through 2 data points (x_0, y_0) and x_1, y_1. We use two equations from college algebra.

$$y - y_1 = m(x - x_1) \quad \text{and} \quad m = \frac{y_1 - y_0}{x_1 - x_0} \tag{3.2.1}$$

Combining these, we end up with:

$$y = \frac{y_1 - y_0}{x_1 - x_0}(x - x_1) + y_1. \tag{3.2.2}$$

Now, to derive a formula similar to that used for the Lagrange polynomial, we perform some algebra. We begin by swapping the terms $y_1 - y_0$ and $x - x_1$:

$$y = \frac{x - x_1}{x_1 - x_0}(y_1 - y_0) + y_1. \tag{3.2.3}$$

Distributing the fraction:

$$y = \frac{x - x_1}{x_1 - x_0}y_1 - \frac{x - x_1}{x_1 - x_0}y_0 + y_1. \tag{3.2.4}$$

Multiplying the right-most y_1 term by $\frac{x_1 - x_0}{x_1 - x_0} = 1$:

$$y = \frac{x - x_1}{x_1 - x_0}y_1 - \frac{x - x_1}{x_1 - x_0}y_0 + \frac{x_1 - x_0}{x_1 - x_0}y_1. \tag{3.2.5}$$

Combining to the y_1 terms:

$$y = -\frac{x - x_1}{x_1 - x_0}y_0 + \frac{x - x_0}{x_1 - x_0}y_1. \tag{3.2.6}$$

and finally flipping the denominator of the first term to get rid of the negative:

$$y = \frac{x - x_1}{x_0 - x_1}y_0 + \frac{x - x_0}{x_1 - x_0}y_1. \tag{3.2.7}$$

In the above, we can observe that, when $x = x_1$, it follows that the first term cancels out with a zero on top and the second term ends up as $1 \cdot y_1 = y_1$, as desired. Similarly, if $x = x_0$, then the first term ends up as $1 \cdot y_0 = y_0$ and the second term cancels out with a zero on top, causing the entire expression to be y_0, as desired.

For three data points $(x_0, y_0), (x_1, y_1), \text{and} (x_2, y_2)$, we can derive a polynomial that behaves similarly:

$$y = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}y_0 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}y_1 + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}y_2. \tag{3.2.8}$$

In the above, plugging in, for example, $x = x_1$ has the first and third terms cancelling out with zero and the middle term turning into $1 \cdot y_1$, as desired. We can follow this pattern to arrive at the full Lagrange polynomial.

THE LAGRANGE POLYNOMIAL

Given $n + 1$ data points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$ with $x_0 < x_1 < \cdots < x_n$, the Lagrange polynomial is the nth degree polynomial passing through each of these points and written as:

$$y = \frac{(x - x_1)(x - x_2)\cdots(x - x_n)}{(x_0 - x_1)(x_0 - x_2)\cdots(x_0 - x_n)}y_0 + \frac{(x - x_0)(x - x_2)\cdots(x - x_n)}{(x_1 - x_0)(x_1 - x_2)\cdots(x_1 - x_n)}y_1 + \cdots + \frac{(x - x_0)(x - x_1)\cdots(x - x_{n-1})}{(x_n - x_0)(x_n - x_1)\cdots(x_n - x_{n-1})}y_n \tag{3.2.9}$$

Equivalently, we can use product notation:

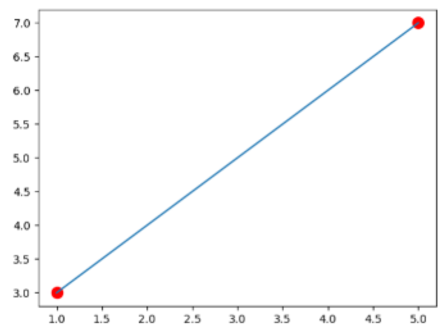
$$y = \sum_{i=0}^{n}\left(y_i \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}\right) \tag{3.2.10}$$

Note that in the above polynomial, each numerator is written such that, for $x = x_i$, each coefficient vanishes except for the coefficient to y_i, which evaluates to y_i. Thus the above polynomial passes through each of the desired data points and, as can be checked, is of degree n.

Now, let's work with Python. To do this, we use function blocks for the first time. We have used functions in the past in Python, such as sin(x) or cos(x). Here, we create our own function using the **def** keyword. Below, we define f(o), where o is the dynamic variable. At the end of a function block, we "return", using the **return** keyword, the result of the function calculation. In the code below, the variable o represents the variable x in the Lagrange Polynomial definition above. We do this since x is already used for our data points.

```python
import numpy as np
import matplotlib.pyplot as plot

#Data goes through the points (1,3) and (5,7)
x=[1,5]
y=[3,7]

#Set the number of data points
pts=len(x)-1
prange=np.linspace(x[0],x[pts],50)

plot.plot(x,y,marker='o', color='r', ls='',markersize=10)

def f(o):
    z=((o-x[1])/(x[0]-x[1]))*y[0] + ((o-x[0])/(x[1]-x[0]))*y[1]
    return z

plot.plot(prange,f(prange))
plot.show()
```

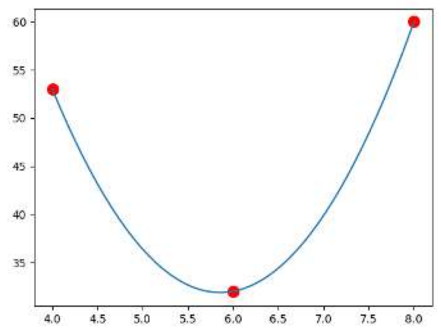
output:



Now, we do the same as the above except with 3 data points. Notice how the only changes are the data point lists and the function. Below, we again use the "\" symbol to let Python know to continue on the next line. This is done to make the polynomial easier to read.

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

#data goes through the points (4,1), (6,1) and (8,0)
x=[4,6,8]
y=[53,32,60]

n=len(x)-1
prange = np.linspace(x[0],x[n],50)

plot.plot(x,y,marker='o', color='r', ls='', markersize=10)

def f(o):
  z=\
  (((o-x[1])*(o-x[2]))/((x[0]-x[1])*(x[0]-x[2])))*y[0]+\
  (((o-x[0])*(o-x[2]))/((x[1]-x[0])*(x[1]-x[2])))*y[1]+\
  (((o-x[0])*(o-x[1]))/((x[2]-x[0])*(x[2]-x[1])))*y[2]
  return z

plot.plot(prange,f(prange))
plot.show()
```

output:



Using the product notation of Lagrange Polynomials, we can even come up with a program that accepts any number of data points.

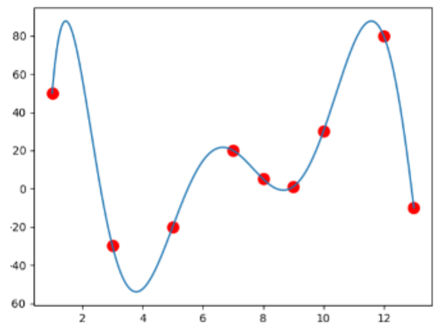
$$y=\sum_{i=0}^{n}\left(y_i\prod_{\substack{j=0\\ j\neq i}}^{n}\frac{x-x_j}{x_i-x_j}\right) \tag{3.2.11}$$

In the code below, i and j represent the i and j in the definition above.

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

x=[1,3,5,7,8,9,10,12,13]
y=[50,-30,-20,20,5,1,30,80,-10]

n=len(x)-1
prange = np.linspace(min(x),max(x),500)

plot.plot(x,y,marker='o', color='r', ls='', markersize=10)

def f(o):
  sum = 0
  for i in range(n+1):
    prod = y[i]
```

```
        for j in range(n+1):
            if i!= j:
                prod=prod*(o-x[j])/(x[i]-x[j])
        sum = sum + prod
    return sum

plot.plot(prange,f(prange))
plot.show()
```

output:



While Lagrange polynomials are among the easiest methods to understand intuitively and are efficient for calculating a specific $y(x)$, they fail when when attempting to either find an explicit formula $y=a_0+a_1x+\cdots+a_nx^n$ or when performing incremental interpolation, that is, adding data points after the initial interpolation is performed. For incremental interpolation, we would need to complete re-perform the entire evaluation.

3.2.2: Newton interpolation

Newton interpolation is an alternative to the Lagrange polynomial. Though it appears more cryptic, it allows for incremental interpolation and provides an efficient way to find an explicit formula $y=a_0+a_1x+\cdots+a_nx^n$.

Newton interpolation is all about finding coefficients and then using those coefficients to calculate subsequent coefficients. Since an important part of Newton interpolation is that it can be used for incremental interpolation, let's start with a single data point and show the calculations as we add points.

With one data point (x_0,y_0), the calculation is simple:

$$b_0=y_0 \tag{3.2.12}$$

and the polynomial is

$$y=b_0. \tag{3.2.13}$$

Let's add a new data point (x_1,y_1). The next coefficient b_1 is usually denoted by $[y_0,y_1]$:

$$b_1=[y_0,y_1]=\frac{y_1-y_0}{x_1-x_0} \tag{3.2.14}$$

and the polynomial is

$$y=b_0+b_1(x-x_0). \tag{3.2.15}$$

Notice how we did not have to re-calculate the entire polynomial, only the new coefficient to $(x-x_0)$.

With a third data point (x_2,y_2), we find the coefficient $b_2=[y_0,y_1,y_2]$:

$$b_2=[y_0,y_1,y_2]=\frac{[y_1,y_2]-[y_0,y_1]}{x_2-x_0}=\frac{\frac{y_2-y_1}{x_2-x_1}-\frac{y_1-y_0}{x_1-x_0}}{x_2-x_0} \tag{3.2.16}$$

with polynomial

$$y=b_0+b_1(x-x_0)+b_2(x-x_0)(x-x_1). \tag{3.2.17}$$

You may have noticed the recursive nature of the previous definitions. This continues for Newton interpolation in general.

NEWTON INTERPOLATION

Given $n+1$ data points $(x_0,y_0),(x_1,y_1),\ldots,(x_n,y_n)$ with $x_0<x_1<\cdots<x_n$, the Newton interpolation polynomial is the nth degree polynomial passing through each of these points and written as:

$$y=b_0+b_1(x-x_0)+b_2(x-x_0)(x-x_1)+\cdots+b_n(x-x_0)(x-x_1)\cdots(x-x_{n-1}) \tag{3.2.18}$$

where

$$b_0=y_0 \tag{3.2.19}$$
$$b_2=[y_0,y_1]=\frac{y_1-y_0}{x_1-x_0} \tag{3.2.20}$$
$$b_2=[y_0,y_1,y_2]=\frac{\frac{y_2-y_1}{x_2-x_1}-\frac{y_1-y_0}{x_1-x_0}}{x_2-x_0} \tag{3.2.21}$$
$$\vdots \tag{3.2.22}$$
$$b_n=[y_0,y_2,\ldots,y_n]=\frac{[y_1,y_2,\ldots,y_n]-[y_0,y_1,\ldots,y_{n-1}]}{x_n-x_0} \tag{3.2.23}$$

Let's begin by finding a polynomial with three data points using Newton's Method.

Example: Use Newton's method to calculating a polynomial through the points $(1,3)$, $(5,7)$ and $(8,0)$.

For this example, we will use a "tableau" to help organize our data.

x_0	y_0		
x_1	y_1	$[y_1, y_0]$	
x_2	y_2	$[y_2, y_1]$	$[y_0, y_1, y_2]$

We start by filling in our data points:

1	3		
5	7	$[y_1, y_0]$	
8	0	$[y_2, y_1]$	$[y_0, y_1, y_2]$

Then we calculate:

$$[y_1, y_0] = \frac{7-3}{5-1} = 1 \tag{3.2.24}$$

$$[y_2, y_1] = \frac{0-7}{8-5} = -\frac{7}{3} \tag{3.2.25}$$

and fill in this data:

1	3		
5	7	1	
8	0	$-\frac{7}{3}$	$[y_0, y_1, y_2]$

Now, we calculate the remaining item, using the previously-calculated terms:

$$[y_0, y_1, y_2] = \frac{[y_1, y_2] - [y_0, y_1]}{x_2 - x_0} = \frac{-\frac{7}{3} - 1}{8-1} = -\frac{10}{21} \tag{3.2.26}$$

1	3		
5	7	1	
8	0	$-\frac{7}{3}$	$-\frac{10}{21}$

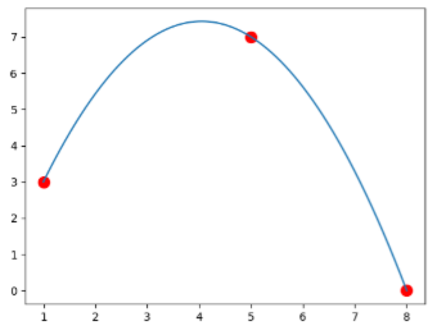
Note, now, that b_0, b_1, and b_2 are written on the top diagonal. Thus our final polynomial is:

$$y = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) = 3 + 1(x-1) - \frac{10}{21}(x-1)(x-5) \tag{3.2.27}$$

Now, let's program this for three data points to check our work!

```python
import numpy as np
import matplotlib.pyplot as plot

x=[1,5,8]
y=[3,7,0]

n=len(x)-1
prange = np.linspace(min(x),max(x),500)

plot.plot(x,y,marker='o', color='r', ls='', markersize=10)

def f(o):
  b0=y[0]
  b1=(y[1]-y[0])/(x[1]-x[0])
  b1p=(y[2]-y[1])/(x[2]-x[1])
  b2=(b1p-b1)/(x[2]-x[0])
  poly=b0+b1*(o-x[0])+b2*(o-x[0])*(o-x[1])
  return poly

plot.plot(prange,f(prange))
plot.show()
```

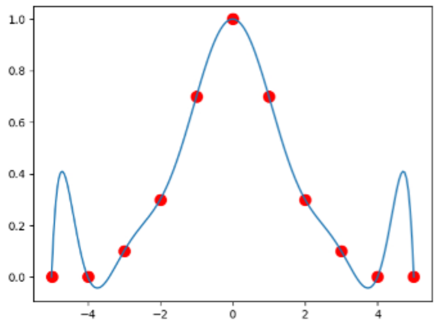
output:



A general program can be made by using a recursive function.

```python
import numpy as nm
import matplotlib.pyplot as plot

x=[-5,-4,-3,-2,-1,0,1,2,3,4,5]
y=[0,0,0.1,0.3,0.7,1,0.7,0.3,0.1,0,0]

plot.plot(x,y,marker='o', color='r', ls='',markersize=10)

def grad(a,b):
    if a==0: return y[b]
    return (grad(a-1,b)-grad(a-1,a-1))/(x[b]-x[a-1])


def f(o):
    yres=0
    for p in range(len(x)):
        prodres=grad(p,p)
        for q in range(p):
            prodres*=(o-x[q])
        yres+=prodres
    return yres
prange=nm.linspace(x[0],x[-1],200)
plot.plot(prange,f(prange))
plot.show()
```

```
output:
```

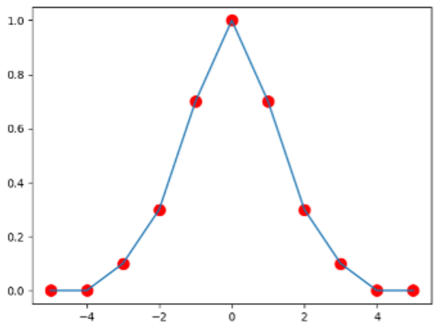


Note that above, we have something strange happening near the end points. This is known as **Runge's phenomenon** and mainly occurs at the endges of an interval when using polynomial interpolation with polynomials of high degree over a set of points that are equally spaced. Because of Runge's phenomenon, it is often the case that going to higher degrees does not always improve accuracy. To combat this, we can use a method such as cubic splines, which we discuss here.

3.2.3: Cubic Splines

A **spline** is a function defined piecewise by polynomials. Instead of having a single polynomial that goes through each data point, as we have done so far, we instead define several polynomials between each of the points. While defining several polynomials might take more work, it is usually preferred since it gives similar results while avoiding Runge's phenomenon.

A **linear spline** is created by simply drawing lines between each of the data points. Using our data points from earlier, we can create a reasonable approximation of our underlying function.



While simplistic, this approximation is likely better than that generated when using the same data points and Newton's method. This is due to the absence of Runge's phenomenon. To make the picture look even better, we can replace the lines by cubic functions. When doing so, we will end up with a number of cubic functions equal to one less than the number of data points - that is, one for every interval between the data points. We call this function $s(x)$, which can be defined as follows.

s(x) = \left\{ \begin{array}{lr} s_0(x)=a_0x^3+b_0x^2+c_0x+d_0 & \text{if } x_0\leq x\leq x_1\\ s_1(x)=a_1x^3+b_1x^2+c_1x+d_1 & \text{if } x_1\leq x\leq x_2\\ \vdots\\ s_{n-1}(x)=a_{n-1}x^3+b_{n-1}x^2+c_{n-1}x-

Index

Glossary

Sample Word 1	Sample Definition 1

Detailed Licensing

Overview

Title: Mathematical Computing with Python

Webpages: 24

Applicable Restrictions: Noncommercial

All licenses found:

- Undeclared: 62.5% (15 pages)
- CC BY-NC-SA 4.0: 37.5% (9 pages)

By Page

- Mathematical Computing with Python - *CC BY-NC-SA 4.0*
  - Front Matter - *Undeclared*
    - TitlePage - *Undeclared*
    - InfoPage - *Undeclared*
    - Table of Contents - *Undeclared*
    - Licensing - *Undeclared*
  - 1: Introduction to Python - *CC BY-NC-SA 4.0*
    - 1.1: Replit - *CC BY-NC-SA 4.0*
    - 1.2: Python Basics - *Undeclared*
    - 1.3: Plotting - *Undeclared*
    - 1.4: Logic and Loops - *Undeclared*
    - 1.5: Random Numbers - *Undeclared*
  - 2: Monte-Carlo Simulation - *CC BY-NC-SA 4.0*
    - 2.1: Introduction to Stochastic Simulation - *CC BY-NC-SA 4.0*
    - 2.2: Monte Carlo Simulation - *CC BY-NC-SA 4.0*
    - 2.3: Integration - *CC BY-NC-SA 4.0*
  - 3: Interpolation and Curve Fitting - *CC BY-NC-SA 4.0*
    - 3.1: Introduction to Interpolation and Curve Fitting - *CC BY-NC-SA 4.0*
    - 3.2: Polynomial Interpolation - *Undeclared*
  - Back Matter - *Undeclared*
    - Index - *Undeclared*
    - Glossary - *Undeclared*
    - Detailed Licensing - *Undeclared*
    - Detailed Licensing - *Undeclared*

Detailed Licensing

Overview

Title: Mathematical Computing with Python

Webpages: 24

Applicable Restrictions: Noncommercial

All licenses found:

- Undeclared: 62.5% (15 pages)
- CC BY-NC-SA 4.0: 37.5% (9 pages)

By Page

- Mathematical Computing with Python - *CC BY-NC-SA 4.0*
  - Front Matter - *Undeclared*
    - TitlePage - *Undeclared*
    - InfoPage - *Undeclared*
    - Table of Contents - *Undeclared*
    - Licensing - *Undeclared*
  - 1: Introduction to Python - *CC BY-NC-SA 4.0*
    - 1.1: Replit - *CC BY-NC-SA 4.0*
    - 1.2: Python Basics - *Undeclared*
    - 1.3: Plotting - *Undeclared*
    - 1.4: Logic and Loops - *Undeclared*
    - 1.5: Random Numbers - *Undeclared*
  - 2: Monte-Carlo Simulation - *CC BY-NC-SA 4.0*
    - 2.1: Introduction to Stochastic Simulation - *CC BY-NC-SA 4.0*
    - 2.2: Monte Carlo Simulation - *CC BY-NC-SA 4.0*
    - 2.3: Integration - *CC BY-NC-SA 4.0*
  - 3: Interpolation and Curve Fitting - *CC BY-NC-SA 4.0*
    - 3.1: Introduction to Interpolation and Curve Fitting - *CC BY-NC-SA 4.0*
    - 3.2: Polynomial Interpolation - *Undeclared*
  - Back Matter - *Undeclared*
    - Index - *Undeclared*
    - Glossary - *Undeclared*
    - Detailed Licensing - *Undeclared*
    - Detailed Licensing - *Undeclared*