MAS2806

Scientific Computation with Python:

Redacted

Assignment 2

November

2022

Contents

Question 1

Plots

The plots relating to particular problems and their respective code will be presented throughout the document when relevant. However, for the high-resolution images, the pages displaying images for each plot are as follows:

Python Scripts

Python scripts, including high-resolution plot images, created within this assignment can be found at the following GitHub repository:

https://github.com/j9mq/mas2806

Question 1a)

Consider the function

$$f(x) = x^5 - 2x^4 - 10x^3 + 18x^2 + 10x - 9$$

Complete the below code to set r to the roots of $f(x)$. ***(5 marks)***

Solution

We are looking to determine the roots of this polynomial using Python and the NumPy library.

I will be using the command **`np.roots()`** which takes in an array of integer values as its input and returns an integer array of the roots of $f(x)$.

```
import numpy as np

# Complete this line
p = [1,-2,-10,18,10,-9,]

r = np.roots(p)
```

If we were to print r, the console would return the array

```
print(p)
```

- `[1, -2, -10, 18, 10, -9]`

Question 1b)

In this part, use the Newton-Raphson code from the course material. (***10 marks***)

(i) Use the Newton-Raphson Method with an epsilon of $1 \cdot 10^{10}$, to give the root of $f(x)$ found with an initial guess, $x_0 = 1$

(ii) Use the Newton-Raphson Method with an epsilon of $1 \cdot 10^{10}$, to give the root of $f(x)$ found with an initial guess, $x_0 = 1.5$

Solution

Firstly, we will need to formulate a script to calculate the roots of a function, $f(x)$, using the Newton-Raphson method.

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

The function will take four input values:

- The function, $f(x)$
- The derivative of the function, $f'(x)$
- The initial value, x_1
- Epsilon, ε, value, a small integer of our choosing

We will formulate our functions $f(x)$ and $f'(x)$ as their own functions within Python. This allows us to easily call back to the functions currently as well as in our later plots and computations.

```
def f(x):
    return x**5 - 2 * x**4 - 10 * x**3 + 18 * x**2 + 10 * x - 9


def dfdx(x):
    "Derivative of f(x)"
    return 5 * x**4 - 8 * x**3 - 30 * x**2 + 36 * x + 10
```

Now, we can create our Newton-Raphson function:

```
""" Newton-Rapson Method """


def newraph(f, dfdx, x0, eps):

    x = x0
    n = 0

    while abs(f(x)) > eps:
        x = x - f(x) / dfdx(x)
        n += 1
    return x, n
```

We will save all three functions we have created thus far into a Python script named '**root_finding_methods.py**' which is saved in a single directory that all the scripts will be saved and executed from within this assignment.

This allows us to execute the Newton-Raphson function in other scripts to mitigate the repetition of defining $f(x)$, $f'(x)$, etc. as well as greatly improving the readability of our code.

Assigning variables r and n to our roots and iterations respectively allows us to call the Newton-Raphson function using the command:

```
import root_finding_methods as rm

x_0 = 1
eps = 1e-10

r, n = rm.newraph(
    lambda x: f(x),
    lambda x: dfdx(x),
    x_0,
    eps,
)
```

With the example of our initial guess $x_0 = 1.0$ and epsilon = $1 \cdot 10^{-10}$.

Next, let's utilise our newly defined Newton-Raphson function to determine the root of the function, $f(x)$, with varying initial guesses.

We will begin by setting $x_0 = 1.0$

```
import root_finding_methods as rm

r, n = rm.newraph(
    lambda x: x**5 - 2 * x**4 - 10 * x**3 + 18 * x**2 + 10 * x - 9,
    lambda x: 5 * x**4 - 8 * x**3 - 30 * x**2 + 36 * x + 10,
    1.5,
    1e-10,
)

print("Root found at {} after {} iterations using Newton Raphson.
\n".format(r,n))
```

Which produces the string:

```
Root found at 0.5425172220949075 after 4 iterations using Newton
Raphson.
```

We can repeat this with initial value x_0= 1.5 and observe:

```
Root found at 3.1226789593239204 after 6 iterations using Newton
Raphson.
```

Thus, to 5 decimal places, our roots of $f(x)$ using the Newton-Raphson method are

= **0.54252**

and

= **3.12268**

with initial values 1.0 and 1.5 respectively.

Note:

From this point on, any functions that will be called from the script '**root_finding_methods.py**' will be imported as follows:

```
import root_finding_methods as rm
```

Question 1c)

In handout 5, I introduced the **Secant Method**, but did not go into its details.

In your report, write your own description of the method, including comparing the features of the method with the Newton-Raphson and Bisection Methods. **(15 marks)**

(This part should be 250-500 words max and could also include illustrations)

Introduction

Within numerical analysis, The Secant Method is a recursive root finding procedure that allows the effective approximation of a root of a given function, f

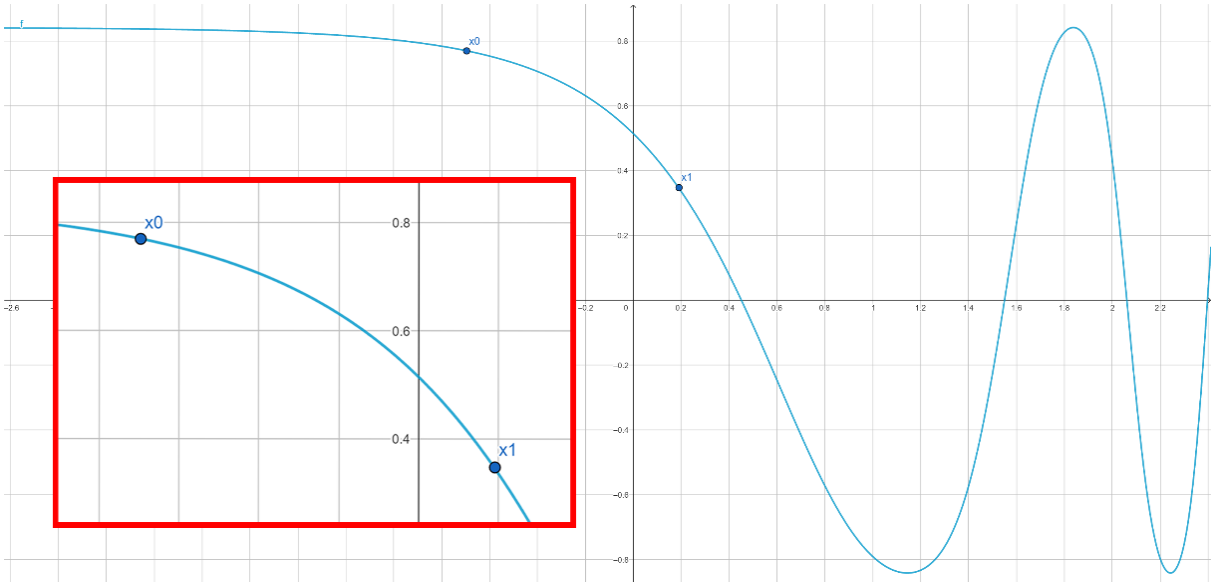
Unlike Newton's method of root convergence, The Secant Method does not require the derivative, f' as its input and can therefore be extremely effective in calculating the roots of complex expressions whose derivative would be difficult to compute.

This method to find the convergence of a root approximates the root using a secant line rather than a tangent and thus, requires two initial points on f .

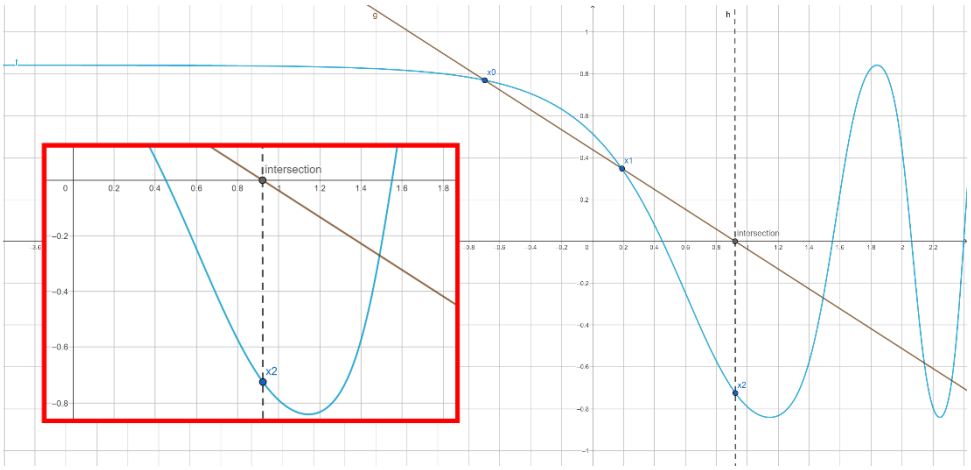
For example, we can visualise The Secant Method with the given f :

$$f(x) = \sin(\cos(e^x))$$

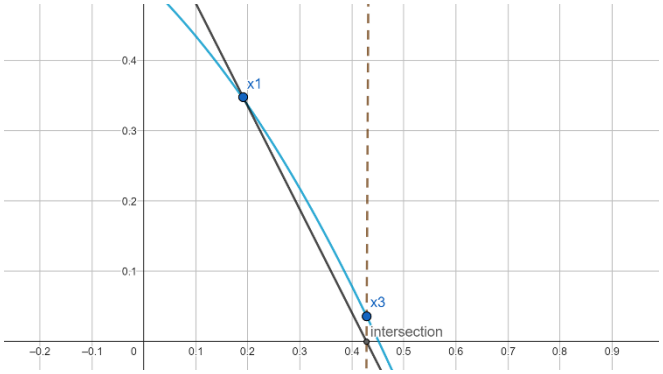
Which can be plotted as follows:



We will take two initial values on the function: x_0 and x_1 respectively. We intersect the two initial values with a secant line which will intersect the x axis as shown:



Plot an arbitrary point, x_2, on the function at the same x value at which the secant value intersects the x axis. We can disregard our initial x_0 value and now, consider the two points x_1 and x_2 as our region boundaries. Repeat this step again by which the region between our two values becomes increasingly smaller.



We will repeat this recurrence relation until the difference between the two values x_n and x_{n-1} is infinitesimally small.

If we were to continue decreasing the region, we would eventually divide by a difference of 0 which would arise to the mathematical error in which the result is undefined.

The recurrence relation of the Secant Method with initial values x_0 and x_1 is as follows:

$$x_2 = x_1 - f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)}$$

$$x_3 = x_2 - f(x_2)\frac{x_2 - x_1}{f(x_2) - f(x_1)}$$

And thus, by continuing this relation n times for an infinitesimally small difference. The formula for The Secant Method becomes:

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}$$

With our computations of The Secant Method in Python, we will ensure that our approximations for the root of f are not undefined by assigning a small, chosen value epsilon and computing the recurrance relation while the modulus, or absolute value of $f(n-1)$ is less than the chosen epsilon.

Question 1d)

Create a function for the secant method with the following specification (**15 marks**)

- Your function should be called **`sec_roots`**
- Your function should have input arguments **`sec_roots(f,x0,x1,eps)`** where
    - f is the function for which we are seeking a root
    - x_0 and x_1 are the initial guesses.
    - eps is the accuracy with which to seek a root
- Your function should return two values as a list in the following order:
    - the root found
    - the number of iterations to find the root

Solution

Using the previously discussed ideas regarding The Secant Method, we will begin by defining a function named **`sec_roots`** which will take in four inputs as described in the question.

We will begin by initialising an array of two initial values x_1 and x_2, a variable named iterations, and a variable 'a' which we will use in our while loop to ensure the final value in our approximation is taken rather than the value beforehand.

```
def sec_roots(f,x0,x1,eps):

    # Initialise an array of our starting guesses
    x = [x0, x1]
    iterations = 0
    a = 2
```

Next, we will begin our while loop with the condition that our code will stop when our last value x_n is no longer greater than our chosen, small value epsilon.

This ensures our solution will not be undefined as well as stopping our IDE from being in an infinite while loop – which is necessary since this is an approximation method.

The previously defined Secant Method can be applied under the conditions of the while loop to read:

```
while abs(f(x[a-1])) > eps:
        x_tmp = x[a-1] - (f(x[a-1]) * (x[a-1] - x[a-2])/
                          (f(x[a-1]) - f(x[a-2])))

        # Adds the next value to the array
        x.append(x_tmp)

        # Increase iteration counter and onto next value in the sequence
        iterations += 1
        a += 1
```

We will return the last value in the array created subtracted by one, due to Python's method of indexing arrays. Hence, our completed function becomes:

```
def sec_roots(f,x0,x1,eps):

    # Initialise an array of our starting guesses
    x = [x0, x1]
    iterations = 0
    a = 2

    while abs(f(x[a-1])) > eps:
        x_tmp = x[a-1] - (f(x[a-1]) * (x[a-1] - x[a-2])/
                          (f(x[a-1]) - f(x[a-2])))

        # Adds the next value to the array
        x.append(x_tmp)

        # Increase iteration counter and onto next value in the sequence
        iterations += 1
        a += 1

    lastvalue = len(x)-1

    return(x[lastvalue], iterations,)
```

Question 1e)

Use your Secant Method function from part **d)** in order to answer this part. ***(10 marks)***

(i) Use the Secant Method with an epsilon of $1 \cdot 10^{-10}$, to give the root found with initial guesses of [0.8, 1.2].

(ii) Use the Secant Method with an epsilon of $1 \cdot 10^{-10}$, to give the root found with initial guesses of [1.3, 1.7].

Solution

Using our **`sec_roots`** function, we can execute the command:

```
import root_finding_methods as rm

r, n = rm.sec_roots(
    lambda x: x**5 - 2 * x**4 - 10 * x**3 + 18 * x**2 + 10 * x - 9,
    1.3,
    1.7,
    1e-10,
)

print("Root found at {} after {} iterations using Secant.
\n".format(r,n))
```

Thus, to 5 decimal places, our roots of $f(x)$ using The Secant method are both

= **0.54252**

with initial values [0.8, 1.2] & [1.3, 1.7].

Question 1f)

Consider the following illustration, which demonstrates the sensitivity to initial conditions of the Newton-Raphson Method, for the roots of a similar problem:

$$f(x) = x^5 - 2x^4 - 10x^3 + 18x^2 + 10x - 9$$

Create your own plot like the above for both the Newton-Raphson and Secant Methods applied to the function $f(x)$ introduced at the start of the question.

Root finding scatter plot for function f(x)

Firstly, we will begin by plotting the graphs as they appear in the question.

For the above graph, we will need to plot:

- The function $f(x)$ as a line in matplotlib
- A scatter plot with markers indicating the roots of $f(x)$.

Let us refer to our functions $f(x)$, and $f'(x)$ from the **`root_finding_methods.py`** script

```
def f(x):
    return x**5 - 2 * x**4 - 10 * x**3 + 18 * x**2 + 10 * x - 9


def dfdx(x):
    "Derivative of f(x)"
    return 5 * x**4 - 8 * x**3 - 30 * x**2 + 36 * x
```

Next, we will initialise an NumPy array x which computes an array of 1000 values between -5 and 5.

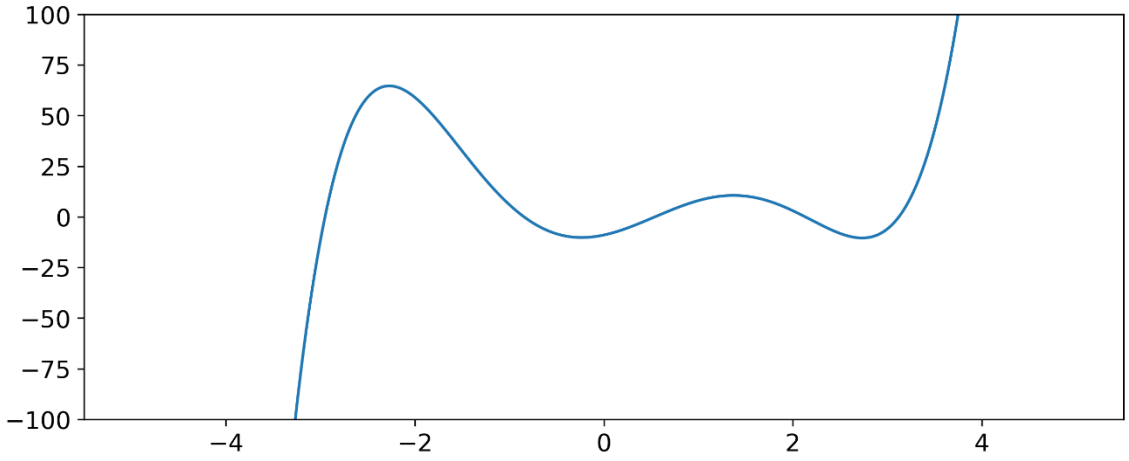
As well as this, we can create variables for our scatter diagram markers, marker colours, and coefficients of our function f which we can call p.

```
# Initialise x array for axis purposes
x = np.linspace(-5, 5, 1000)

# Initialise variables for scatter markers
markers = np.zeros(5,)
colors = [
    "orange",
    "blue",
    "purple",
    "green",
    "yellow",
]

# Finding roots of f(x)
p = np.roots(rm.p)
```

We can use **matplotlib** to plot $f(x)$ onto our axis with y boundaries: $-100 \leq y \leq 100$.

```
# Setting y axis to -100 <= y <= 100 and adjusting figure size
fig = plt.figure(figsize=[10, 4])
plt.ylim(
    [
        -100,
        100,
    ]
)

# Line plot
plt.plot(x, rm.f(x))
```

Which then produces the plot:



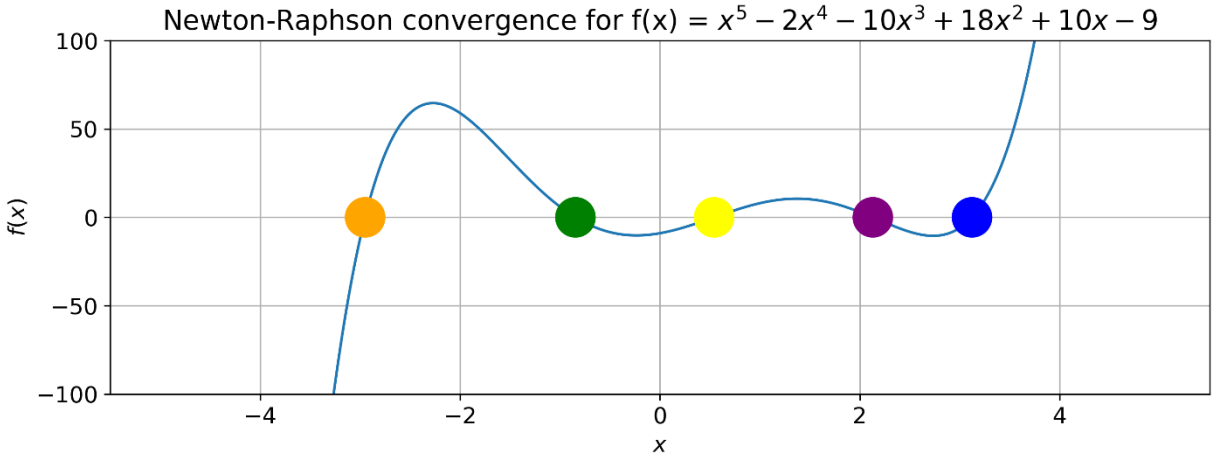
Using the command **plt.scatter**, we can add coloured markers to the roots of the function (which we found previously using NumPy's **np.roots** command.

Adding labels to the axis, a title, as well as some visual clarity commands intended to improve DPI scaling and add a grid to our plot, our final code for the root scatter plot then becomes:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import root_finding_methods as rm

# Initialise x array for axis purposes
x = np.linspace(-5, 5, 1000)

# Initialise variables for scatter markers
markers = np.zeros(5,)
```

```
colors = [
    "orange",
    "blue",
    "purple",
    "green",
    "yellow",
]

# Finding roots of f(x)
p = np.roots(rm.p)

# Setting y axis to -100 <= y <= 100 and adjusting figure size
fig = plt.figure(figsize=[10, 4])
plt.ylim(
    [
        -100,
        100,
    ]
)

# Line plot
plt.plot(x, rm.f(x))

# Scatter plot
plt.scatter(
    p,
    markers,
    c=colors,
    marker="o",
    s=500,
    zorder=3,
)

# Axis labelling
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
plt.title("Newton-Raphson convergence for f(x) = $x^5 - 2x^4 - 10x^3 +
18x^2 + 10x - 9$")

# Visual clarity
plt.rcParams.update({"font.size": 13})
plt.tight_layout()
plt.grid()

# Figure DPI
mpl.rcParams["figure.dpi"] = 300

plt.savefig(
    "root_graph.png",
)
plt.show()
```

The plot exported by this script is as follows:



Root convergence plot for function f(x) using the Newton-Raphson method

The method for plotting the root convergence using Newton's Method begins very similarly as we must initialise some essential variables.

Firstly, we will use our previously defined x array of 1000 linear values.

```
eps = 1e-10
```

We will define a small value, epsilon, to be used in the previously defined Newton's function.

Let us create a function named **`new_raph_graph()`** with zero inputs. Initialise a value u = 0 and an empty array 'a', which we will use in our while loop.

```
def new_raph_graph():
    """Function to plot a graph using Newton's of root finding"""

    u = 0
    a = []

    while u <= len(x) - 1:  # To stay in bound for axis 0 with size n-1
        r, n = rm.newraph(
            rm.f,
            rm.dfdx,
            x[u],
            eps,
        )
```

As Python indexing begins with the value 0 rather than 1, we must subtract 1 from the length of our NumPy array x and we will write this as our main condition for the while loop.

Within the while loop, we will run the previously imported Newton's method and input the function f, its derivative f', the value u at each point in the x array, and the epsilon value.

We will append the initially empty array 'a' at each completion of the while loop by iterations n. Hence, when the while loop has completed, we can return a final iterations integer

```
    while u <= len(x) - 1:  # To stay in bound for axis 0 with size n-1
        r, n = rm.newraph(
            rm.f,
            rm.dfdx,
            x[u],
            eps,
        )
        a.append(n)
```

To draw the coloured areas on the plot, we will create numerous control flow statements which utilise the root found, r, the array 'p' which was created using **`np.roots()`**, and the array of colours.

If the modulus of our root found using Newton's method subtracted to the actual root is less than our defined value epsilon, then the **`plt.vlines()`** function will draw single vertical lines of a particular colour corresponding to the markers on the previous graph.

We are using epsilon here as the root, r will not be *exactly* the value found using **`np.roots()`** due to data representational and storage issues within Python. Thus, we must use a very small value, ε, and an inequality operator rather than the condition which r is exactly equal to a value within p.

Thus, our control flow statements are as follows:

```
# Drawing individual lines with distinguished colours
        if abs(r - p[0]) <= eps:
            plt.vlines(
                x[u], ymin=0, ymax=n, colors=colors[0],
            )
        elif abs(r - p[1]) <= eps:
            plt.vlines(
                x[u], ymin=0, ymax=n, colors=colors[1],
            )
```

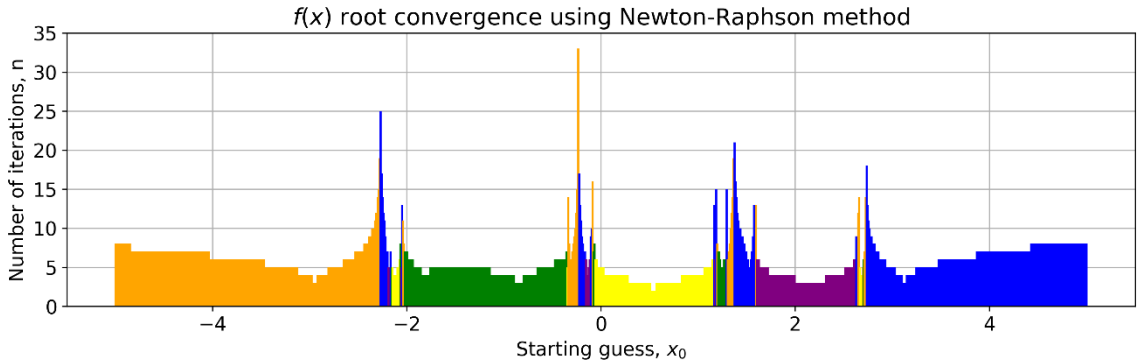
And this repeats up to the final value in the array p[4].

We will then break the control flow once there are no more values to be found and increment 'u' by 1 The function returns 'a' which can be used to plot the starting guess 'x' against iterations 'n'. Executing the previous commands for visual clarity, figure size, with y limit $0 \leq y \leq 35$.

We can plot our root convergence graph using the command:

```
# Plot
plt.plot(
    x, new_raph_graph(), alpha=0,
)
plt.savefig("newton_graph.png",)

plt.show()
```

Therefore, the plot exported by the script is as follows:



Root convergence plot for function f(x) using The Secant Method

Similarly, to plotting our graph based on the Newton-Raphson method, our Secant graph will utilise the commands which we have previously defined.

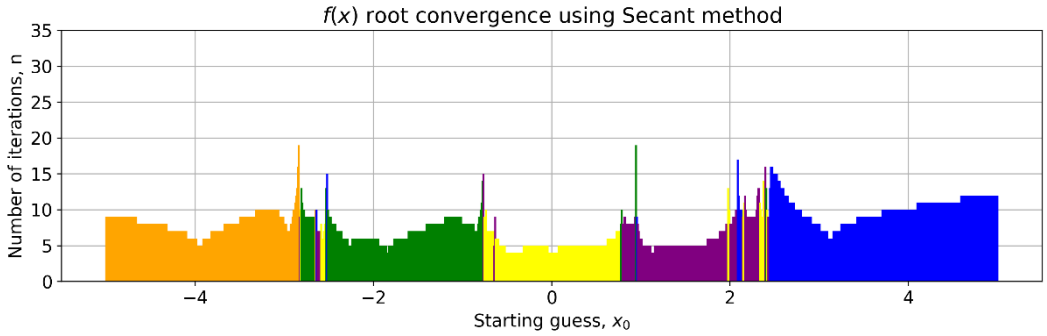
Rather than initialising the function f and its derivative in our while loop, we will instead utilise our function f and our two initial values, which are the parameters required for the Secant Method

```
    while u <= len(x) - 1:  # To stay in bound for axis 0 with size 399
        r, n = rm.sec_roots(
            rm.f,
            x[u],
            x[u] + 1,
            eps,
        )
        a.append(n)
```

As well as this we must replace our Newton-Raphson plot command to our Secant graph function, **`secant_graph()`**

```
# Plot
plt.plot(
    x, secant_graph(), alpha=0,
)
plt.savefig("secant_graph.png",)

plt.show()
```

Hence, by running the current script with the same limits, our plot for The Secant Method becomes:



Question 1g)

You should find that, in some cases, neither the Newton-Raphson or Secant Method find the root closest to an initial guess.

Consider the following hybrid method which uses a bound like the Bisection Method, but updates the boundaries using the Secant Method, rather than using the value of the function at the midpoint:

- Start with lower and upper bounds $[x_d, x_u]$.
- Calculate $x_s = x_u - f(x_u)\dfrac{x_u - x_d}{f(x_u) - f(x_d)}$ (the Secant Method formula).
- Update the bounds $[x_d, x_u]$ as follows:
  - Set $x_u = x_s$ if $f(x_u)$ and $f(x_s)$ have the same sign; or
  - Set $x_d = x_s$ if $f(x_d)$ and $f(x_s)$ have the same sign

Code up this method to find an example starting guess x_0, for which the Newton-Raphson Method and Secant Method don't find the closest root, but this method does. Comment on this in your report.

Solution:

As the question entails, we will be using a similar function to the Bisection method whereby we begin with two initial values and, as the function progresses, our bound between our two values becomes increasingly smaller until we are left with just a single remaining value.

We will begin with our previously used Numpy array, x, which takes 1000 linearly spaces values between -5 and 5. As well as this, we will define our small value, epsilon, which we will use later.

```
# Initialise x array
x = np.linspace(
    -5,
    5,
    1000,
)

eps = 1e-10
```

We will use our previously defined p variable which is derived using the **`np.roots()`** function on the coefficients of f.

To define our hybrid method, we will call a function method3 which takes four inputs of our function, a lower and upper bound, as well as our epsilon value.

```
def method3(f,xd,xu,eps):
```

We will then begin to transfer the 'pseudo-code' given in the question into Python. By transferring our mathematical expressions into Python, we must first let our script know when to stop, to prevent an inevitable infinite loop from occurring.

As previously executed, we will let our method3 function run when the difference from our upper and lower initial values is less than the defined epsilon.

Then, by writing the pseudo formulae utilising The Secant Method, our code then becomes:

```
def method3(f, xd, xu, eps):

    while abs(xu - xd) > eps:

        xs = xu - (rm.f(xu) * ((xu - xd) / (rm.f(xu) - rm.f(xd))))

        if (rm.f(xu) * rm.f(xs)) > 0:
            xu = xs
        elif (rm.f(xd) * rm.f(xs)) > 0:
            xd = xs
        else:
            break

    return xs
```

Where our *xs* is the single value returned from the Secant Method.

By assigning a constant, say **m3**, we can test our function with example bounds such as 1.5 and 2.0.

```
m3 = method3(rm.f, 1.5, 2.0, eps)
print("Method 3 export = {}".format(m3))
```

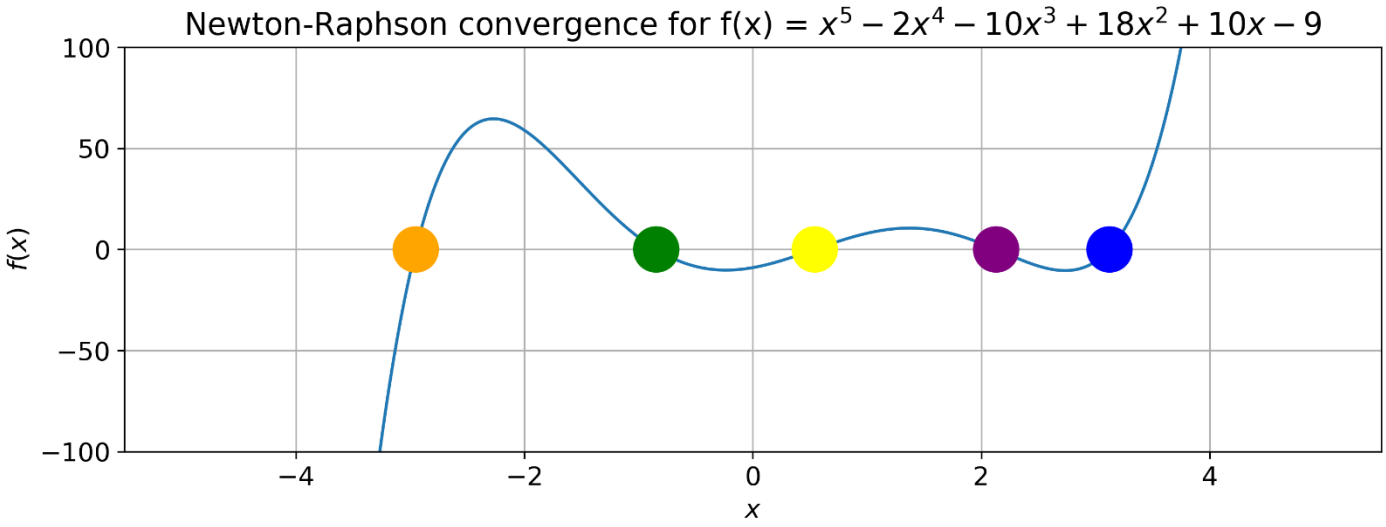
Which then returns the string:

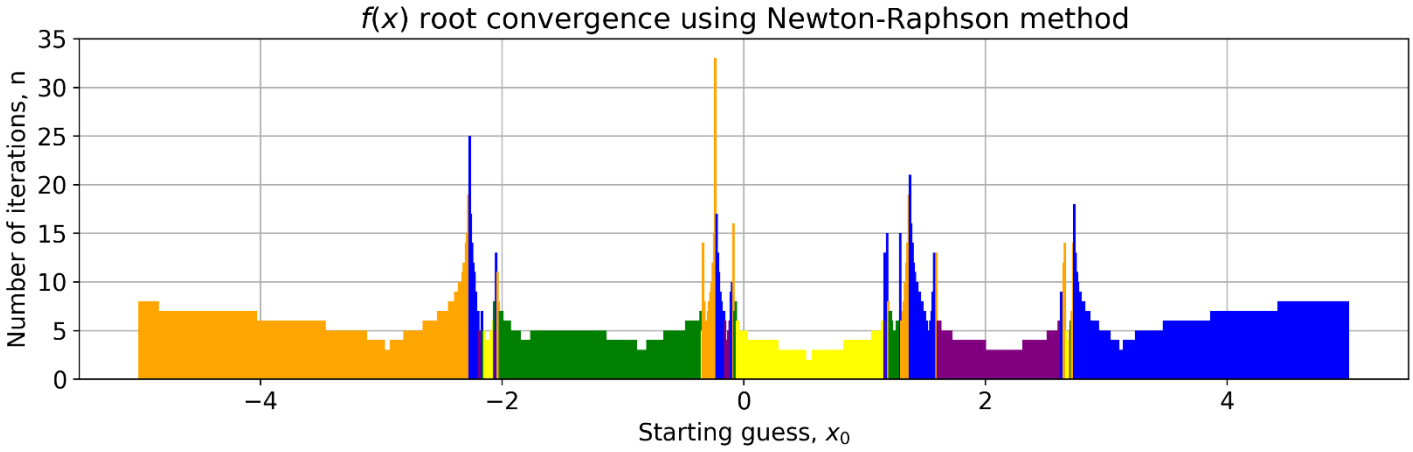
```
Method 3 export = 2.207792207792208
```

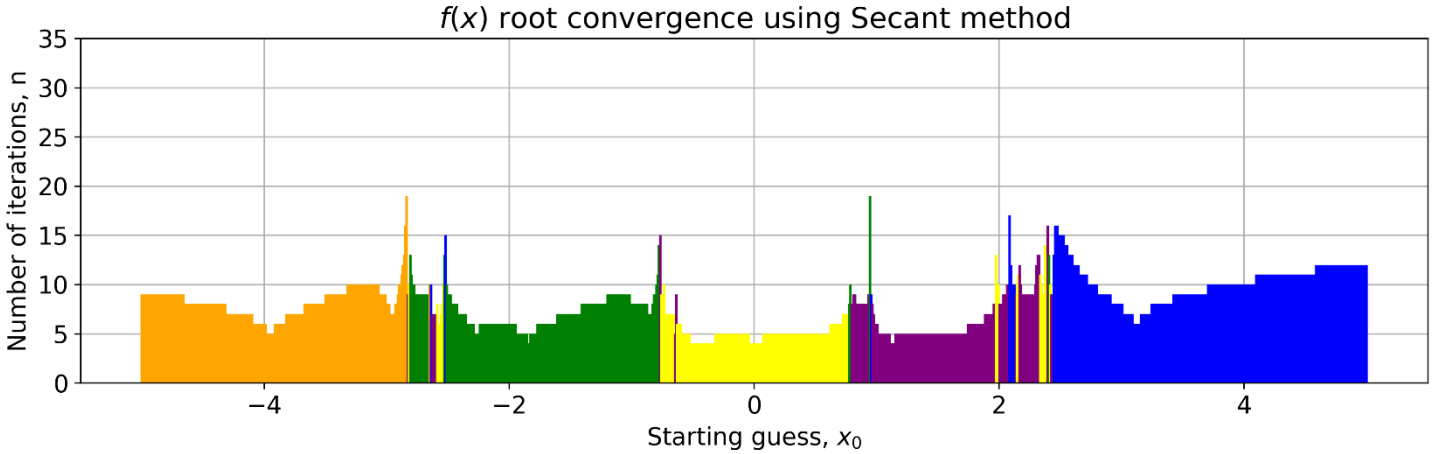
Which we can observe is not the value which we would expect from executing identical starting values using only The Secant Method.

```
Root found at 2.130171291480174 after 5 iterations using Secant.
```

Plots:


Newton-Raphson convergence for f(x) = $x^5 - 2x^4 - 10x^3 + 18x^2 + 10x - 9$
100
50
0
−50
−100
f(x)
−4
−2
0
2
4
x


f(x) root convergence using Newton-Raphson method
35
30
25
20
15
10
5
0
Number of iterations, n
−4
−2
0
2
4
Starting guess, x_0


f(x) root convergence using Secant method
35
30
25
20
15
10
5
0
Number of iterations, n
−4
−2
0
2
4
Starting guess, x_0