CONTRIBUTION

The group talked about the grades that will be given should be the same for each member of the group. Each group member participated and contribute ideas to finish this user manual before the due date. Each member would like to say thank you for their fellow group members.

Inside the Module

```
### Brute force algorithm(f(x)=0)
def f_of_x(f,roots,tol,i, epochs=100):

    x_roots=[] # list of roots
    n_roots= roots # number of roots needed to find
    incre = i #increments
    h = tol #tolerance is the starting guess

    for epoch in range(epochs): # the list of iteration that will be using
     if np.isclose(f(h),0): # applying current h or the tolerance in the equation and the app
        x_roots.insert(len(x_roots), h)
        end_epochs = epoch
        if len(x_roots) == n_roots:
          break # once the root is found it will stop and print the root
     h+=incre # the change of value in h wherein if the roots did not find it will going to l

    return x_roots, end_epochs # returning the value of the roots and the iteration or the ep
```

```
### brute force algorithm (in terms of x)
def in_terms_of_x(eq,tol,epochs=100):

    funcs = eq # equation to be solved
    x_roots=[] # list of roots
    n_roots = len(funcs) # How many roots needed to find according to the length of the equat
    # epochs= begin_epochs # number of iteration
    h = tol # tolerance or the guess to adjust

    for func in funcs:
      x = 0 # initial value or initial guess
      for epoch in range(epochs): # the list of iteration that will be using
        x_prime = func(x)
        if np.allclose(x, x_prime,h):
          x_roots.insert(len(x_roots),x_prime)
          break # once the root is found it will stop and print the root
        x = x_prime
    return x_roots, epochs # returning the value of the roots and the iteration or the epochs
```

```
### newton-raphson method
```

```
def newt_raphson(func_eq,prime_eq, inits, epochs=100):

  f = func_eq # first equation
  f_prime = prime_eq # second equation
  # epochs= max_iter # number of iteration
  x_inits = inits # guess of the roots in range
  roots = [] # list of roots

  for x_init in x_inits:
    x = x_init
    for epoch in range(epochs):
      x_prime = x - (f(x)/f_prime(x))
      if np.allclose(x, x_prime):
        roots.append(x)
        break # once the root is found it will stop and print the root
      x = x_prime
  return roots, epochs # returning the value of the roots and the iteration or the epochs
```

Inside also of the module there is an import numpy and import matplotlib the module is given the name first_two_method for this activity to avoid confusion to the last three method.

On the package

The package was name numeth_yon for the numeth that stands for the course numerical method while the yon is the group number.

Explaining how to use your package and module with examples.

```
'''
Import the Numpy package to the IDLE (import numpy as np), and from numeth_yon package import
it is needed for the first_two_method to be next to the function that was inside in the first
the, f_of_x, in_terms_of_x and the newt_raphson, that wiil seen below:
'''

import numpy as np
from numeth_yon import first_two_method
```

```
'''
For the user to use f_of_x function, it is needed to provide the roots, the iteration is alre
the estimated guess, and a number of increase must be provided. The function created that has
is to find the roots of the given equation.
is to find the roots of the given equation.
'''

sample1 = lambda x: x**2+x-2
roots, epochs = first_two_method.f_of_x(sample1,2,-10,1) # the first_two_method is the module
print("The root is: {}, found at epoch {}".format(roots,epochs+1))
# Output: The root is: [-2, 1], found at epoch 12
```

```
'''
In this method of using Brute Force Algorithm in terms of x, the user must have the equation
number of iteration was already set to 100 as default value  and tolerance to adjust the gues
'''

sample2 = lambda x: 2-x**2
sample3 = lambda x: np.sqrt(2-x)

funcs = [sample2, sample3]
roots, epochs = first_two_method.in_terms_of_x(funcs,1e-05) # the first_two_method is the mod
print("The root is {} found after {} epochs".format(roots,epochs))
# Output: The root is [-2, 1.00000172977337] found after 100 epochs
```

```
'''
To use the newt_raphson, the user must provide an equation, the derivative of the equation,
the number of repetitions was set to default value of 100, and the range of searching value f
The function of the function, is to find the roots of the given equation.
'''

g = lambda x: 2*x**2 - 5*x + 3
g_prime = lambda x: 4*x-5

# the first_two_method is the module that is next to the function inside of the module
roots, epochs = first_two_method.newt_raphson(g,g_prime, np.arange(0,5))
x_roots = np.round(roots,3)
x_roots = np.unique(x_roots)
# Output: The root is [1.  1.5] found after 100 epochs
```

To further understand please refer to the PDF version.

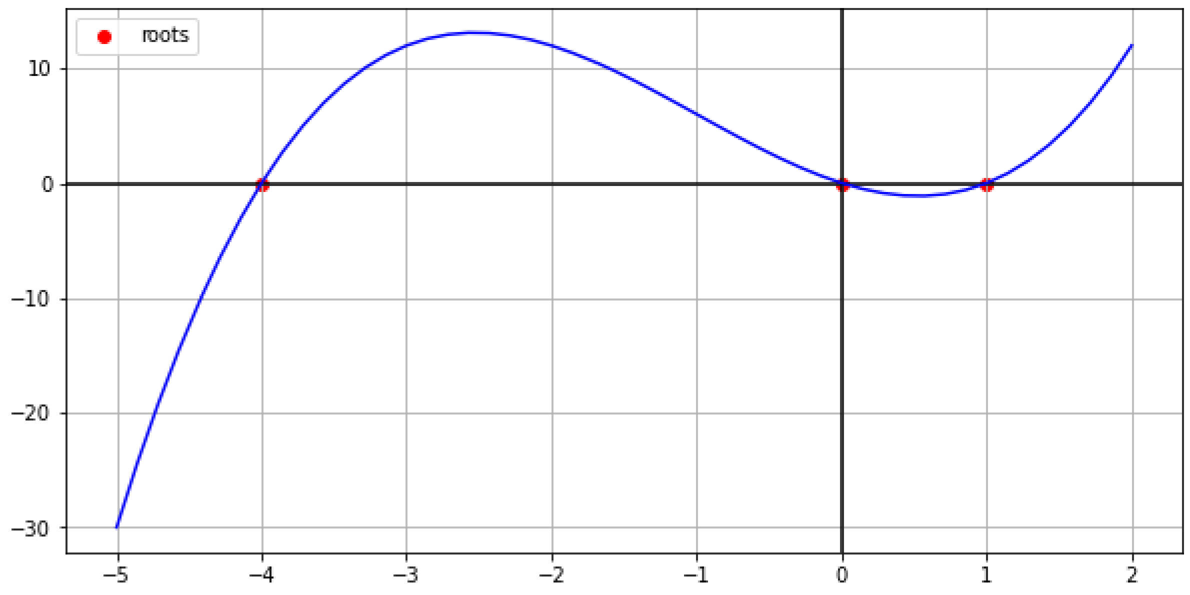
Activity 2.1

1. Identify **two more polynomials** preferably **orders higher than 2** and **two transcendental functions**. Write them in **LaTex**.
2. Plot their graphs you may choose your own set of pre-images.
3. Manually solve for their roots and plot them along the graph of the equation.

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

$$f(x) = x^3 + 3x^2 - 4x$$

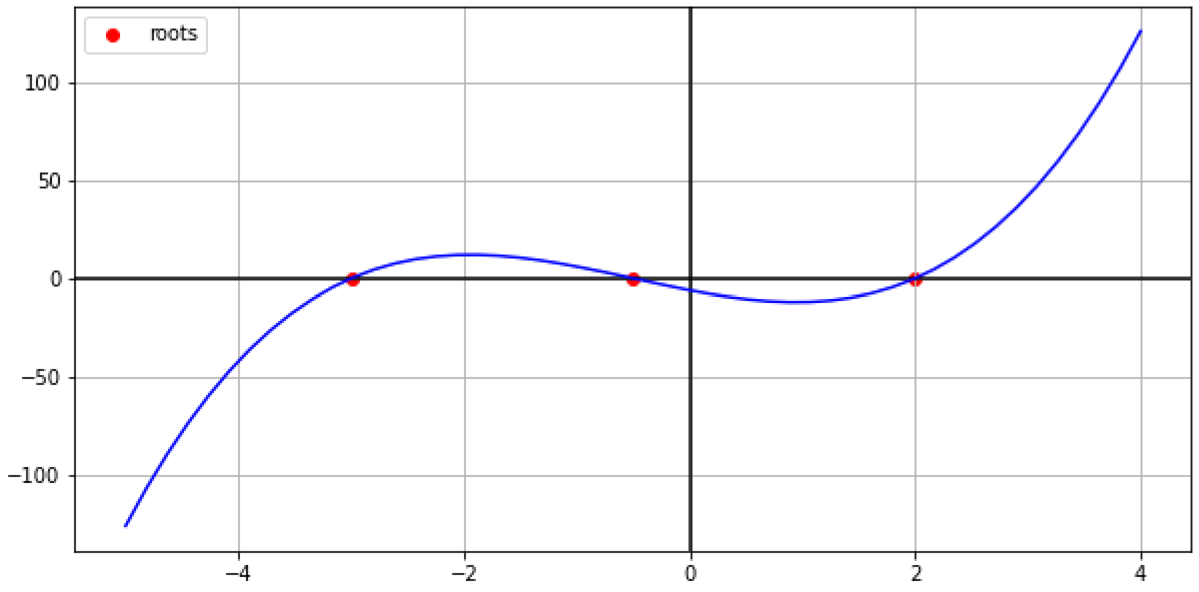
```
##Input the function on the define f(x)
def f(x):
    return x**3+3*x**2-4*x

x0, x1,x2 = -4, 0, 1 ## Roots of the function f(x)

## Plotting the roots in a graph
X = np.linspace(-5,2,dtype=float)
Y = f(X)

## Creating the grid of the graph

plt.figure(figsize=(10,5))
plt.axhline(y=0,color='black')
plt.axvline(x=0,color='black')
plt.grid()

## Plotting the roots in the graph
plt.plot(X,Y,color='blue')
plt.scatter([x0,x1,x2],[0,0,0], c='red', label='roots')

plt.legend()
plt.show()
```



$$f(x) = 2x^3 - 3x^2 - 11x - 6$$

```
def f(x):
    return 2*x**3+3*x**2-11*x-6
```

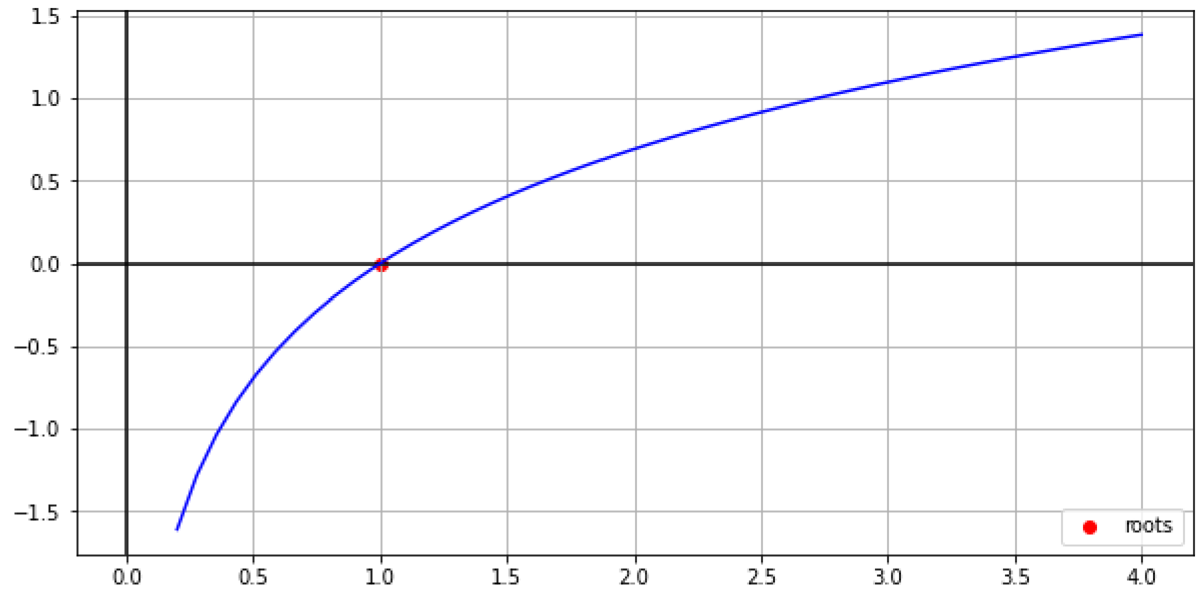
```
x0, x1,x2 = -3, -0.5, 2 ## Roots of the function f(x)

## Plotting the roots in a graph
X = np.linspace(-5,4,dtype=float)
Y = f(X)

## Creating the grid of the graph

plt.figure(figsize=(10,5))
plt.axhline(y=0,color='black')
plt.axvline(x=0,color='black')
plt.grid()

## Plotting the roots in the graph
plt.plot(X,Y,color='blue')
plt.scatter([x0,x1,x2],[0,0,0], c='red', label='roots')

plt.legend()
plt.show()
```



$$f(x) = log(x)$$

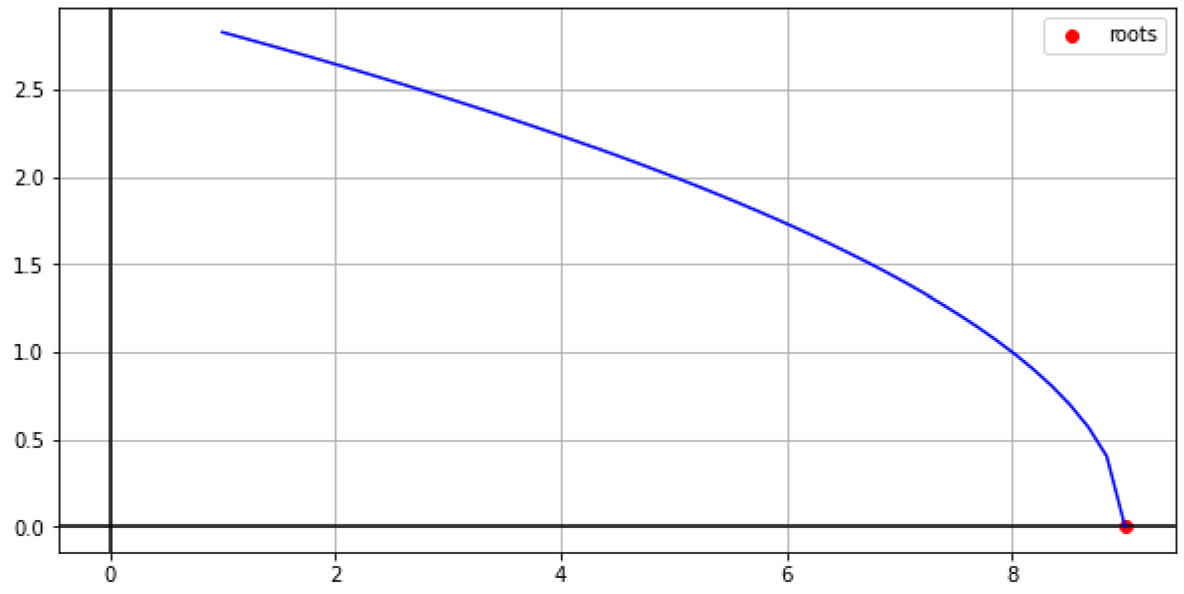
```
def f(x):
    return np.log(x)

x0 = 1 ## Roots of the function f(x)

## Plotting the roots in a graph
X = np.linspace(0.2,4,dtype=float)
Y = f(X)

## Creating the grid of the graph
```

```
plt.figure(figsize=(10,5))
plt.axhline(y=0,color='black')
plt.axvline(x=0,color='black')
plt.grid()

## Plotting the roots in the graph
plt.plot(X,Y,color='blue')
plt.scatter([x0],[0], c='red', label='roots')

plt.legend()
plt.show()
```



$$f(x) = \sqrt{9 - x}$$

```
def f(x):
    return np.sqrt(9-x)

x0 = 9 # Roots of the function f(x)

## Plotting the roots in a graph
X = np.linspace(1,9,dtype=float)
Y = f(X)

## Creating the grid of the graph

plt.figure(figsize=(10,5))
plt.axhline(y=0,color='black')
plt.axvline(x=0,color='black')
plt.grid()

## Plotting the roots in the graph
plt.plot(X,Y,color='blue')
plt.scatter([x0],[0], c='red', label='roots')
plt.scatter([x0],[0], c='red', label='roots')

plt.legend()
plt.show()
```


● roots