How To Use Package and Module

Finding roots of Polynomial

Simple Iteration and Newton-Rhapson Method

submitted by:

Group 4 : Ichi

Aquiro, Freddielyn E.

Canoza, Cherrylyn

Joaquin, Christopher Marc

Note:

This notebook will be used to understand on how to use the module and implement it into package. The modules contains functions on how to find a single root and n number of roots. The user will provide a equation in getting the results.

Step 1: Import the package into the jupyter notebook or google colab notebook.

note: if you're using google colab notebook to import modules, first you have to open your google notebook then after that in the upper left corner you will see an folder type image then click on it. Then go to your documents find the module you want to import. Then hold and drag your module into the folder image in google colab.

```
import numeth_simp_newton as s_num
# importing the module
```

Step 2: Define an equation that you desire.

Sample Equation

First given for Simple Iteration Method:

$$F(x) = x^2 - 5x + 4$$

Second given for Newton-Rhapson Method

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

Step 3: Choose any of the two methods for finding its roots.

- Simple Iteration Method
- Newton-Rhapson Method

Step 4: If you already choose one, provide the required parameters that you wish in finding its root/s.

Simple iteration (Brute Force)

This method is used as the easiest way to compute the eqaution and it will utlilize iterations or looping statements. Brute force are rarely used because its method are straight-forward in solving equations which it rely on the sheer computing power that tries every possibile answers than advanced techniques in improving its efficiency. [1]

```
# The user asked to input its equation which f denotes as its defined equation and h denot
#for Single root
def f(x): return x**2-5*x+4
```

```
#for Single root
s_num.b_force(f,-5)
```

```
54
40
28
18
10
4
0
The root is: [1], found at epoch 6
```

```
#for N number of roots
s_num.brute_nforce(f,-4)
```

```
40
28
18
10
4
```

```
0
-2
-2
0
4
10
18
28
40
54
70
88
108
130
154
180
208
238
270
304
340
378
418
460
504
550
598
648
700
754
810
868
928
990
1054
1120
1188
1258
1330
1404
1480
1558
1638
1720
1804
1890
1978
2068
2160
2254
2350
2448
2548
2650
```

Newton-Rhapson Method

This method is another way in roots finding that uses linear approximation which is similiar to brute force but it it uses an updated functions. [2]

This method require the user to give a equation to solve,a initial guess, and the derivative of given equation f is equal to the defined equation,-5 is the initial guess input by the user, and x_p is equal to the declared derivative equation of f

```
# Given f denotes as the defined equation anf f_prime its the declared derivative equatior
def f(x): return 3*x**3-5*x**2-4*x+4
def f_prime(x): return 9*x**2-10*x-4
```

```
#  Single root
s_num.newt_R(f,f_prime)
```

```
The root is: 0.6666666433828111, found at epoch 4
```

```
#for N number of roots
s_num.newt_N(f,f_prime)
```

```
Epoch:  2
x prime is:  0.6666666433828111
x final:  0.6666666433828111
Epoch:  3
x prime is:  0.6666666666666666
roots:  [0.6666666433828111, 0.6666666433828111]
Iteration Number:  2
x is:  2
Epoch:  0
x prime is:  2.0
roots:  [0.6666666433828111, 0.6666666433828111, 2]
Iteration Number:  3
x is:  3
Epoch:  0
x prime is:  2.404255319148936
x final:  2.404255319148936
Epoch:  1
x prime is:  2.105117829566335
x final:  2.105117829566335
Epoch:  2
x prime is:  2.010154451310177
x final:  2.010154451310177
Epoch:  3
x prime is:  2.000109804807441
x final:  2.000109804807441
Epoch:  4
x prime is:  2.0000000130594087
x final:  2.0000000130594087
Epoch:  5
x prime is:  2.0
roots:  [0.6666666433828111, 0.6666666433828111, 2, 2.0000000130594087]
Iteration Number:  4
x is:  4
```

```
x is:  4
Epoch:  0
x prime is:  3.0
x final:  3.0
Epoch:  1
x prime is:  2.404255319148936
x final:  2.404255319148936
Epoch:  2
x prime is:  2.105117829566335
x final:  2.105117829566335
Epoch:  3
x prime is:  2.010154451310177
x final:  2.010154451310177
Epoch:  4
x prime is:  2.000109804807441
x final:  2.000109804807441
Epoch:  5
x prime is:  2.0000000130594087
x final:  2.0000000130594087
Epoch:  6
x prime is:  2.0
roots:  [0.6666666433828111, 0.6666666433828111, 2, 2.0000000130594087, 2.00000001305
np_roots before round:  [0.6666666433828111, 0.6666666433828111, 2, 2.000000013059408
np_roots after round:  [0.667 0.667 2.    2.    2.   ]
np_roots after sorting to unique:  [0.667 2.   ]
array([0.667, 2.   ])
```

Activity 2.1 included in this laboratory

Finding Roots of Polynomials

A polynomial function is a function that can be expressed in the form of a polynomial.[3] Every value given in its function has a corresponding degree, which so-called *order*. The target of this program is to find the roots even there are tons of given value in a polynomial function. Thus, the programmer will give two examples in with different orders:
First given:

$$F(x) = 5x^4 + 10x^3 - 75x^2$$

Second given:

$$F(x) = -3x^5 - 12x^4 - 12x^3$$

The formula that the progammer will provide is factorization.

1. Get the GCF.
2. Factor out.

3. Transposition

Now, if the user will solve manually the first given example, the roots are x = -5 and x = 3. With this function, the user can check and plot if the user has already found out the roots.

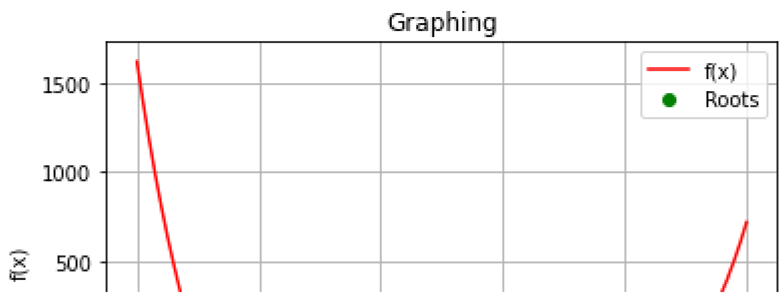
```
#Function for finding roots in Polynomials.
import numpy as np
from numpy.polynomial import Polynomial as npoly
import matplotlib.pyplot as plt

def f(x):
  for i in range(len(x)): ###Getting the list given by the user.
    x[i] = float(x[i]) #For calling purposes
  p=npoly(x) ###Finding coefficients with the given roots.
  xzeros=p.roots() ### return the roots of a polynomial with coefficients given.
  for i in range (len(xzeros)): ###Getting all the roots.
    print("x=",xzeros[i]) ###Printing roots.
###Graphing
  x=np.linspace(xzeros[0]-1,xzeros[-1]+1,100) ###for locating the value in x axis
  y=p(x) ###for locating the value in y axis
  fig, ax=plt.subplots() ###Creating Figures
  ax.plot(x,y,'r', label= 'f(x)') ###For plotting
  ax.plot(xzeros,p(xzeros),'go', label='Roots') ###For plotting
  ax.legend(loc='best') ### for legend
  ax.grid()
  plt.xlabel('x') ###Label in x axis
  plt.ylabel('f(x)') ###Label in y axis
  plt.title('Graphing') ###Title of the graph
  plt.show() #Output
```

```
z = np.array([0,0,-75,10,5])
```

The program puts every values of polynomials inside of array. The programmer is still get the other non-present values of order like the normal number without variable and the 1st order value even there is no present value. The programmer has a pattern of putting the polynomial values in array, first is the least order towards to the greatest order. When the programmer used their built in function, this is now the result.

```
f(z)
```

```
x= -5.0
x= 0.0
x= 0.0
x= 3.0
```

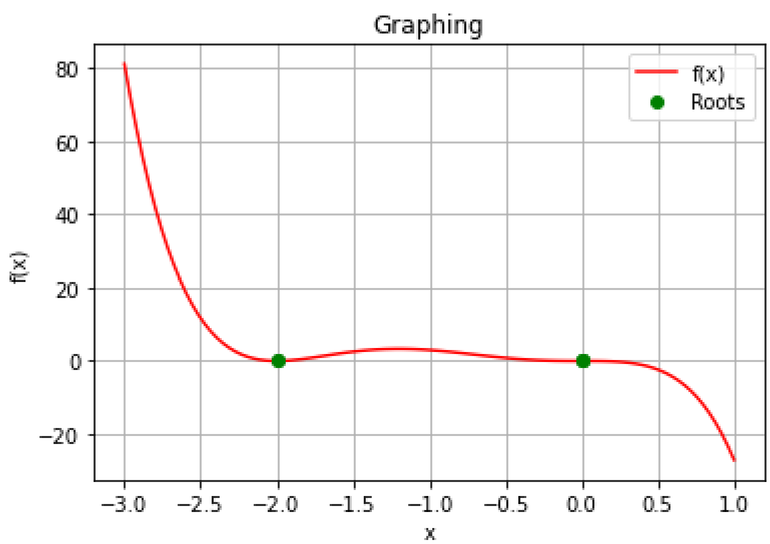


Therefore, the manual computation of the user are the same value as what the progammer did in this program. Next, is the second given, the programmer do the same process. He puts all the values of polynomial inside of the array.

```
1 o = np.array([0,0,0,-12,-12,-3])
```

```
1 f(o)
```

```
x= -1.9999999999999998
x= -1.9999999999999998
x= 0.0
x= 0.0
x= 0.0
```



References:

[1] freeCodeCamp.org (2018), about "*Brute Force Algorithms Explained*" Online

[2] Mathematical Python(2019), about "*Newton's Method*" Online

[3] BYJU'S.(2021), about "*Polynomial Function Definition*" Online