Roots of Equations: User Manual

submitted by:

Aquiro, Freddielyn E.

Canoza, Cherrylyn

Joaquin, Christopher Marc

Note:

This user manual will be used to understand on how to use the module and import the 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.

Methods

1. Simple Iterations Method (Brute Force)
2. Newton-Rhapson Method
3. Bisection Method
4. Regula Falsi Method
5. Secant Method

Definitions

Finding Roots of Polynomials

A polynomial function is a function that can be expressed in the form of a polynomial.[1] 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

Transcendental function

Transcendental function is a term that refers to the ability to transcend one' A function that can't be expressed as a finite combination of the algebraic operations of addition, subtraction, multiplication, division, raising to a power, and extracting a root in mathematics. The functions log x, sin x, cos x, ex, and any functions containing them are examples. In algebraic terms, such functions can only be expressed as infinite sequence. [2]

First given:

$$f(x) = sin(x)xcos(x)$$

$$g(x) = cos(x)cos(x)xsin(x)$$

Second given:

$$f(x) = 2sinx$$

$$g(x) = 2cosx$$

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. [3]

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. [4]

Bisection Method

The bisection method is used to find the roots of a polynomial equation. It separates the interval and subdivides the interval in which the root of the equation lies. The principle behind this method is the intermediate theorem for continuous functions. It works by narrowing the gap between the positive and negative intervals until it closes in on the correct answer. [5]

Regula Falsi Method

It is also known as *method of false position*, it is a numerical method for solving an equation in one unknown. It is quite similar to bisection method algorithm and is one of the oldest approaches. It was developed because the bisection method converges at a fairly slow speed. In simple terms, the method is the trial and error technique of using test ("false") values for the variable and then adjusting the test value according to the outcome. [6]

Secant Method

The secant method is very similar to the bisection method except instead of dividing each interval by choosing the midpoint the secant method divides each interval by the secant line connecting the endpoints. [7]

For activity 2.1 included in the laboratory

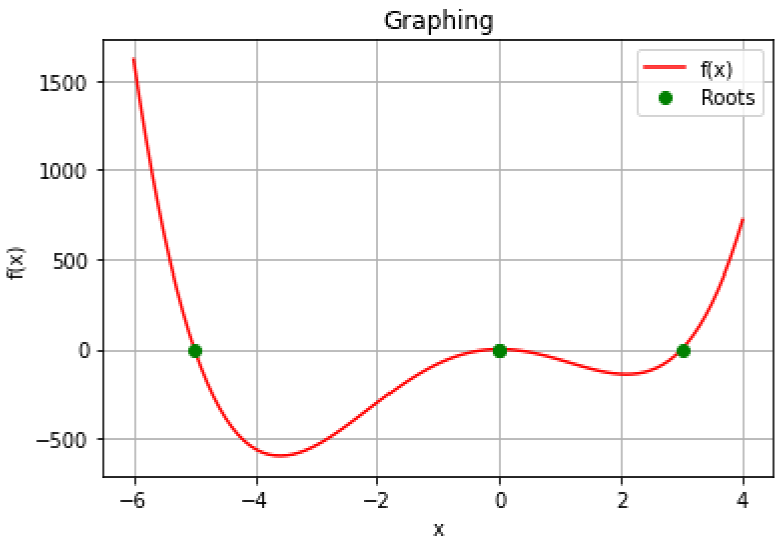
```
###For the actitivity 2.1 Finding The Roots of Polynomials and Transcedental
#Function for finding roots in Polynomials.
import numpy as np
from numpy.polynomial import Polynomial as npoly
import matplotlib.pyplot as plt

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.title('Graphing') ###Title of the graph
plt.show() #Output
```

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

```
f(z)
```

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




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

```
f(o)
```

```
x= -1.9999999999999998
```

Explanation

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.

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


```
from scipy.optimize import brentq
from scipy import optimize

plt.figure(figsize=(12,8))
plt.style.use('bmh')
x = np.linspace(0,20,num=50)
plt.ylim(-10,10)
plt.plot(x,np.sin(np.cos(x)),label='$sin(cos(X))$')
plt.legend(loc=1)
plt.savefig('fun.png',dpi=300,bbox_inches = 'tight')

def f(x):
    # The function
    return np.sin(np.cos(x))

def roots(N):

    roots = np.zeros(N)
    margin = 1e-8

    for i in range(1,N):
        left = (3*i - 1)*np.pi/2
        right = (3*i + 1)*np.pi/2
        roots[i] = brentq(f, left + margin, -right - margin, rtol = 1e-14)
    return roots

result = roots(1000)

left = 200
right = 300

def f(x): return np.sin(np.cos(x))
def g(x): return -np.cos(np.cos(x))*np.sin(x)
epochs = 10
n_roots = 3
```

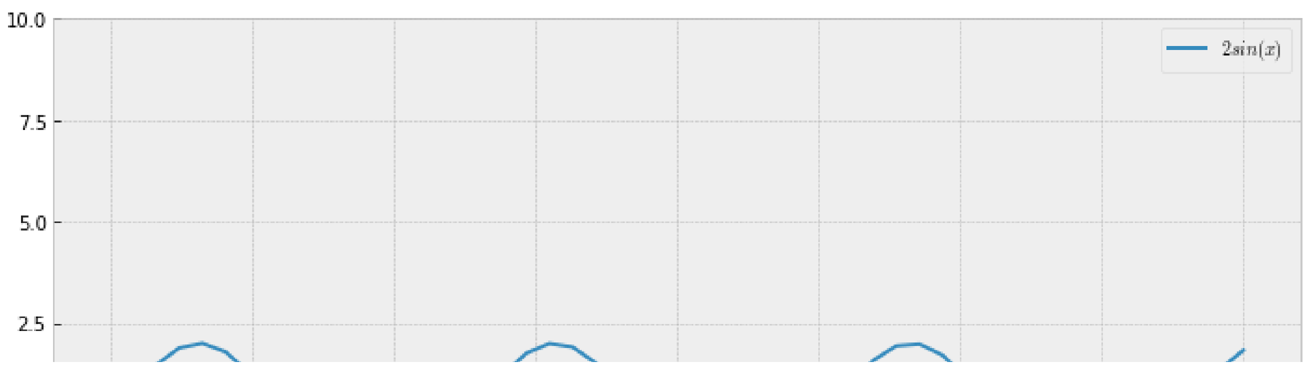
```
n_roots = 3
x_roots = []
g_elem = []
end_epoch = 0
h = 0

for epoch in range(epochs):
  print("g(h) when h = ", h, " : ", g(h))
  print("Is 0 equal to ", g(h))
  if np.allclose(0,g(h),1e-03):
    x_roots.append(h)
    g_elem.append(g(h))
    print("Current x_roots: ", x_roots)
    print("Current g_elem: ", g_elem)
    end_epoch = epoch
    print("end_epoch: ", end_epoch)
    print("xroot length: ", len(x_roots), " ; n_roots: ", n_roots)
    if len(x_roots)==n_roots:
      break
  h+=1e-4
  print("h: ", h)
print(f"The root is: {x_roots}, found at epoch {end_epoch+1}")
```

```
g(h) when h =  0  :  -0.0
Is 0 equal to  -0.0
Current x_roots:  [0]
Current g_elem:  [-0.0]
end_epoch:  0
xroot length:  1  ; n_roots:  3
h:  0.0001
g(h) when h =  0.0001  :  -5.4030230917499074e-05
Is 0 equal to  -5.4030230917499074e-05
h:  0.0002
g(h) when h =  0.0002  :  -0.00010806046381910875
Is 0 equal to  -0.00010806046381910875
h:  0.00030000000000000003
g(h) when h =  0.00030000000000000003  :  -0.00016209070068893942
Is 0 equal to  -0.00016209070068893942
h:  0.0004
g(h) when h =  0.0004  :  -0.00021612094351110107
Is 0 equal to  -0.00021612094351110107
h:  0.0005
g(h) when h =  0.0005  :  -0.0002701511942697031
Is 0 equal to  -0.0002701511942697031
h:  0.0006000000000000001
g(h) when h =  0.0006000000000000001  :  -0.0003241814549488542
Is 0 equal to  -0.0003241814549488542
h:  0.0007000000000000001
g(h) when h =  0.0007000000000000001  :  -0.0003782117275326616
Is 0 equal to  -0.0003782117275326616
h:  0.0008000000000000001
g(h) when h =  0.0008000000000000001  :  -0.0004322420140052321
Is 0 equal to  -0.0004322420140052321
h:  0.0009000000000000002
g(h) when h =  0.0009000000000000002  :  -0.0004862723163506703
Is 0 equal to  -0.0004862723163506703
```

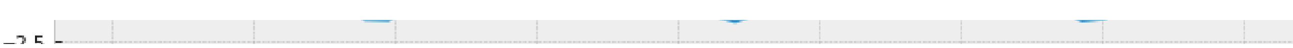
```
plt.figure(figsize=(12,8))
plt.style.use('bmh')
x = np.linspace(0,20,num=50)
plt.ylim(-10,10)
plt.plot(x,2*np.sin(x),label='$2sin(x)$')
plt.legend(loc=1)
plt.savefig('fun.png',dpi=300,bbox_inches = 'tight')

def f(x):
    # The function
    return np.sin(np.cos(x))

def roots(N):

    roots = np.zeros(N)
    margin = 1e-8

    for i in range(1,N):
        left = (3*i - 1)*np.pi/2
        right = (3*i + 1)*np.pi/2
        roots[i] = brentq(f, left + margin, -right - margin, rtol = 1e-14)
```

```
    return roots

result = roots(1000)

left = 200
right = 300


def f(x): return 2*np.sin(x)
def g(x): return 2*np.cos(x)
epochs = 10
n_roots = 3
x_roots = []
g_elem = []
end_epoch = 0
h = 0

for epoch in range(epochs):
  print("g(h) when h = ", h, " : ", g(h))
  print("Is 0 equal to ", g(h))
  if np.allclose(0,g(h),1e-03):
    x_roots.append(h)
    g_elem.append(g(h))
    print("Current x_roots: ", x_roots)
    print("Current g_elem: ", g_elem)
    end_epoch = epoch
    print("end_epoch: ", end_epoch)
    print("xroot length: ", len(x_roots), " ; n_roots: ", n_roots)
    if len(x_roots)==n_roots:
      break
  h+=1e-4
  print("h: ", h)
print(f"The root is: {x_roots}, found at epoch {end_epoch+1}")
```

```
g(h) when h =  0  :  2.0
Is 0 equal to  2.0
h:  0.0001
g(h) when h =  0.0001  :  1.99999999
Is 0 equal to  1.99999999
h:  0.0002
g(h) when h =  0.0002  :  1.9999999600000002
Is 0 equal to  1.9999999600000002
h:  0.00030000000000000003
g(h) when h =  0.00030000000000000003  :  1.9999999100000008
Is 0 equal to  1.9999999100000008
h:  0.0004
g(h) when h =  0.0004  :  1.999999840000002
Is 0 equal to  1.999999840000002
h:  0.0005
g(h) when h =  0.0005  :  1.9999997500000053
Is 0 equal to  1.9999997500000053
h:  0.0006000000000000001
g(h) when h =  0.0006000000000000001  :  1.9999996400000108
Is 0 equal to  1.9999996400000108
h:  0.0007000000000000001
g(h) when h =  0.0007000000000000001  :  1.99999951000002
Is 0 equal to  1.99999951000002
h:  0.0008000000000000001
g(h) when h =  0.0008000000000000001  :  1.999999360000034
Is 0 equal to  1.999999360000034
h:  0.0009000000000000002
g(h) when h =  0.0009000000000000002  :  1.9999991900000547
Is 0 equal to  1.9999991900000547
h:  0.0010000000000000002
The root is: [], found at epoch 1
```



Module



```
# Simple iteration (Brute Force) single root
#For single root.
def b_force(f,h):
  epochs =50
  x_roots = []
  for epoch in range(epochs):
    x_guess = f(h)
    print(x_guess)
    if x_guess == 0:
```

```
      x_roots.append(h)
      break
    else:
         h+=1
  return print(f"The root is: {x_roots}, found at epoch {epoch}")
```

```
# Finding n number of roots.
def brute_nforce(f,h,epochs = 10): #default
  n_roots = 3
  x_roots = []
  end_epoch = 0
  for epoch in range(epochs):
    print(f(h))
    if np.allclose(0,f(h)):
      x_roots.append(h)
      end_epoch = epoch
      if len(x_roots)==n_roots:
        break
    h+=1
  return print(f"The root is: {x_roots}, found at epoch {end_epoch+1}")
```

```
#Newton- Rhapson Method single root
## Single Root
def newt_R(f,f_prime,epochs):
  x = 0
  root = 0
  for epoch in range(epochs):
    x_prime = x - (f(x)/f_prime(x))
    if np.allclose(x, x_prime):
      root = x
      break
    x = x_prime
  return print(f"The root is: {root}, found at epoch {epoch}")
```

```
# Findng n number of  roots
def num_newt(f,f_prime,epochs):
  x_inits = np.arange(0,5)
  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
      x = x_prime
  np_roots = np.round(roots,3)
  print("np_roots before round: ",roots)
  np_roots = np.round(roots,3)
  print("np_roots after round: ", np_roots)
```

```
  print("np_roots after round: ", np_roots)
  np_roots = np.unique(np_roots)
  print("np_roots after sorting to unique: ", np_roots)
  return np_roots
```

```
def bisect_n(func, iv1, iv2, epochs , tol):
    x_inits = np.arange(-10,10)
    arr_len = len(x_inits) - 1
    end_bisect = 0
    roots = []
    y1, y2 = func(iv1), func(iv2)
    end_bisect = 0
    if np.sign(y1) == np.sign(y2):
          print("Root cannot be found in the given interval")
    else:
          for iter in range(arr_len):
            iv1 = x_inits[iter]
            iv2 = x_inits[iter+1]
            for bisect in range(epochs):
                midp = np.mean([iv1,iv2])
                y_mid = func(midp)
                y1 = func(iv1)
                if np.allclose(0, y1,tol):
                    roots.append(iv1)
                    end_bisect = bisect
                    break
                if np.sign(y1) != np.sign(y_mid): #root for first-half interval
                  iv2 = midp
                else: #root for second-half interval
                  iv1 = midp
    # n_roots = roots
    # n_roots = np.unique(np.round(n_roots,5))
    return roots, end_bisect
```

```
## Regula Falsi Method
## Finding multiple roots
def rfalsi_n(f,a,b,tol):
  x_inits = np.arange(-10,10)
  arr_len = len(x_inits) - 1
  y1, y2 = f(a), f(b)
  root = None
  n_roots = []
  pos = 0
  if np.allclose(0,y1):
    root = a
    n_roots.append(a)

  elif np.allclose(0,y2):
    root = b
    n_roots.append(b)

```

```
  elif np.sign(y1) == np.sign(y2):
    print("No root here")
  else:
    for iter in range(arr_len):
      a = x_inits[iter]
      b = x_inits[iter+1]
      for pos in range(0,100):
        c = b - (f(b)*(b-a))/(f(b)-f(a)) ##false root
        if np.allclose(0,f(c),tol):
          root = c
          n_roots.append(c)
        if np.sign(f(a)) != np.sign(f(c)):
         b,y2 = c,f(c)
        else:
          a,y1 = c,f(c)

  roots = n_roots
  n_roots = np.unique(np.round(n_roots,6))
  return roots, n_roots, pos
```

```
#secant Method
def sec_n(f,a,b,epochs):
  x_inits = np.arange(-10,10)
  arr_len = len(x_inits) - 1
  root = None
  n_roots = []
  end_epoch = 0
  for iter in range(arr_len):
    a = x_inits[iter]
    b = x_inits[iter+1]
    for epoch in range(epochs):
      c = b - (f(b)*(b-a))/(f(b)-f(a))
      if np.allclose(b,c):
        root = c
        n_roots.append(root)
        end_epoch = epoch
        break
      else:
        a,b = b,c
  roots = n_roots
  n_roots = np.unique(np.round(n_roots,6))
  return roots, n_roots, end_epoch
```

Step 1: Import the module *(module name: numeth_simp_n_newt_method)* 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_roe_package_modified as pyg
```

Step 2: The user will choose any methods to use.

1. Simple Iteration Method
2. Newton-Rhapson Method
3. Bisection Method
4. Regula Falsi Method
5. Secant Method

Step 3: The user will input the equations he/she desire.

Note: the following codes are importing the package.

Simple Iteration

Parameters

for single root

1. f : equation of the single root.
2. h : the tolerance.

return

1. x_root : the number of roots.
2. epoch : the number of iterations where the root is located.

For multiple roots

1. f : equation of the multiple roots.
2. h : the tolerance
3. epochs: user's desired number of roots.

return

1. x_roots : calculated list of roots from the equation
2. end_epoch+1: texpected number of iterations for roots.

Step 4: Displaying the output of the package.

```
# Sample equation inputted to be solve
def f(x): return x**2-5*x+4
```

```
#single root
pyg.b_force(f,-5)
```

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

```
pyg.brute_nforce(f,-4)
```

```
1258
1330
1404
1480
1558
1638
1720
1804
1890
1978
2068
2160
2254
2350
2448
2548
2650
2754
2860
2968
3078
3190
3304
3420
3538
3658
3780
3904
```

```
4030
4158
4288
4420
4554
4690
4828
4968
5110
5254
5400
5548
5698
5850
6004
6160
6318
6478
6640
6804
6970
7138
7308
7480
7654
7830
8008
8188
8370
8554
The root is: [1, 4], found at epoch 9
```

Newton- Rhapson Method

Parameters

For single root

1. f : is the equation given by the user.
2. f_prime : derivative of the equation.
3. epochs : user desired number of roots.

return

1. root : calculated list of roots from the equation
2. epoch : expected number of iterations for roots.

For multiple roots

1. f : is the equation given by the user.
2. f_prime : derivative of the equation.

3. epochs : user desired number of roots.
4. x_inits : values returning to its interval

return

1. np_roots : calculated list of roots from the equation
2.

```
## For single root
def f(x): return 90*x**3+200.153*x**2-173.12*x-10
def f_prime(x): return 270*x**2+400.306*x-173.12
```

```
pyg.newt_R(f,f_prime,epochs =100)
```

```
The root is: -0.05442285534619281, found at epoch 3
```

```
pyg.num_newt(f,f_prime,epochs =100)
```

```
np_roots before round:  [-0.05442285534619281, 0.7092145930473319, 0.7092145813706895,
np_roots after round:  [-0.054  0.709  0.709  0.709  0.709]
np_roots after sorting to unique:  [-0.054  0.709]
array([-0.054,  0.709])
```

Bisection Method

Parameters

1. f : is the equation given by the user.
2. iv1 : first interval
3. iv2 : second interval
4. nm_roots : user expected no. of roots.
5. tol : tolerance
6. epochs: no. of iterations per roots.

return

1. roots : calculated list of roots from the equation
2. end_bisect : expected value of roots where its located.

```
#sample equation
def func(x): return 90*x**3+200.153*x**2-173.12*x-10
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -3.5,iv2 = 2,epochs= 100, tol= 1e-6)
```

```
print("the roots found are : " ,n_roots, " found at epochs", end_epoch)
```

```
the roots found are :  [-2.878714 -0.054423  0.709215]  found at epochs 0
```

These are the test cases for polynomial

```
#test 1
def func(x): return x**3 -6*x**2 - 9*x + 54
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -3.5,iv2 = 2,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
the roots found are :  [-3.  3.  6.]  found at epochs 0
```

```
#test 2
def func(x): return (x**2-2*x-2)*(3*x**2+10*x-8)
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -1 ,iv2 = 4,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
the roots found are :  [-4.       -0.732051  0.666667  2.732051]  found at epochs 33
```

```
#test 3
def func(x): return x**3-x**2-x+1
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -1 ,iv2 = 4,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
the roots found are :  [-1.        0.999939  1.      ]  found at epochs 0
```

```
#test4
def func(x): return(x**2-6*x+5)*(x**2+3*x-18)
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -1 ,iv2 = 3,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
the roots found are :  [-6.  1.  3.  5.]  found at epochs 0
```

```
#test5
def func(x): return x**5-2*x**4+3.6*x**3-0.51*x**2-1.33*x-0.2
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -1.1 ,iv2 = 2, epochs= 100, tol= 1e-6)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
The number of roots are not in sorted and unique: <function roots at 0x7f40f3ced170>
the roots found are :  [-0.36549  0.87554]  found at epochs 29
```

```
#test6
def func(x): return -5*x**5-2*x**4+x**2+1.33*x-0.4,
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -2 ,iv2 = 6,epochs= 100, tol= 1e-6)
print("The number of roots are not in sorted and unique:", roots)
```

```
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)

```

```
The number of roots are not in sorted and unique: <function roots at 0x7f40f3ced170>
the roots found are :  [-0.800747  0.261054]  found at epochs 27
```

```
#7
def func(x): return 2*x**4-4*x**3- 0.78*x*2-1.785*x+1.4
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -1 ,iv2 = 2,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
the roots found are :  [0.369393 2.265636]  found at epochs 31
```

```
#test8
def func(x): return -2.54*x**2+1.024*x+0.0625
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -3 ,iv2 = 0,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
the roots found are :  [-0.053844  0.456993]  found at epochs 26
```

```
#test9
def func(x): return -1.72*x**3+2.24*x-0.1685*x+0.001
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -5 ,iv2 = 5,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
the roots found are :  [-1.097192e+00 -4.830000e-04  1.097675e+00]  found at epochs 29
```

```
#test10
def func(x): return 1.72*x**3+2.24*x-0.1685*x+0.001
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -5 ,iv2 = 5,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)

```

```
the roots found are :  [-0.000483]  found at epochs 27
```

```
#test11
def func(x): return -0.4*x**5+6.2*x**3-0.69*x+0.473*x+0.01
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -10 ,iv2 = 10,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)

```

```
the roots found are :  [-3.932494 -0.207136  3.932599]  found at epochs 34
```

```
#test12
def func(x): return -1.79*x**2-2.273*x+1.21
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -1 ,iv2 = 5,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
the roots found are :  [-1.673712  0.403879]  found at epochs 27
```

```
#test13
def func(x): return  -0.37*x**3+1.753*x**2-0.98*x-0.3
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -1 ,iv2 = 5,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
the roots found are :  [-0.217564  0.924564  4.030838]  found at epochs 27
```

```
#test14
def func(x): return  6.81*x**3+15.654*x**2-10
n_roots, end_bisect = pyg.bisect_n (func,iv1 = 0 ,iv2 = 1,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
the roots found are :  [-1.885733  0.699803]  found at epochs 30
```

```
#test15
def func(x): return   90*x**3+200.153*x**2-173.12*x-10
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -3.5,iv2 = 2,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
the roots found are :  [-2.878714 -0.054423  0.709215]  found at epochs 35
```

These are the Trigo functions

```
def func(x): return  np.sin(2*x)+np.cos(2*x)
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -1,iv2 = 0,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
the roots found are :  [-9.817477 -8.246681 -6.675884 -5.105088 -3.534292 -1.963495 -0.
  1.178097  2.748894  4.31969   5.890486  7.461283]  found at epochs 24
```

```
def func(x): return  2*np.sin(2*x)
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -1,iv2 = 4,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
the roots found are :  [-9.424778 -7.853982 -6.283185 -4.712389 -3.141593 -1.570796 -0.
  1.570796  3.141593  4.712389  6.283185  7.853982]  found at epochs 28
```

```
def func(x): return  np.tan(x)-np.sin(2*x+1)
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -1,iv2 = 4,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
the roots found are :  [-8.779783 -5.63819  -2.496598  0.644995  3.786588  6.92818 ]  f
```

These are the Log functions

```
def func(x): return  np.log(x**2+0.5*x+0.7)
n_roots, end_bisect = pyg.bisect_n (func,iv1 = -1,iv2 = 0,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
the roots found are :  [-0.85208  0.35208]  found at epochs 27
```

```
def func(x): return   np.log(x**2+0.5*x+0.7)
n_roots, end_bisect = pyg.bisect_n (func,iv1 = 1.2,iv2 = 0,epochs= 100, tol= 1e-6)
print("the roots found are : " ,n_roots, " found at epochs", end_bisect)
```

```
the roots found are :  [-0.85208  0.35208]  found at epochs 27
```

Regula Falsi

Parameters

1. f : is the equation given by the user.
2. a & b : interval for expected roots.
3. n_roots : no. of roots.
4. pos : breakpoint / stop point.

return

1. roots : list of roots from the equation
2. n_roots : sorted roots
3. pos : values of the roots found.

```
## sample equation
def f(x): return 2*x**2 - 5*x + 3
roots, n_roots, pos, = pyg.rfalsi_n(f, a = 1.1, b = 0,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)

```

```
The number of roots are not in sorted and unique: [0.9999999905347476, 0.99999999242779
the roots found are in sorted and unique:  [1.  1.5]  found at 99
```

These are the polynomial test cases 1-15

```
#test 1
def f(x): return x**3 -6*x**2 - 9*x + 54
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -10, b = 10,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-3.0000000001522427, -3.000000000106
the roots found are in sorted and unique:  [-3.  3.  6.]  found at 99
```

```
#test#2
def f(x): return (x**2-2*x-2)*(3*x**2+10*x-8)
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -3, b = 6,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-4.000000000025875, -4.0000000000201
the roots found are in sorted and unique:  [-4.    -0.732  0.667  2.732]  found at 99
```

```
#test#3
def f(x): return x**3-x**2-x+1
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -3, b = 6,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-1.0000000018931783, -1.000000001419
the roots found are in sorted and unique:  [-1.  1.]  found at 99
```

```
#test4
def f(x): return (x**2-6*x+5)*(x**2+3*x-18)
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -3, b = 6,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-6.000000000009087, -6.0000000000053
the roots found are in sorted and unique:  [-6.  1.  3.  5.]  found at 99
```

```
#5
def f(x): return x**5-2*x**4+3.6*x**3-0.51*x**2-1.33*x-0.2
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -3, b = 1,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-0.18054869874867882, -0.18054870204
the roots found are in sorted and unique:  [-0.181  0.876]  found at 99
```

```
#6
```

```
#6
def f(x): return -5*x**5-2*x**4+x**2+1.33*x-0.4
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -3, b = 1,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-0.8007467179953826, -0.800746717166
the roots found are in sorted and unique:  [-0.801]  found at 99
```

```
#7
def f(x): return 2*x**4-4*x**3-0.78*x*2-1.785*x+1.4
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -1, b = 2  ,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [0.3693933943920742, 0.36939339531554
the roots found are in sorted and unique:  [0.369 2.266]  found at 99
```

```
#test8
def f(x): return  -2.54*x**2+1.024*x+0.0625
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -1, b = 0,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-0.053843880676820444, -0.0538438792
the roots found are in sorted and unique:  [-0.054  0.457]  found at 99
```

```
#test9
def f(x): return  -1.72*x**3+2.24*x-0.1685*x+0.001
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -1, b = 0,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-1.0971921327914136, -1.097192132346
the roots found are in sorted and unique:  [-1.097 -0.     1.098]  found at 99
```

```
#test10
def f(x): return  1.72*x**3+2.24*x-0.1685*x+0.001
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -1, b = 0,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-0.00048274670557280075, -0.00048274
the roots found are in sorted and unique:  [-0.]  found at 99
```

```
# 11
def f(x): return  -0.4*x**5+6.2*x**3-0.69*x+0.473*x+0.01
-1.79*x**2-2.273*x+1.21
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -3, b = 1,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-3.9324939396736074, -3.932493939663
the roots found are in sorted and unique:  [-3.932  3.933]  found at 99
```

```
# test12
def f(x): return -1.79*x**2-2.273*x+1.21
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -2, b = 0,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-1.6737117731696989, -1.673711772697
the roots found are in sorted and unique:  [-1.674  0.404]  found at 99
```

```
# test13
def f(x): return -0.37*x**3+1.753*x**2-0.98*x-0.3
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -2, b = 0,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-0.2175641618319215, -0.217564160893
the roots found are in sorted and unique:  [-0.218  0.925  4.031]  found at 99
```

```
#test 14
def f(x): return 6.81*x**3+15.654*x**2-10
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -2, b = 4,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-1.885733274345083, -1.8857332741814
the roots found are in sorted and unique:  [-1.886 -1.113  0.7  ]  found at 99
```

```
#test 15
def f(x): return 90*x**3+200.153*x**2-173.12*x-10
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -1, b = 0,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-2.8787139477447465, -2.878713947742
the roots found are in sorted and unique:  [-2.879 -0.054  0.709]  found at 99
```

```
#test1
def f(x): return np.sin(2*x)+np.cos(2*x)
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -3, b = 2,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-9.817477042471058, -9.8174770424681
the roots found are in sorted and unique:  [-9.817 -8.247 -6.676 -5.105 -3.534 -1.963 -
  5.89   7.461  9.032]  found at 99
```

```
#test2
def f(x): return  2*np.sin(2*x)
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -10, b = 10,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-9.424777960760006, -9.4247779607693
the roots found are in sorted and unique:  [-8.639400e+01 -9.425000e+00 -7.854000e+00 -
 -3.142000e+00 -1.571000e+00  0.000000e+00  1.571000e+00  3.142000e+00
  4.712000e+00  6.283000e+00  7.854000e+00  3.476172e+03]  found at 99
```

```
#test3
def f(x): return  np.tan(x)-np.sin(2*x+1)
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -1, b = 1,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-8.779782845641531, -8.7797828448811
the roots found are in sorted and unique:  [-8.78000000e+00 -5.63800000e+00 -2.49700000
  3.78700000e+00  6.92800000e+00  2.32654431e+05]  found at 99
```

These are the Log Functions

```
#test 1
def f(x): return np.log(x**2+0.5*x+0.7)
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -1, b = 0,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
The number of roots are not in sorted and unique: [-0.8520797284645353, -0.852079728938
the roots found are in sorted and unique:  [-0.852  0.352]  found at 99
```

```
#test 2
def f(x): return np.log(x**3-0.0125*x+1.7)
roots, n_roots, pos, = pyg.rfalsi_n(f, a = -1, b = 0,tol = 1e-06)
```

```
roots, n_roots, pos, = pyg.Ffalsi_n(f, a = -1, b = 0,tol = 1e-06)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at" , pos)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2: RuntimeWarning: invalid

The number of roots are not in sorted and unique: [-0.8925966540368541, -0.892596656858
the roots found are in sorted and unique:  [-0.893]  found at 99
```

Secant Method

Parameters1. f : is the equation given by the user.

2. a & b : interval for expected roots.
3. tol: tolerance.
4. epochs : user desired number of roots.

return

1. roots : list of roots from the equation
2. n_roots : sorted roots
3. end_epoch: values of the roots found.

these are the testing values of polynomial from 1-15

```
# sample equation Test 1
def f(x): return x**3 -6*x**2 - 9*x + 54
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-3.0000000000001936, -3.000000000565
the roots found are in sorted and unique:  [-3.  3.  6.]  found at epochs 6
```

```
# sample equation Test 2
def f(x): return (x**2-2*x-2)*(3*x**2+10*x-8)
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-4.000000000003347, -4.0000000002169
the roots found are in sorted and unique:  [-4.        -0.732051  0.666667  2.732051]  f
```

```
# sample equation Test 3
```

```
# sample equation Test 3
def f(x): return x**3-x**2-x+1
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-1.000000000001092, -1.0000000011736
the roots found are in sorted and unique:  [-1.        1.        1.00001   1.000012  1.
```

```
# sample equation Test 4
def f(x): return (x**2-6*x+5)*(x**2+3*x-18)
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-6.000000000007218, -6.0000000002102
the roots found are in sorted and unique:  [-6.  1.  3.  5.]  found at epochs 9
```

```
# sample equation Test 5
def f(x): return x**5-2*x**4+3.6*x**3-0.51*x**2-1.33*x-0.2
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-0.365490023573712, -0.3654900240147
the roots found are in sorted and unique:  [-0.36549  -0.180549  0.87554 ]  found at ep
```

```
# sample equation Test 6
def f(x): return -5*x**5-2*x**4+x**2+1.33*x-0.4
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-0.800746716531163, -0.8007467165731
the roots found are in sorted and unique:  [-0.800747  0.261054  0.636798]  found at ep
```

```
# sample equation Test 7
def f(x): return 2*x**4-4*x**3-.78*x*2-1.785*x+1.4
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [0.36939339656227954, 0.3693933959630
the roots found are in sorted and unique:  [0.369393 2.265636]  found at epochs 12
```

```
# sample equation Test 8
def f(x): return -2.54*x**2+1.024*x+0.0625
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-0.0538438736878614, -0.053843873773
the roots found are in sorted and unique:  [-0.053844  0.456993]  found at epochs 11
```

```
# sample equation Test 9
def f(x): return -1.72*x**3+2.24*x-0.1685*x+0.001
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-1.0971921364461015, -1.097192130626
the roots found are in sorted and unique:  [-1.097192e+00 -4.830000e-04  1.097675e+00]
```

```
# sample equation Test 10
def f(x): return 1.72*x**3+2.24*x-0.1685*x+0.001
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-0.0004827418810057591, -0.000482741
the roots found are in sorted and unique:  [-0.000483]  found at epochs 13
```

```
### sample equation Test 11 (modify)
def f(x): return -0.4*x**5+6.2*x**3-0.69*x+0.473*x+0.01
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-3.932493940692583, -3.9324939466402
the roots found are in sorted and unique:  [-3.932494 -0.207136  0.04956   0.157471  3.
```

```
# sample equation Test 12
def f(x): return -1.79*x**2-2.273*x+1.21
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-1.6737117721925672, -1.673711771101
the roots found are in sorted and unique:  [-1.673712  0.403879]  found at epochs 9
```

```
# sample equation Test 13
def f(x): return -0.37*x**3+1.753*x**2-0.98*x-0.3
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-0.2175641565031178, -0.217564156493
the roots found are in sorted and unique:  [-0.217564  0.924564  4.030838]  found at ep
```

```
# sample equation Test 14
def f(x): return 6.81*x**3+15.654*x**2-10
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-1.885733283577479, -1.8857332739032
the roots found are in sorted and unique:  [-1.885733 -1.112748  0.699803]  found at ep
```

```
# sample equation Test 15
def f(x): return 90*x**3+200.153*x**2-173.12*x-10
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-2.878713974356992, -2.8787139480620
the roots found are in sorted and unique:  [-2.878714 -0.054423  0.709215]  found at ep
```

These are the Trigo function

```
# test1
def f(x): return np.sin(2*x)+np.cos(2*x)
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-9.817477205771207, -8.2466807156486
the roots found are in sorted and unique:  [-9.81747700e+00 -8.24668100e+00 -6.67588400
 -3.53429200e+00 -1.96349500e+00 -3.92699000e-01  1.17809700e+00
  2.74889400e+00  4.31969000e+00  5.89048600e+00  7.46128300e+00
  9.03207900e+00  1.84568600e+01  3.18675246e+03]  found at epochs 2
```

```
# test2
def f(x): return 2*np.sin(2*x)
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
```

```
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-9.424777960760006, -7.8539816339768
the roots found are in sorted and unique:  [-523.075177   -9.424778   -7.853982   -6.28
   -1.570796    0.          1.570796    3.141593    4.712389    6.283185
    7.853982]  found at epochs 5
```

```
# test3
def f(x): return np.tan(x)-np.sin(2*x+1)
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-8339995217420718.0, -8.779782757414
the roots found are in sorted and unique:  [-8.33999522e+15 -6.53284460e+01 -8.77978300
 -2.49659800e+00  6.44995000e-01  3.78658800e+00  6.92818000e+00
  1.00697730e+01  8.23264040e+01  2.12184335e+10]  found at epochs 9
```

Test cases for Log function

```
# test1
def f(x): return np.log(x**2+0.5*x+0.7)
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [0.35207972888902017, -0.852079730475
the roots found are in sorted and unique:  [-0.85208  0.35208]  found at epochs 8
```

```
# test1
def f(x): return np.log(x**3-0.0125*x+1.7)
roots, n_roots, end_epoch = pyg.sec_n(f, a = -10, b = 10,epochs =100)
print("The number of roots are not in sorted and unique:", roots)
print("the roots found are in sorted and unique: " ,n_roots, " found at epochs", end_epoch
```

```
The number of roots are not in sorted and unique: [-0.892596656994978]
the roots found are in sorted and unique:  [-0.892597]  found at epochs 10
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2: RuntimeWarning: invalid
```

References:

[1] BYJU'S.(2021), about "*Polynomial Function Definition*" Online[Accessed: 04-March-2021]

[2] Britannica(2021), about"*Transcendental function* " Online [Accessed: 04-March-2021]

[3] freeCodeCamp.org (2018), about "*Brute Force Algorithms Explained*" Online [Accessed: 04-March-2021]

[4] Mathematical Python(2019), about "*Newton's Method*" Online [Accessed: 04-March-2021]

[5]BYJU'S.(2021), about, "*Bisection Method - Definition, Procedure, and Example*" Online [Accessed: 07-Mar-2021].

[6] E. Chopra (2021), about "*Regula Falsi Method for finding root of a polynomial*," OpenGenus IQ: Learn Computer Science, 24-Sep-2019. Online [Accessed: 07-Mar-2021]

[7] MK Docs (2014), about " *Secant Method*" Online [Accessed: 07-March-2021]