Aquino, Nathaniel B.
Lazo, Alessandra Franchesca O.

CMSC 117
Problem Set 1

Problem 1

Given the following graph of a polynomial function, find a root of the function by applying the given method and identifying inputs. Reason out your answers.

(a) Bisection Method at the interval [-6, 2]

(b) Newton Method, with initial iterate x_0 as shown in the figure.

a.) Starting with the interval [-6,2], we calculated the midpoint using c= $\frac{a+b}{2}$ such that, c= $\frac{-6+2}{2}$ which resulted to c= -2. Since $f(-2)$ is not close enough to zero, $f(-6)$ and $f(-2)$ have different signs, and $f(-6)f(-2) < 0$, then the root is in the subinterval [-6,-2] as shown in figure 1.

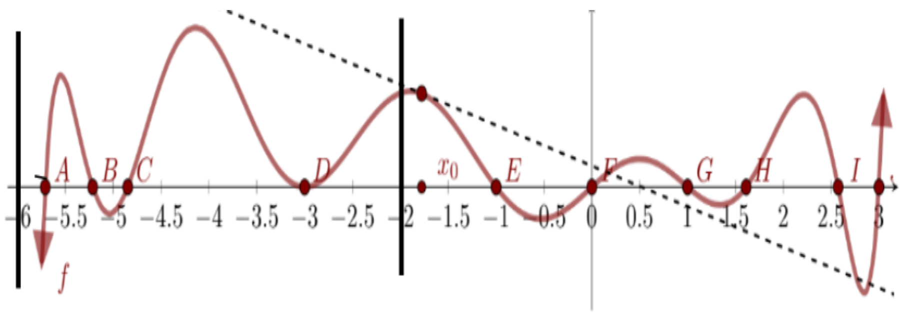


Figure 1: subinterval [-6,-2]

We calculated the midpoint of the subinterval [-6,-2], which is c = $\frac{-6-2}{2}$ = -4. Since $f(-4)$ is not close enough to zero, $f(6)$ and $f(-4)$ have different signs, and $f(-6)f(-4) < 0$, then the root is in the subinterval [-6,-4] as shown in figure 2.

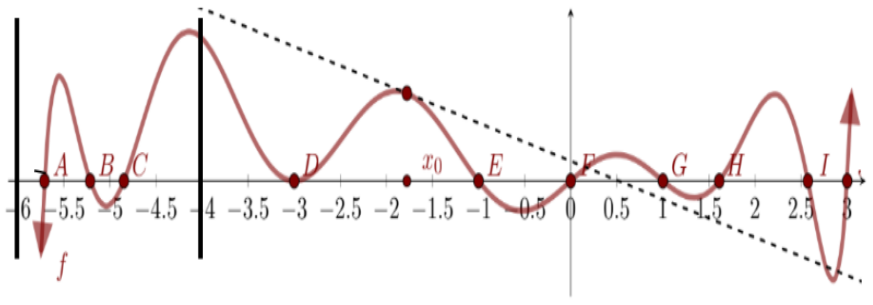


Figure 2: subinterval [-6,-4]

We calculated the midpoint of the subinterval [-6,-4], which is c = $\frac{-6-4}{2}$ = -5. Since $f(-5)$ is not close enough to zero, $f(-4)$ and $f(-5)$ have different signs, and $f(-5)f(-4) < 0$, then the root is in the subinterval [-5,-4] as shown in figure 3.

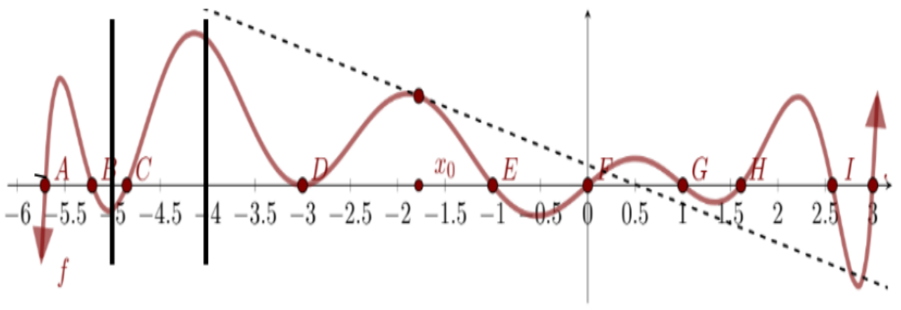


Figure 3: subinterval [-5,-4]

We calculated the midpoint of the subinterval [-5,-4], which is c = $\frac{-5-4}{2}$ = -4.5. Since $f(-4.5)$ is not close enough to zero, $f(-4.5)$ and $f(-5)$ have different signs, and $f(-5)f(-4.5) < 0$, then the root is in the subinterval [-5,-4.5] as shown in figure 4.

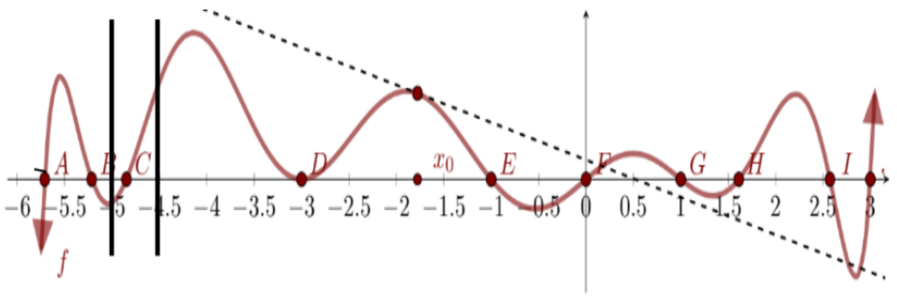


Figure 4: subinterval [-5,-4.5]

We calculated the midpoint of the subinterval [-5,-4.5], which is c = $\frac{-5-4.5}{2}$ = -4.75. Since $f(-4.75)$ is close enough to zero, $f(-4.5)$ and $f(-5)$ have different signs, and $f(-5)f(-4.75) < 0$, then the root is in the subinterval [-5,-4.75] as shown in figure 5.

We calculated the midpoint of the subinterval [-5,-4.75], which is c = $\frac{-5-4.75}{2}$ = -4.875. Since $f(-5)f(-4.75) = 0$, then root $c = -4.875$ which is also point C as shown in figure 6.

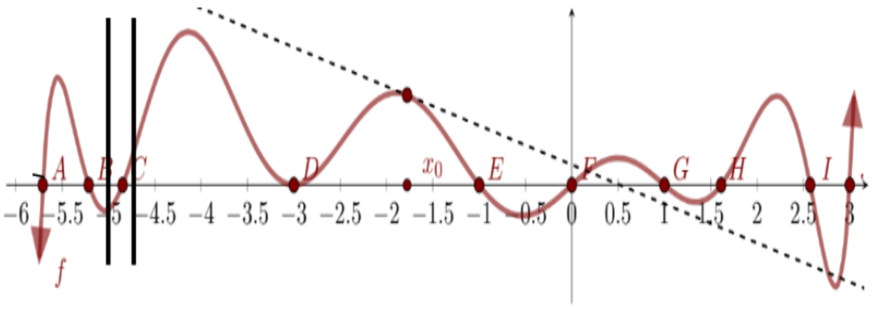


Figure 5: subinterval [-5,-4.75]

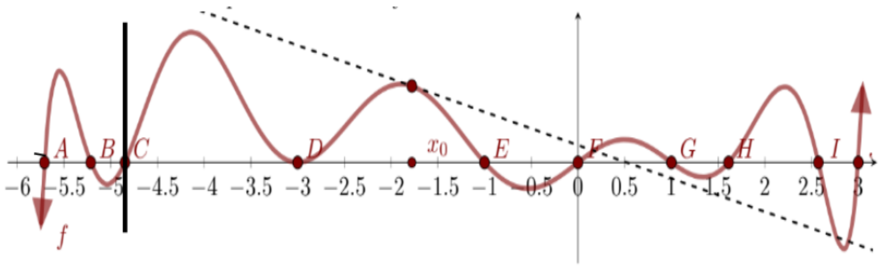


Figure 6: subinterval [-5,-4.75]

b.) We begin with an initial estimate x_0, as depicted in the given. The next estimate, x_1, is calculated as the point where the tangent to the graph at x_0 crosses the x-axis. In this case, $x_1 = 0.5$ and $f'(x_1) = 0$. Since the tangent at x_1 is horizontal, it does not intersect the x-axis. Thus, there is no further estimates can be derived.

Problem 2

Use the Newton-Raphson Method to approximate the value $\sqrt{2}\sqrt[3]{3}\sqrt[4]{4}$ correct to 9 decimal places. Use the initial iterate $x_0 = 3$. How many iterations do we need to achieve the approximate value?

```

def newton_raphson(f,fprime,x,maxit,tol,wa,wf):
  """ Computes a root of function using newton raphson method
  Inputs

  Args:
      f (_type_)       :  callable function
      fprime (_type_) :  callable function
                         derivative function
      x (_type_)       : initial iterated value
      maxit (_type_)   : int
      tol (_type_)     : scalar
      wa (_type_)      : _float
      wf (_type_)      : _float

  Raises:
      RuntimeError: string for error message

  Returns:
      x                : scalar
                         root of function f
      k                : integer
                       : current iterate
      err              : scalar
                         error
  """

  err= tol +1
  fx=f(x)
  k=0
  while err> tol and k < maxit:
    xold=x
    x= x-(fx/fprime(x))
    fx=f(x)
    err= wa*abs(x-xold) + wf*abs(fx)
    k+=1
  if err > tol and k == maxit:
  return np.array([x,k,err])
```

Algorithm 1: rootscalar.py - newton$_r aphson$

The function *newton-raphson* take several parameters such as: **'f'** which calculates the root, **'fprime'** which is the derivative of the function 'f', **'x'** which is the initial guess for the root, **'maxit'** which is the maximum number of iterations, **'tol'** which is the tolerance for convergence, **'wa'** which is the weight factor for the change in root, and **'wf'** which is the weight factor for the function value.

The variable **'err'** is initialized to a value greater than the tolerance to ensure the loop starts, the variable **'fx'** stores the function value at the current 'x', and the variable **'k'** is the counter for the number of iterations.

The function iteratively applies the Newton-Raphson fomula. It continues until the error **'err'** is within the tolerance **'tol'** or the maximum number of iterations **'maxit'** is reached. The error is calculated using a weighted sum of the change in the root and the function value.

An error will be raised when the loop exits due to reaching the maximum number of iterations without meeting the tolerance criteria. Otherwise, the function will return an array containing the root approximation, the number of iterations performed, and the final error estimate.

```python
import numpy as np
from rootscalar import *

def fun(x):
    return x**3-24

def fprime(x):
    return 3*x**2

x=3
maxit=100
tol=10e-9
wa=0.5
wf=0.5


root=newton_raphson(fun,fprime,x,maxit,tol,wa,wf)
actual= 2**(1/2)*3**(1/3)*4**(1/4)

print(f"{'Approximate':<25} {'Actual x':<25} {'Iteration k':<15}")
print(f"{root[0]:<+25.9e} {actual:<+25.9e} {root[1]:<15}")
```

Algorithm 2: Item2Imp.py

Two functions are defined: **'fun(x)'** which represents the original function $f(x) = x^3 - 24$, and **'fprime(x)'** which represents its derivative $f(x) = 3x^2$.

Initial parameters were called such as: **'x'** which is set to 3, **'maxit'** which is set to 100, **'tol'** which is set to 10^{-9}, and **'wa'** and **'wf'** which is both set to 0.5. The function **'newton-raphson'** is also called to find the root of the function. The actual root of the given function is calculated manually using the given mathematical expression. The results is then printed in a formatted way that shows the approximate root found using the Newton-Raphson method, the actual root was calculated manually, and the number of iterations performed by the algorithm to reach the solution was also printed.

Table 1: **Newton-Raphson Approximation Result**

Approximate	Actual x	Iteration k
$+2.884499141e+00$	$+2.884499141e+$ 00	4.0

Problem 3

Use the Newton-Horner Method with refinement in your `rootpoly.py` to find all the roots of the polynomial $p_{n-1}(x) = \sum_{k=1}^{n}(k^k (\text{mod } 7))x^{k-1}$ for $n = 20, 40, 50$. Use $z = 1 + j$, `maxit = refmax = 100`, `tol =` 10^3 `eps`, and `reftol =` 10^{-3}. Plot the roots you obtain on each n. Include the plot and maximum function value of approximate roots in your documentation.

```python
def horner(p, z):
    lenp = len(p)
    b = np.zeros_like(p, dtype=complex)
    b[-1] = p[-1]
    for k in range(lenp - 2, -1, -1):
        b[k] = p[k] + b[k + 1] * z
    return [b[0], b[1:]]
```

Algorithm 3: rootpoly.py - horner

The function *horner*(p, z) takes two parameters: **'p'** which is a list representing the coefficients of a polynomial, and **'z'** which is the point at which the polynomial needs to be evaluated.

The length of the input polynomial **'p'** is calculated and stored in the variable **'lenp'**. An array **'b'** of the same length as the input polynomiam **'p'** is initialized with zeros. This array will store the intermediate results during the evaluation process. The last element of array **'b'** is set to be equal to the last coefficient of the input polynomial. For each coefficient **'p[k]'**, it multiplies the previously computed result **b[k+1]** by the evaluation point **'z'**, adds the current coefficient **'p[k]**, and stores the result in **'b[k]'**.

The function will return a list containing two elements: the evaluated polynomial value and the point **'z'** which is stored in **'b[0]'** and the remaining coefficients after division by **'(x-z)'** which is stored in **'b[1:]'**.

```python
def newtonhorner(p, z, maxit, refmax, tol, reftol, ref):
    erref = tol * reftol
    n = len(p)
    zm = np.zeros(n - 1, dtype=complex)
    eps = np.finfo(float).eps
    pnm = [p]

    for m in range(n-1):
        k = 0
        z = complex(z, z)
        err = tol + 1

        if m == n - 2:
            k += 1
            b = pnm[m]
            z = -b[0] / b[1]
        else:
            while err > tol and k < maxit:
                k += 1
                zold = z
                pz, qnm = horner(pnm[m], z)
                qz = horner(qnm, z)[0]
                if abs(qz) > eps:
                    z = zold - pz / qz
                    err = max(abs(z-zold),abs(pz))
                else:
                    print("Division by a small number")
                    err = 0
                    z = zold

        if ref:
            kref = 0
            zref = z
            err = tol + 1
            while err > errref and kref < refmax:
                kref += 1
                pz, qn_1 = horner(p, zref)
                qz = horner(qn_1, zref)[0]
                if abs(qz) > eps:
                    zref2 = zref - pz / qz
                    err = max(abs(zref - zref2), abs(pz))
                    zref = zref2
                else:
                    print("Division by a small number")
                    err = 0
            z = zref
        zm[m] = z
        pnm.append(horner(pnm[m], z)[1])

    return [zm, pnm]
```

Algorithm 4: rootpoly.py - newtonhorner

The function *newtonhorner* take several parameters such as: **'p'** which is the list of coefficients representing the polynomial, **'z'** which is the initial guess for the root, **'maxit'** which is the maximum number of iterations for Newton's method, **'refmax'** which is the maximum number of iterations for refining the root (if ref is True), **'tol'** which is the tolerance for convergence, **'reftol'** which is the tolerance for convergence criteria in root refinement, and **'ref'** which is the boolean flag indicating whether to perform root refinement.

Several variables were called such as: **'errref'** which is a threshold for refinement tolerance, **'n'** which is the degree of the polynomial, **'zm'** which is an array to store intermediate roots, **'eps'** which is the machine epsilon for floating-point arithmetic, and **'pnm'** which is a list containing the polynomial at different stages of division.

The function iterates through the coefficients of the input polynomial. If **'m'** is not the last coefficient, the function iteratively applies Newton's method to find a root **'z'**. Then iteration stops when the error **'err'** is within the tolerance **'tol'** or the maximum number of iterations **'maxit'** was reached.

Intermediate roots and modified polynomials after each division step are stored in **'zm'** and **'pnm'**, respectively. The function returns a list containing intermediate roots **'zm'** and modified polynomials **'pnm'** after each step of the Newton's method.

```
import numpy as np
import matplotlib.pyplot as plt
from rootpoly import *

plt.style.use('seaborn-v0_8-whitegrid')


def poly(n):
    #pn-1(x) = sum of k=1 to n (k^k ( mod 7))x^k-1 for n=20,40,50
    p=[]
    for k in range(1,n+1):
        p.append(np.mod(k**k,7))
    return p

def findmax(n,a):
    p = poly(n)
    max_fv = max(np.linalg.norm(horner(p, ak)[0]) for ak in a)
    return max_fv

n_values = [20,40,50]


for n in n_values:
    a, b = newtonhorner(poly(n), complex(1, 1), 100, 100, 1e-3, 1e-3, True)
    plt.figure(figsize=(10, 6))
    plt.plot(a.real, a.imag, '.', markersize=5)
    plt.xlabel('Real')
    plt.ylabel('Imaginary')
    plt.title(f'Newton-Horner Plot for n={n}')
    plt.grid(True)
    plt.axis("equal")
    plt.show()
    max_fv = findmax(n,a)
    print(f"Maximum function value of approximate roots for n={n}: {max_fv:.15e}")
```

Algorithm 5: Item3Imp.py

The function $poly(n)$ calculates the coefficient of a polynomial $P_{n-1}(x) = \sum_{k=1}^{n}(k^k mod 7)k^{k-1}$ for the given value of n. The coefficients are calculated using the given formula and stored in the list **'p'**.

The function $findmad(n, a)$ takes the polynomial degree **'n'** and a list of complex roots **'a'**. It calculates the maximum function value by evaluating the polynomial at each root and finding the maximum magnitude among all the values.

The loop iterated through the **'nvalues'** list (which contains the polynomial degree 20, 40, and 50). For each **'n'**, it calculates the roots using the **'newtonhorner'** function and then plots these roots on a complex plane. After plotting, it calculates and prints the maximum function value among the roots for the given polynomial degree.

The following plots show the solution of the polynomial for $n = 20, n = 40, n = 50$, respectively.

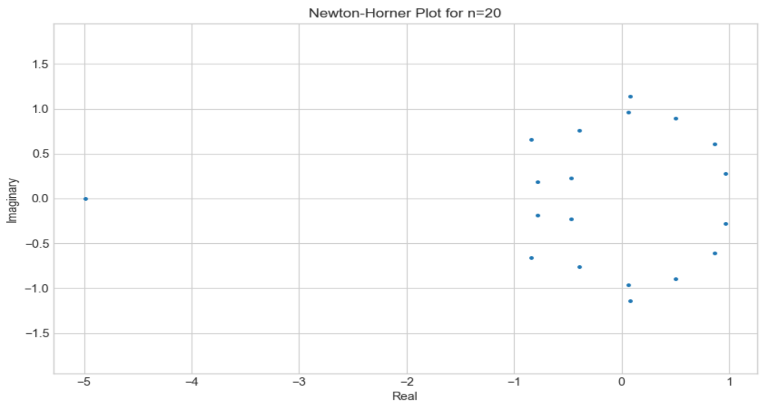


Figure 7: plot for $n = 20$

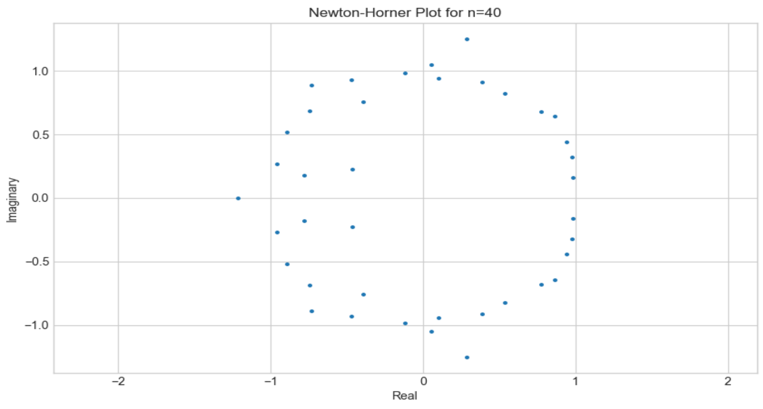


Figure 8: plot for $n = 40$

Problem 4

Modify `rootscalar.py` to find an approximate solution of $e^{\cos x} - 3\sin x = 0$ on the interval [-2, 2] using Inexact Newton-Raphson Method (forward, backward, central finite difference), Steffensen Method, and Picard Fix Point Method with `tol` = 10^{-10}, `maxit = 1000`, `wa = 0.5`, and `wf = 0.5`. Use your best judgement for other inputs e.g. initial iterates.

```python
def newton_raphson(f,fprime,x,maxit,tol,wa,wf):
  """ Computes a root of function using newton raphson method
  Inputs

```

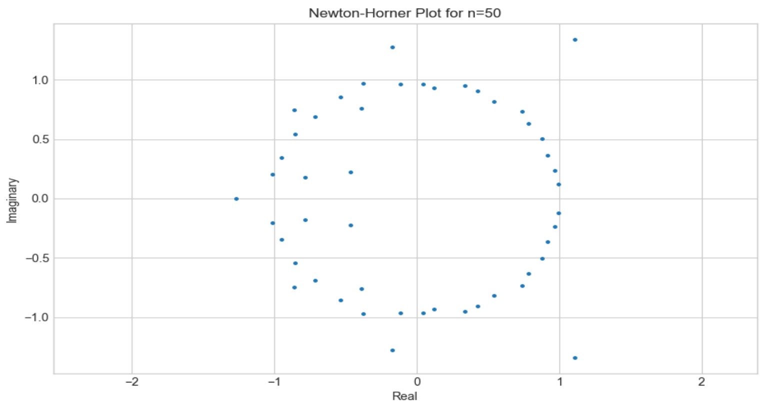
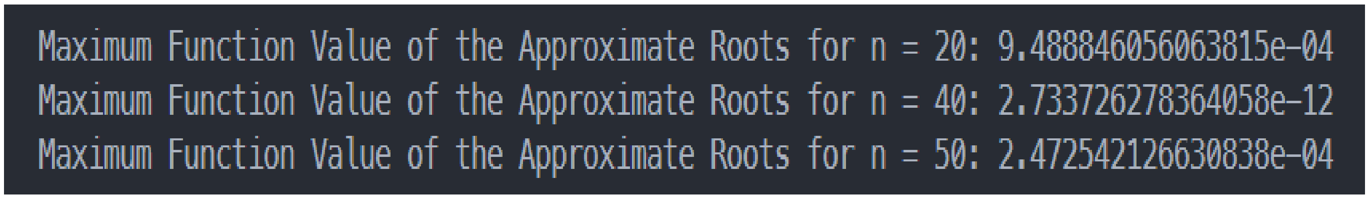


Figure 9: plot for $n = 50$



```
Maximum Function Value of the Approximate Roots for n = 20: 9.488846056063815e-04
Maximum Function Value of the Approximate Roots for n = 40: 2.733726278364058e-12
Maximum Function Value of the Approximate Roots for n = 50: 2.472542126630838e-04
```

Figure 10: Newton-Horner Approximation Result

```
Args:
    f (_type_)       :  callable function
    fprime (_type_) :  callable function
                      derivative function
    x (_type_)       : initial iterated value
    maxit (_type_)  : int
    tol (_type_)     : scalar
    wa (_type_)      : _float
    wf (_type_)      : _float

Raises:
    RuntimeError: string for error message

Returns:
    x                : scalar
                      root of function f
    k                : integer
                     : current iterate
    err              : scalar
                      error
"""

err= tol +1
```

```
  fx=f(x)
  k=0
  while err> tol and k < maxit:
    xold=x
    x= x-(fx/fprime(x))
    fx=f(x)
    err= wa*abs(x-xold) + wf*abs(fx)
    k+=1
  if err > tol and k == maxit:
    raise RuntimeError("Error in maximum iterate and / or tolerance!")
  return np.array([x,k,err])
```

Algorithm 6: newtonraphson.py - newtonraphson

The function *newton-raphson* take several parameters such as: **'f'** which calculates the root, **'fprime'** which is the derivative of the function 'f', **'x'** which is the initial guess for the root, **'maxit'** which is the maximum number of iterations, **'tol'** which is the tolerance for convergence, **'wa'** which is the weight factor for the change in root, and **'wf'** which is the weight factor for the function value.

The variable **'err'** is initialized to a value greater than the tolerance to ensure the loop starts, the variable **'fx'** stores the function value at the current 'x', and the variable **'k'** is the counter for the number of iterations.

The function iteratively applies the Newton-Raphson fomula. It continues until the error **'err'** is within the tolerance **'tol'** or the maximum number of iterations **'maxit'** is reached. The error is calculated using a weighted sum of the change in the root and the function value.

An error will be raised when the loop exits due to reaching the maximum number of iterations without meeting the tolerance criteria. Otherwise, the function will return an array containing the root approximation, the number of iterations performed, and the final error estimate.

```
def newton_raphson_inexact(f,i,x,maxit,tol,wa,wf):
  """_summary_

  Args:
      f (_type_)     : callable function
      i (_type_)     : integer
      x (_type_)     : integer ( initial iterated value)
      maxit (_type_): int
      tol (_type_)   : scalar
      wa (_type_)    : float
      wf (_type_)    : flaot

  Raises:
      RuntimeError: string for error message

  Returns:
      _type_: scalar
  """
  err = tol + 1
  fx = f(x)
  k = 0
  h = np.sqrt(np.finfo(float).eps)
  while err > tol and k < maxit:
    xold = x
    if i == 1: #forward finite difference
      fprime = (f(x+h)-f(x))/h
    elif i == 2: # backward finite difference
      fprime = (f(x)-f(x-h))/h
    elif i == 3: # central finite differenc
      fprime = (f(x+h)-f(x-h))/(2*h)
      x = x - fx/fprime
      fx = f(x)
      err = wa*abs(x-xold) + wf*abs(fx)
      k += 1
      if err > tol and k == maxit:
        raise RuntimeError("Error in maximum iterate and / or tolerance!")
      return np.array([x, k, err])
```

Algorithm 7: newtonraphson.py - NewtonRaphsonInexact

The function *newton-raphson-inexact* take several parameters such as: **'f'** which is the parameter for which the root is being calculated, **'i'** which is the integer flag indicating the type of finite difference approximation for the derivative calculation, **'x'** which is the initial guess for the root, **'maxit'** which is the maximum number of iterations allowed, **'tol'** which is the tolerance for convergence, **'wa'** which is the weight factor for the change in the root, and **'wf'** which is the weight factor for the function value.

The variable **'err'** is initialized to a value greater than the tolerance to ensure the loop starts, the variable **'fx'** stores the function value at the current 'x', the variable **'k'** is a counter for the number of iterations, and the variable **'h'** is a small constant used for finite difference approximation

The derivative **'fprime'** is approximated using finite differences based on the value of the integer flag **'i'**. If **'i'** is 1, it uses forward finite differences, if **'i'** is 2, it uses backward finite differences, and if **'i'** is 3, it uses central finite differences.

Inside the loop, the function iteratively applies the Newton-Raphson formula with the approximated derivative. The iteration stops when the error **'err'** is within the tolerance **'tol'** or the maximum number of iterations **'maxit'** is reached. The error is calculated as a weighted sum of the change in the root and the function value.

The function will return an array containing the root approximation, the number of iterations performed, and the final error estimate. Otherwise, if the loop exits due to reaching the maximum number of iterations without meeting the tolerance criteria, an error will be raised.

```
def steffensen(f,x,maxit,tol,wa,wf):
  """_summary_

  Args:
      f (_type_): callable function
      x (_type_): initial iterated value
      maxit (_type_): int
      tol (_type_): scalar
      wa (_type_): float
      wf (_type_): float
```

```

Raises:
    RuntimeError: String for error message

Returns:
    _type_: scalar type
"""
err= tol +1
fx=f(x)
k=0
while err> tol and k < maxit:
  xold=x
  fx=f(x)
  q=(f(x+fx)-fx)/fx
  x=x-(fx/q)
  err= wa*abs(x-xold) + wf*abs(fx)
  k+=1
if err > tol and k == maxit:
  raise RuntimeError("Error in maximum iterate and / or tolerance!")
return np.array([x,k,err])
```

Algorithm 8: newtonraphson.py - Steffensen

The function *steffensen* take several parameters such as: **'f'** which is the parameter for which the root is being calculated, **'x'** which is the initial guess for the root, **'maxit'** which is the maximum number of iterations allowed, **'tol'** which is the tolerance for convergence, **'wa'** which is the weight factor for the change in the root, and **'wf'** which is the weight factor for the function value

The variable **'err'** is initialized to a value greater than the tolerance to ensure the loop starts, the variable **'fx'** stores the function value at the current 'x', and the variable **'k'** is a counter for the number of iterations.

Inside the loop, the function applies Steffensen's method iteratively: **'q'** is the acceleration factor, calculated using the difference quotient; **'x'** is updated using the formula $x = x - f(x)/q$. The error is calculated as a weighted sum of the change in the root and the function value.

The function will return an array containing the root approximation, the number of iterations performed, and the final error estimate. Otherwise, if the loop exits due to reaching the maximum number of iterations without meeting the tolerance criteria, it will raise an error.

```
def fixpoint(g,x,maxit,tol):
    err = tol + 1
    k = 0
    while err> tol and k < maxit:
        xold = x
        x = g(x)
        err = abs(x-xold)
        k += 1
    if err > tol and k == maxit:
        raise RuntimeError("Error in maximum iterate and / or tolerance!")
    return np.array([x, k, err])
```

Algorithm 9: newtonraphson.py - Fixpoint

The function *fixpoint* take several parameters such as: **'g'** which is the function for which the fixed point is being calculated, **'x'** which is the initial guess for the fixed point, **'maxit'** which is the maximum number of iterations allowed, and **'tol'** which is the tolerance for convergence.

The variable **'err'** is initialized to a value greater than the tolerance to ensure the loop starts, and the variable **'k'** is a counter for the number of iterations.

Inside the loop, the function applied the fixed-point iteration formula: $x_{new} = g(X_{old})$.The iteration stops when the absolute difference between the new and old values $(x - x_{old})$ is within the tolerance or the maximum number of iterations is reached.

The function returns an array containing the fixed-point approximation, the number of iterations performed, and the final error estimate. Otherwise, the loop exits due to reaching the maximum number of iterations without meeting the tolerance criteria, raising an error.

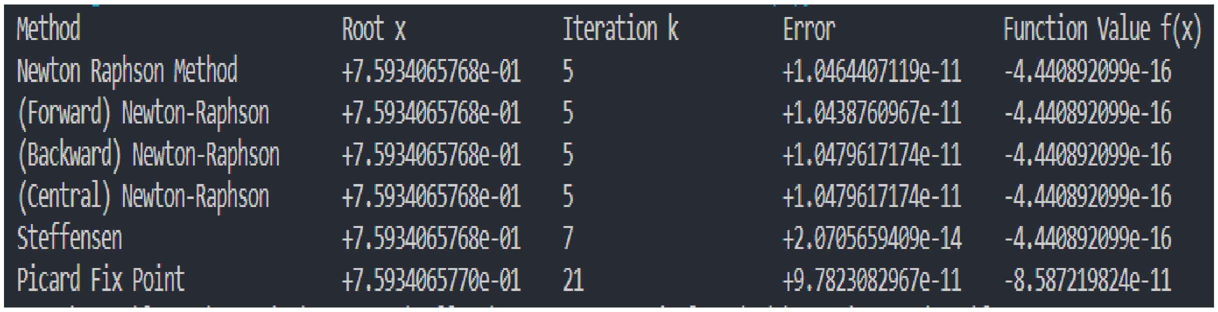
```python
import numpy as np
from rootscalar import *

def funct (x):
    return np.exp(np.cos(x))-(3*np.sin(x))

def gfunct (x):
    return x*np.exp(np.cos(x))/ (3*np.sin(x))

def functprime (x) :
    return -np.exp(np.cos(x))*np.sin(x)-(3*np.cos(x))

# def funct ( x ) :
#     return np . cos ( x ) - x
# def gfunct ( x ) :
#     return np . cos ( x )
# def functprime ( x ) :
#     return - np.sin ( x ) - 1

n= newton_raphson(funct, functprime ,0 ,1000 ,1e-10 ,0.5 ,0.5)
n_forward = newton_raphson_inexact(funct ,1 ,0 ,1000 ,1e-10 ,0.5 ,0.5)
n_backward = newton_raphson_inexact(funct ,2 ,0 ,1000 ,1e-10 ,0.5 ,0.5)
n_central = newton_raphson_inexact(funct ,3 ,0 ,1000 ,1e-10 ,0.5 ,0.5)
f = steffensen (funct , 0 ,1000 ,1e-10 ,0.5 ,0.5)
fix = fixpoint (gfunct , 1 , 1000 , 1e-10)

print(f"{'Method':<30} {'Root x':<25} {'Iteration k':<20} {'Error':<25} {'Function Value f(x)
    ':<20}")

def printformat(name, x, f):
    return print(f"{name:<30} {x[0]:<+25.16e} {str(int(x[1])):<20} {x[2]:<+25.15e} {f(x[0]):<+20.9
    e}")


printformat("Newton Raphson Method", n, funct)
printformat("(Forward) Newton-Raphson", n_forward , funct)
printformat("(Backward) Newton-Raphson", n_backward , funct)
printformat("(Central) Newton-Raphson", n_central , funct)
printformat("Steffensen", f, funct)
printformat("Picard Fix Point",fix, funct)
```

Algorithm 10: Item4Imp.py

The function **'funct(x)'** represents the given function $f(x) = e^{cos(x)} - 3sin(x)$. The function **'gfunct(x)'** represents a function used for fixed-point iteration. Lastly, the function **'functprime(x)'** represents the derivative of $funct(x)$.

There are several root-finding methods used such as: **'newton-raphson'** which applies the Newton-Raphson method, **'newton-raphson-inexact'** which applies an inexact Newton-Raphson method with different finite difference schemed, **'steffensen'** which applies Steffensen's method, and **'fixpoint'** which applies the fixed-point iteration method using $gfunct(x)$.

The results are printed in a tabular format, displaying the method name, the estimate root, the number of iterations, the error, and the function value at the estimated root for each method.



```
Method                     Root x             Iteration k        Error              Function Value f(x)
Newton Raphson Method      +7.5934065768e-01  5                  +1.0464407119e-11  -4.440892099e-16
(Forward) Newton-Raphson   +7.5934065768e-01  5                  +1.0438760967e-11  -4.440892099e-16
(Backward) Newton-Raphson  +7.5934065768e-01  5                  +1.0479617174e-11  -4.440892099e-16
(Central) Newton-Raphson   +7.5934065768e-01  5                  +1.0479617174e-11  -4.440892099e-16
Steffensen                 +7.5934065768e-01  7                  +2.0705659409e-14  -4.440892099e-16
Picard Fix Point           +7.5934065770e-01  21                 +9.7823082967e-11  -8.587219824e-11
```

Figure 11: Newton-Raphson Approximation Result