Week 7 assignment

Question 3:

```
def Bisection(n,a,err):
    # checking whether n th root can be found or not
    if a<0 and (a%2==0 and n%2==0):
        try:
            raise Exception(str(n)+"th root does not exist ")
        except Exception as inst:
                print(type(inst))
                print(inst)
        return None
```

- In order to find nth root, we first check whether the integer is negative and both the **a** and **n** are not even integers

```
# nth root of a  means finding the solution of x^n-a
f= lambda x: x**n -a
# implementing bisection to approximate the solution of f
while(abs(an-bn) >err):
    cn=(an+bn)/2
    if (f(cn)<0 and f(an)<0) or (f(cn)>0 and f(an)>0):
        an=cn
    else:
        bn=cn
print("The",n,"th root of",a,"is",(an+bn)/2)
```

- The above code finds the root of the equation $x^n - a = 0$ with help of bisection method

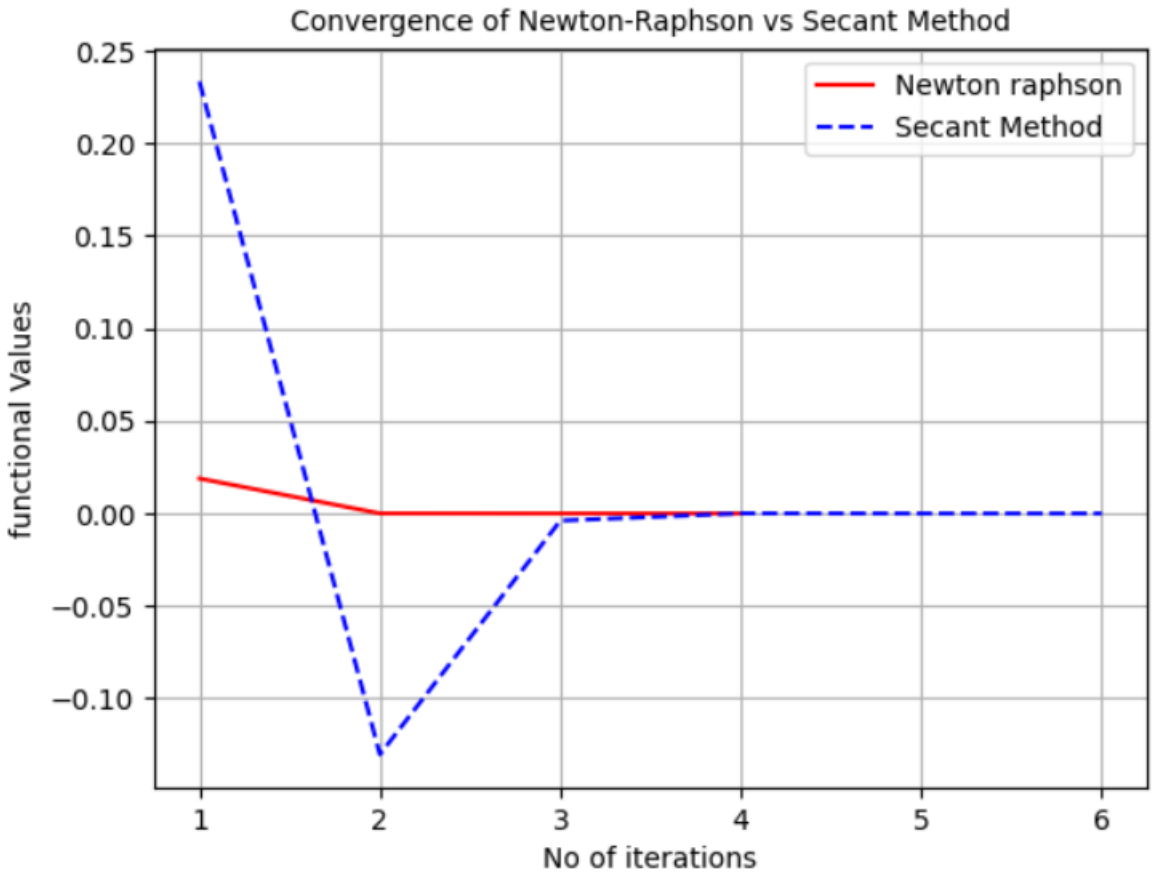
Question 4:

```
f = lambda x: x - np.cos(x)  # function f
df = lambda x: 1 + np.sin(x)  # derivative of f
# Calculation by secant method
x0 = 0
x1 = 5
h = 0
secant_value = []
secant_counter = []
while abs(f(x1)) > 10 ** (-10):  # error threshold is 10^-10
    if (f(x1) - f(x0) != 0):
        x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
        x0 = x1
        x1 = x2
        h = h + 1
        secant_counter.append(h)
        secant_value.append(f(x2))
```

- The above code finds the root of a equation $\boldsymbol{x - cosx = 0}$ by secant method

```
# calculation by Newton Raphson method
k = 0
x0 = 1
newton_value = []
newton_counter = []
while abs(f(x0)) > 10 ** (-10):
    h = f(x0) / df(x0)
    xk = x0 - h
    x0 = xk
    k = k + 1
    newton_counter.append(k)
    newton_value.append(f(x0))
    # print(x0)
```

- The above code generates the root of the same function via **Newton-Raphson method**
- Also the code generates a plot comparing the convergence of the two methods in which clearly see that the **Newton-Raphson method converges faster**



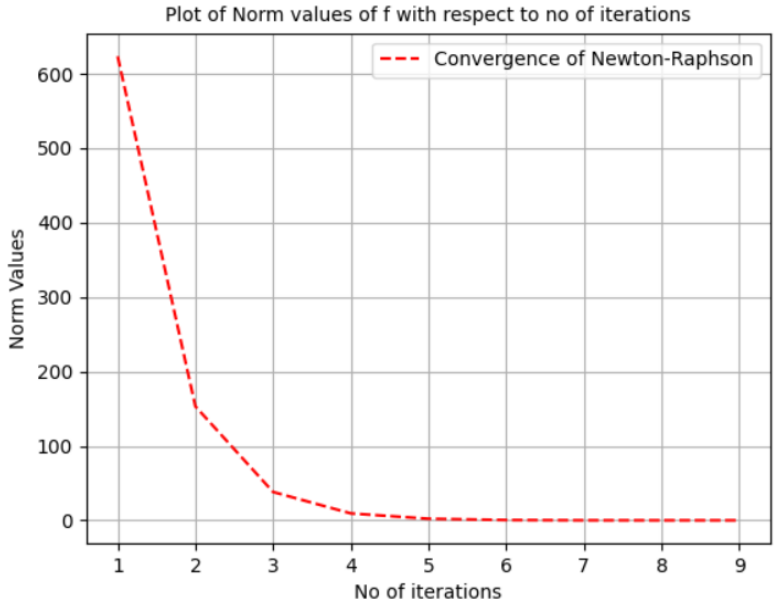
Question 5:

```
def Newton_Raphson(F, J, x, eps):
    F_value = F(x)
    # to start the iteration
    F_norm = np.linalg.norm(F_value, ord=2)  # l2 norm of vector
    norm_values = []
    counter = []
    iteration_counter = 0
    # calculation for the norm
    while abs(F_norm) > eps:
        delta = np.linalg.solve(J(x), -F_value) # solving the linear
equation as menioned before
        x = x + delta
        F_value = F(x)
        F_norm = np.linalg.norm(F_value, ord=2)
        norm_values.append(F_norm)
        iteration_counter += 1
        counter.append(iteration_counter)
```

- The above code implements the Newton-Raphson method for a vector valued function by simplifying the original equation as follows

$$x_{k+1} = x_k - j(x_k)^{-1}F(x_k)$$
$$(x_{k+1} - x_k)j(x_k) = -F(x_k)$$
$$\Delta x\, j(x_k) = -F(x_k)$$

- Then we solve the equation via **NumPy. linalg**
- Also, we plot the convergence of the method and a sample plot is as follows



Question 6:

```
self.array = array
# creating a polynomial out of the given array elemnts
p = Polynomial([1])
for i in self.array:
    p = p * Polynomial([-i, 1])

coef = p.get_coef()
degree = len(coef) - 1  # getting degree of the Polynomial
dp = p.derivative()
# suitable upper and lowerbounds to start Aberth's method
upper = 1 + 1 / abs(coef[-1]) * max(abs(coef[x]) for x in
range(degree))
lower = abs(coef[0]) / (abs(coef[0]) + max(abs(coef[x]) for x in
range(1, degree + 1)))
```

- In this above code we calculate a suitable lower bound and upper bound to calculate an initialization to start our process
- We have created a polynomial object with the input given by virtue of our polynomial class

```
for i in range(degree):
    radius = random.uniform(lower, upper)
    angle = random.uniform(0, np.pi * 2)
    root = complex(radius * np.cos(angle), radius * np.sin(angle))
    roots.append(root)
iteration = 0
# calculation for Abeth's method
while iteration < 1000:

    for i in range(len(roots)):
        ratio = p[roots[i]] / dp[roots[i]]
        offset = ratio / (1 - (ratio * sum(1 / (roots[i] - j)
                                           for j in (roots) if j !=
roots[i])))

        roots[i] -= offset
    iteration += 1
print("The estimated roots are :", roots)
```

- The above code is used to generate an initial approximation for the root
- Next, we calculate the iterations using Aberth's rule
- A sample output for the following input **q.Aberth([-1,1,0])**

```
C:\Users\avij1\PycharmProjects\pythonProject1\ve
The estimated roots are : [(1+0j), 0j, (-1+0j)]

Process finished with exit code 0
```

Question 7:

```
s = function  # the continuous function
x0, xn = a, b
x = np.linspace(x0, xn, num=5)
y = [s(i) for i in x]
model = np.polyfit(x, y, len(y) - 1)  # generating a polynomial close
to function
p = Polynomial(list(model))
roots = p.Aberth()  # finding the roots of the Polynomial via Aberth
method
```

- The above code creates a polynomial fit with NumPy.polyfit for the given continuous function
- We create a polynomial object with coefficients and use Aberth's method to find the roots of the polynomial
- We only consider the real roots for our purpose.

```
new_roots = [x0]
for i in range(len(roots1)):
    if (i == len(roots1) - 1):
        break
    new_roots.append((roots1[i] + roots1[i + 1]) / 2)
new_roots.append(xn)
interval_roots = []
# Using bisection to get the roots if it is in the sub-intervals
for i in range(len(new_roots) - 1):
    if (s(new_roots[i]) * s(new_roots[i + 1]) < 0):
        r = round(bisection(s, new_roots[i], new_roots[i + 1]), 10)
        interval_roots.append(r)
print(interval_roots)
```

- The above code generates a list with sub-intervals around the roots that we have computed above
- We then find the roots of the function inside those sub-intervals via bisection method.
- A sample output for the input **find_root(lambda x: np.sin(x), -np.pi/2,4*np.pi/3)**

```
The roots in the interval[ -1.5707963267948966 4.1887902047863905 ] are :
 [-0.0, 3.1415926536]
```