S16036

CPL 105

1. Bisection method

I.

a.) What is the Intermediate value theorem?

It is a method which is used to ensure that a root exists within any given range. It is used to find roots in bisection method also. For any function f that is continuous over the interval [a,b], the function will take any value between $f(a)$ and $f(b)$ over the interval.

b.) Explain how the intermediate value theorem is used in bisection method to find roots.

Pick two values of x,x, say x=a and x=b, such that $f(a)$ is negative and $f(b)$ is positive. Then the Intermediate Value Theorem implies that f must have at least one root between x=a and x=b.

c.) Let's consider that there is a function $y(x)$. If there are two values on x axis, γ and δ, where $\delta > \gamma$, what are the conditions for having a root for $y(x)$ between γ and δ.

When we substitute γ and δ to $y(x)$, it should satisfy the following condition.

Condition: $y(\gamma) > 0 \; and \; y(\delta) < 0$

d.) If $\beta = (\gamma+\delta)/2$, how do we determine whether there is a root for $y(\beta)$ using a tolerance?

If $y(\beta)$ is less than the tolerance, we can assume that β is a root of the $y(x)$ function. If not, replace γ or δ accordingly by β, and repeat the process to find the root.

e.) Discuss some practical considerations and limitations of the bisection method.

Practical uses:

- To solve complex polynomial equations
- Used in thermistors by measuring electrical resistance with a change in temperature

Limitations:

- Convergence is slow. So computational time is high
- Cannot handle certain functions
- Requires knowledge of a bracketing interval
- Requires a continuous function in the considered interval
- Average error is higher than other methods

II. Bisection theory

```python
import numpy as np

def my_bisection(f, a, b, tol):
    #f (function) - The function for which the root is to be found
    #a (float) - The lower bound of the interval
    #b (float) - The upper bound of the interval
    #tol (float) - The tolerance level for the root approximation

    # check if a and b bound a root
    if np.sign(f(a)) == np.sign(f(b)):
        raise Exception(
         "The scalars a and b do not bound a root")

    # get midpoint
    m = (a + b)/2

    if np.abs(f(m)) < tol:
        # stopping condition, report m as root
        return m

    elif np.sign(f(a)) == np.sign(f(m)):
        # case where m is an improvement on a.
        # Make recursive call with a = m
        return my_bisection(f, m, b, tol)

    elif np.sign(f(b)) == np.sign(f(m)):
        # case where m is an improvement on b.
        # Make recursive call with b = m
        return my_bisection(f, a, m, tol)
```

III.

```
f1 = lambda x:x**2-np.exp(-x)-2*np.sin(x)  #Lambda says that x should be found
print("a.) root = ",my_bisection(f1, 0, 2, 0.01))

f2 = lambda t : (10*t)+(1/2*(-9.8)*t**2)
print("b.) time taken = ",my_bisection(f2, 1, 5, 0.01))

f3 = lambda q: q**2-3
print("c.) root of 3 = ",my_bisection(f3, 1, 2, 0.01))

f4 = lambda L: L**2 - ((9.8*10*10)/(4*np.pi*np.pi))**2
print("d.) L = ",my_bisection(f4, 24, 27, 0.01))

f5 = lambda r:r**2-(np.log(2)*np.log(2))
print("e.) r = ",my_bisection(f5, 0, 1, 0.001))
```

```
a.) root =  1.4921875
b.) time taken =  2.041015625
c.) root of 3 =  1.734375
d.) L =  24.8236083984375
e.) r =  0.693359375
```

2. Newton Raphson method

I.

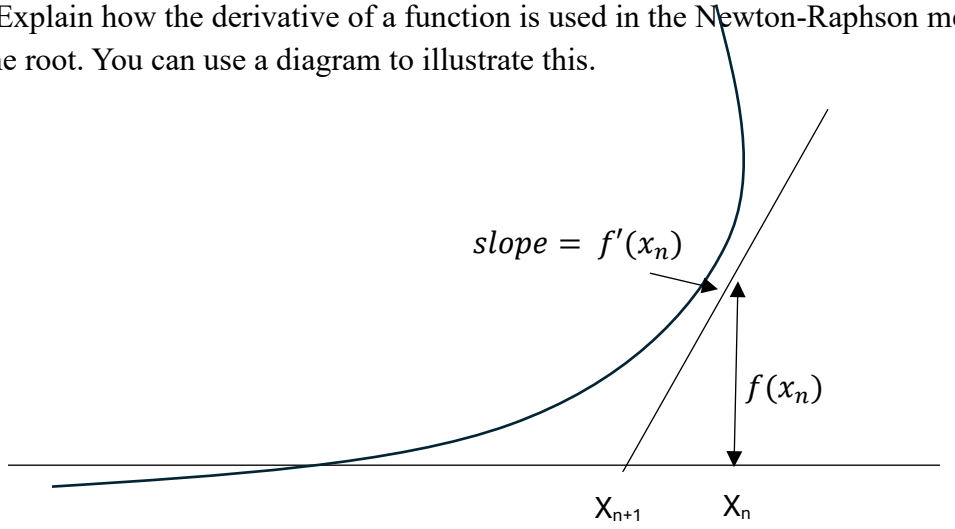
a.) What is the formula used to update the approximation in each iteration of the Newton-Raphson method?

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

b.) How does the choice of the initial guess affect the convergence of the Newton Raphson method?

If the initial guess is close enough to the solution, it will increase the efficiency of computation. It may fail if the initial guess is very far from the solution.

c.) Explain how the derivative of a function is used in the Newton-Raphson method to find the root. You can use a diagram to illustrate this.



First a tangent is drawn to the graph $f(x)$ at point $x = x_n$.

The slope $f'(x_n)$ goes through the point $(x_n, f(x_n))$.

Then the root of the tangent is found by setting y=0 and $x = x_{n+1}$ and substitute to the equation $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$

So, we find the derivative of the function at a particular point first.

Then we calculate the value of the first derivative of the function at a particular point.

Then we relate the first derivative of the function at a specific point to the slope of the tangent line

Finally, we find the tangent line crossing the x axis.

d.) Under what conditions does the Newton-Raphson method fail to converge or converge slowly?

If the starting value x_0 is too far from the root, the sequence may converge or diverge to a different root.
If the tangent gradient $f'(x)$ is very small, the sequence may converge or diverge very slowly
If the function is not continuous or not continuously differentiable near the root, the method may fail.
If the derivative of the function is zero, the method will fail to converge to the function's zero.
If the function is complex and it's difficult to derive a simple expression for its derivative, the method may fail.
If the equation is difficult or impossible to differentiate, the method may fail.

II.

```
import numpy as np

def my_newton(f, df, x0, tol, max_itr):
    #f (function) - The function for which the root is to be found
    #df (function) - The derivative of the function
    #x0 (float) - The initial guess of the root
    #max_itr (int) - Maximum number of iterations
    #tol (float) - The tolerance level for the root approximation

    for i in range(max_itr):
        if abs(f(x0)) < tol:
            return x0
        else:
            return my_newton(f, df, x0 - f(x0)/df(x0), tol, max_itr)
```

III.

```
print("(III)")
f1 = lambda x:x**2-np.exp(-x)-2*np.sin(x)  #Lambda says that x should be found
df1 = lambda x: 2*x+-np.exp(-x)-2*np.cos(x)
print("a.) root = ",my_newton(f1, df1, 1, 1e-6, 100))

f2 = lambda t : (10*t)+(1/2*(-9.8)*t**2)
df2 = lambda t: 10+((-9.8)*t)
print("b.) time taken = ",my_newton(f2, df2, 2, 1e-6, 100))

f3 = lambda q: q**2-3
df3 = lambda q: 2*q
print("c.) root of 3 = ",my_newton(f3, df3, 1, 1e-6, 100))

f4 = lambda L: L**2 - ((9.8*10*10)/(4*np.pi*np.pi))**2
df4 = lambda L: 2*L
print("d.) L = ",my_newton(f4, df4, 20, 1e-6, 100))

f5 = lambda r:r**2-(np.log(2)*np.log(2))
df5 = lambda r: 2*r
print("e.) t = ",my_newton(f5, df5, 1, 1e-6, 100))
```

```
(III)
a.) root =  1.4895884942090234
b.) time taken =  2.0408163265306736
c.) root of 3 =  1.7320508100147276
d.) L =  24.82368999237277
e.) t =  0.693147180561823
```

3. Python Root Finding Functions

I.

```
import numpy as np
from scipy.optimize import fsolve

print("(I)")
f1 = lambda x:x**2-np.exp(-x)-2*np.sin(x)
print("a.) root = ",fsolve(f1,0)[0])    #f1 is the function and 2 is the initiial condition

f2 = lambda t : (10*t)+(1/2*(-9.8)*t**2)
print("b.) time taken = ",fsolve(f2,2)[0])

f3 = lambda q: q**2-3
print("c.) root of 3 = ",fsolve(f3,1)[0])

f4 = lambda L: L**2 - ((9.8*10*10)/(4*np.pi*np.pi))**2
print("d.) L = ",fsolve(f4,20)[0])

f5 = lambda r:r**2-(np.log(2)*np.log(2))
print("e.) L = ",fsolve(f5,1)[0])
```

```
(I)
a.) root =  -1.1360914521498995
b.) time taken =  2.040816326530673
c.) root of 3 =  1.7320508075688772
d.) L =  24.823689992372756
e.) L =  0.6931471805599456
```

II.

The Newton-Raphson technique converges significantly faster than the bisection method, but it is unstable at times and may not converge. The bisection method is a very stable method of identifying roots, but it converges quite slowly compared to other methods. However, because fsolve is a predefined function, utilizing it needs a lot less amount of work. Using this strategy is therefore more efficient than using the other two. Furthermore, we can state that this approach has a greater accuracy because we tested the other two methods with a tolerance level of 0.000001 and obtained results identical to those of the fsolve method. Due to its ability to handle a larger range of functions and lower likelihood of experiencing convergence problems, the fsolve() approach is often more reliable than the Newton-Raphson method. For the same number of iterations, the fsolve() approach yields better precision and is typically faster than the bisection method in solving the problem.