Department of Mathematics

Instructor Name:

Dr. Umair Umar

Student Name:

Hina Khalil (SP20-BSM-015)

Kinat Altaf (SP20-BSM-020)

Subject:

NUMIRICAL COMPUTATION

➢ BISECTION METHOD:

Bisection method is used to find the roots of the polynomial equation. It separates the interval and subdivides the interval in which the root of the equation lies. This method is based on 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. The bisection method is also known as interval halving method, root finding method, binary search method.

❖ Bisection Method Algorithm:

We are trying to find the roots of the equation f(x)=0

If f(x) is continuous between x_l and x_u, and the function is changing the sign, there exist at least one root between x_l and x_u . 1)Choose x_l and x_u as initial guess such that f (x_l)*f (x_u)<0.

2)Find the midpoint between the two points, say "m" and then it is calculated by taking average of upper and lower values of the guess.

$$x_m = \frac{x_m + x_u}{2}$$

3) find f (x_l)*f (x_m)<0. f (x_l)*f (x_m)>0. f (x_l)*f (x_m)=0.

a) If f (x_l)*f (x_m)<0, then $x_l = x_l$, $x_u = x_m$

b) If f (x_l)*f (x_m)>0, then $x_l = x_m$, $x_u = x_u$

c) If f (x_l)*f (x_m)=0. Then the root is x_m and stop there.

4) Find $x_m = \frac{x_m + x_u}{2}$

Now find the tolerance , $|\epsilon_a| = \left|\frac{x_m^{new} - x_m^{old}}{x_m^{new}}\right| * 100$

Check , $|\epsilon_a| \leq |\epsilon_s|$

If this condition satisfied then we are going to do any kind of iterative procedure. We are trying to find the better and better estimation of something. We have got some prespecified tolerance so as to able to figure out when the whole algorithm should stop.

5) otherwise, Go back to step 3.

- **ADVANTAGES AND DRAWBACKS OF BISECTION METHOD:** ❖ **ADVANTAGES:**

1)It is always convergent because as by decreasing the interval size, by changing the new guess as midpoint then interval length keep on diminishes. It keeps on halving as you keep on going from one iteration to other. So, it is always going to find the interval where function is changing sign. So, it is always convergent.

2) Error can be controlled.

❖ **DRAWBACKS:**

1)Convergence is generally slow.

2)We have to perform lots of iteration if we choose the guess close to the root.

3)We cannot find the root of many equations like x^2=0

4)May seek similarity point as root. like f(x)=$\frac{1}{x}$ =o

Example:

$f(x) = x3 - x - 1 = 0$

Initial Guess:

[1,2]

1st **I**teration:

Here

$$f(1) = -1 < 0 \quad and\ f(2) = 5 > 0$$

Now roots lies between 1 and 2

$$c = \frac{a+b}{2}$$

$$c = \frac{1+2}{2} = 1.5$$

$$f(c) = f(1.5) = (1.5)3 - 1.5 - 1 = 0.875 > 0$$

Now roots lie between 1 and 1.5

No	A	f(a)	B	f(b)	C	f(c)	Updated Value
1	1	-1	2	5	1.5	0.875	c=b
2	1	-1	1.5	0.875	1.25	-0.296	c=a
3	1.25	-0.29	1.5	0.876	1.375	0.224	b=c
4	1.25	-0.29	1.37	0.224	1.312	-0.051	c=a
5	1.312	-0.05	1.37	0.224	1.343	0.0826	b=c

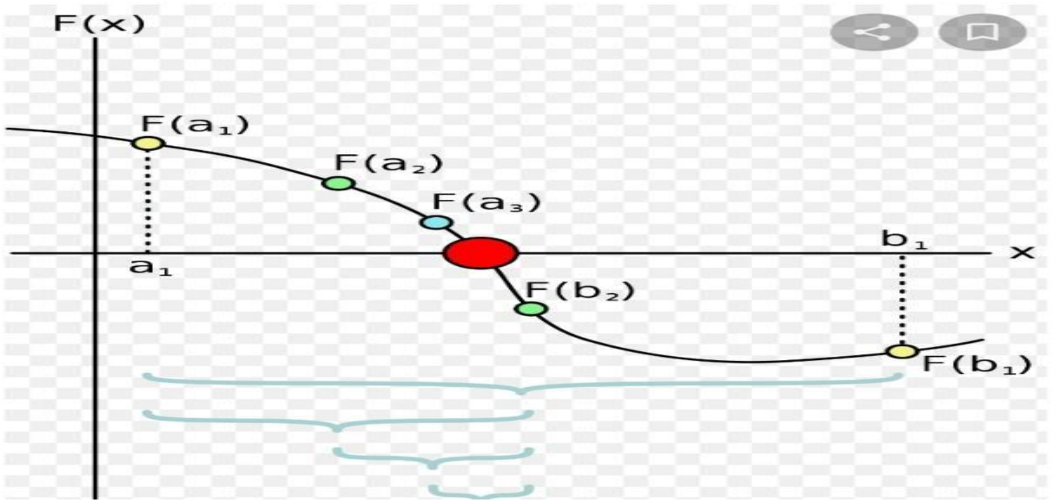
CODE:

```
# Defining Function def
f(x):
    return x**3-x-1


# Implementing Bisection Method def
bisection (x0, x1, e):
    step = 1
    print ('\n\n*** BISECTION METHOD IMPLEMENTATION ***')
condition = True     while condition:         x2 = (x0 + x1)/2
```

```
        print ('Iteration-%d, x2 = %0.6f and f(x2) = %0.6f' % (step, x2, f(x2)))         if f(x0) * f(x2) <
0:
            x1 = x2
else:
            x0 = x2          step =
step + 1         condition =
abs(f(x2)) > e

    print('\required Root is : %0.8f' % x2)



# Input Section x0 = input
('First Guess:1 ') x1 = input
('Second Guess: 2') e = input
('Tolerable Error: ')

# Converting input to float
x0 = float(x0) x1 =
float(x1) e = float(e)

#Note: You can combine above two section like this
# x0 = float (input ('First Guess: '))
# x1 = float (input ('Second Guess: '))
# e = float (input ('Tolerable Error: '))


# Checking Correctness of initial guess values and bisecting if
f(x0) * f(x1) > 0.0:
    print ('Given guess values do not bracket the root.')
print ('Try Again with different guess values.') else:
    bisection (x0, x1, e)
```

➢ FALSE POSITION METHOD:

This is one of the iterative methods that give you the root if the function changes its sign. False position is based on graphical approach. An iterative method that uses the graphical understanding to join f (x_l) and f (x_u)by a straight line. The intersection with x-axis gives an improved version of root. This method is called regular falsie or false position method. It is also known as linear interpolation method. Because it takes the same approach where two points of the function are joined with a straight line.



❖ FALSE POSITION Method Algorithm:

1)Find the root of function f(x).

2) On the interval [x_l,x_u]

F (x_l) and f (x_u)must have different sign.

Somewhere in between must be zero.

3) Compute secant between ((x_l), f (x_l)) and ((x_u), f (x_u)).

4)Find c where secant line crosses the x-axis.

Slope m=$\frac{\Delta y}{\Delta x}$

m1=m2

$$\frac{f(x_l)-f(x_u)}{x_u-x_l}=\frac{0-f(x_u)}{c-x_u}$$

C=x_u-f (x_u).$\frac{x_u-x_l}{f(x_u)-f(x_l)}$

- If f(c) =0 or |f(c) |<$\in$ then stop.
- Else if sign of f (c) =sign of f (x_l) then c=x_l
  - Otherwise, c=x_u

5) Compute the new secant and repeat the process until you find the root.

- **ADVANTAGES AND DRAWBACKS OF FALSE POSITION METHOD:**

  - **ADVANTAGES:**

1)It converges at faster than a linear rate, so that it is more rapidly convergent than the bisection method.
2)It does not require use of the derivatives of function.
3)It require only one function evaluation per iteration.

  - **DRAWBACKS:**

1)It can be very slow.

2)Like bisection method, need an initial interval around the root.

3)There is no error bounds.

False Position Method Example:

Find a root of an $equation\ f(x) = x^4 - 3$ using false position Method.

Solution:

$$Here\ x^4 - 3 = 0$$

$$Let\ f(x) = x^4 - 3$$

First Iteration:

$$a = 1 \quad and\ b = 2$$

$$f(a) = f(1) = -2 < 0 \quad and\ \ f(b) = f(2) = 13 > 0$$

$$c = a - f(a).\frac{b - a}{f(b) - f(a)}$$

$$c = 1 - (-2).\frac{2 - 1}{13 - (-2)}$$

$$c = 1.133$$

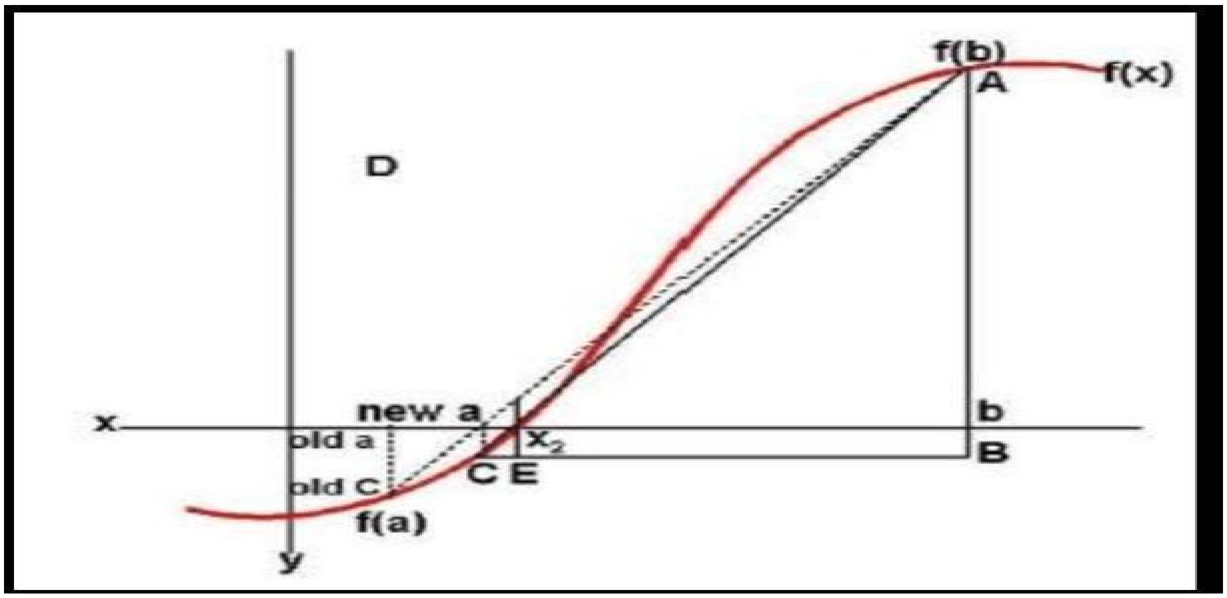
$$f(c) = f(1.133) = (1.133)^4 - 3 = -1.3502$$

No	A	f(a)	B	f(b)	c	f(c)	Updated Value
1	1	-2	2	13	1.133	-1.35	c=a
2	-1.13	-1.35	2	13	1.2149	-0.821	c=a
3	1.214	-0.821	2	13	1.261	-0.467	c=a
4	1.261	-0.467	2	13	1.287	-0.255	c=a

CODE:

```
# Defining Function
def f(x):     return
x**4-3

# Implementing False Position Method def false Position(x0, x1,e):
   step = 1
   print ('\n\n*** FALSE POSITION METHOD IMPLEMENTATION ***')
condition = True     while condition:
      x2 = x0 - (x1-x0) * f(x0)/( f(x1) - f(x0) )         print ('Iteration-%d, x2
= %0.6f and f(x2) = %0.6f' % (step, x2, f(x2)))

      if f(x0) * f(x2) < 0:
         x1 = x2
else:
         x0 = x2

      step = step + 1
condition = abs(f(x2)) > e

   print ('\required root is: %0.8f' % x2)


# Input Section x0 = input
('First Guess: 1') x1 = input
('Second Guess:2 ') e = input
('Tolerable Error: ')
```

```
# Converting input to float
x0 = float(x0)
x1 = float(x1) e
= float(e)

#Note: You can combine above two section like this
# x0 = float(input('First Guess: '))
# x1 = float (input ('Second Guess: '))
# e = float (input ('Tolerable Error: '))


# Checking Correctness of initial guess values and false positioning if
f(x0) * f(x1) > 0.0:
   print ('Given guess values do not bracket the root.')
print ('Try Again with different guess values.') else:
false Position (x0, x1, e)
```



- **NETWON RAPHSON METHOD:**

It is the different method to find the roots of the equation. In this method firstly we differentiate the equation. It uses the idea that a continuous and differentiable function can be approximated by a straight-line tangent to it.

❖ **ALGORITHM FOR NETWON RAPHSON METHOD:**

We are trying to find the root of nonlinear equation f(x)=0.

$$f'_{(x_i)} = \frac{f(x_i)}{x_i - x_i + 1}$$

$$x_i+1 = x_i - \frac{f_{(x_i)}}{f'_{(x_i)}}$$, this is the formula for newton Raphson method.

1)Calculate f'(x).

2)choose an initial guess x_0.

3) $x_i+1 = x_i - \frac{f_{(x_i)}}{f'_{(x_i)}}$

Find $|\in_a| = |\frac{x_{i+1} - x_i}{x_{i+1}}| * 100$

Check | $\in_a$|≤| $\in_s$|, If this condition satisfied then stop the algorithm, if not then repeat the process.

➢ **ADVANTAGES AND DRAWBACKS OF NETWON RAPHSON METHOD:**

❖ **ADVANTAGES:**

1)Converges fast, if it converges. 2)Require only one guess.

❖ **DRAWBACKS:**

1)If the function in this method is divided by zero then this method fails.

2)Root jumping might take place. Like f(x)=sin x=0

We may start from the initial guess which is close to the root, which we are interested in. If nonlinear equation has multiple roots but we might end with any other root which we are not interested in.

Find a root of an $equation\ f(x) = x2 - x - 1$ using Newton Raphson Method.

Solution:

$$Here\ x2 - x - 1$$

$$Let\ f(x) = x2 - x - 1$$

$$We\ know\ that$$
$$f'(x) = 2x - 1$$

$$a = -1\ and\ b = 0$$

$$x_0 = -0.5$$

First Iteration:

$$f(x_0) = f(-0.5) = (-0.5)^2 - (-0.5) - 1 = -0.25$$

$$f'(x_0) = 2(-0.5) - 1 = -2$$

$$x_1 = x_0 - f(x_0)/f'(x_0)$$

$$x1 = -0.5 - \frac{-0.25}{-2}$$

$$x1 = -0.625$$

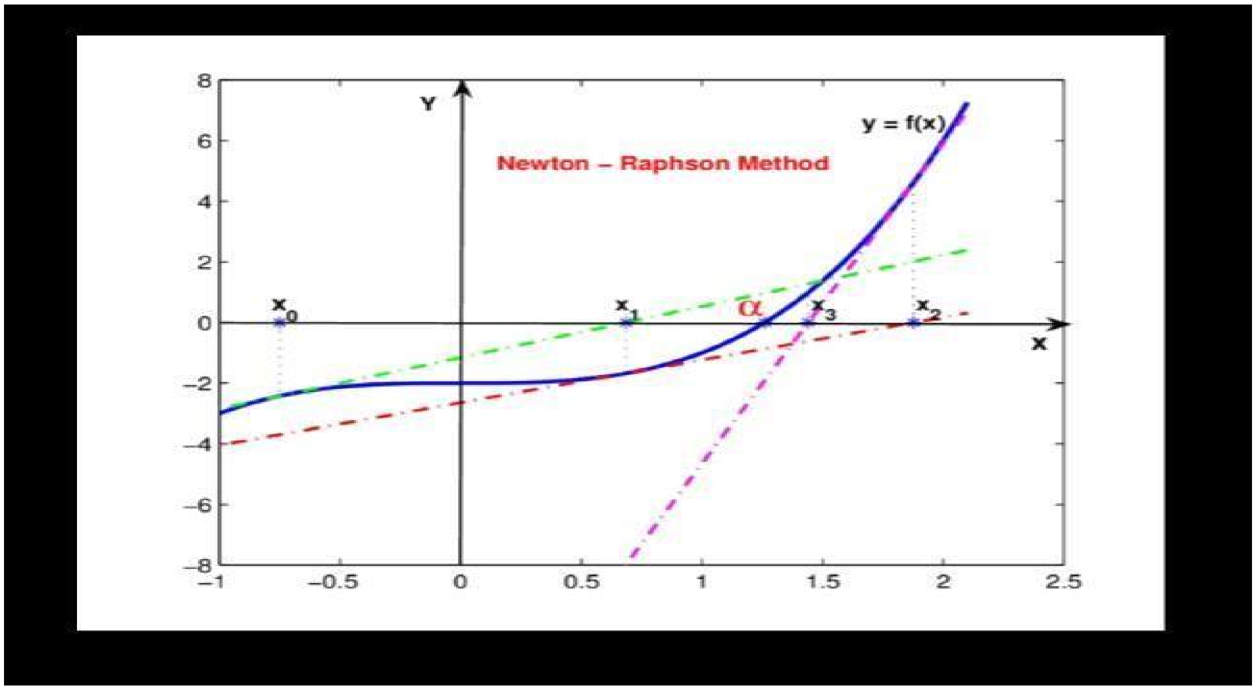
N	x_0	$f(x_0)$	$f'(x_0)$	X_1	Update
1	-0.5	-0.25	-2	-0.625	$X_0=x_1$
2	-0.625	0.0156	-2.25	-0.618	$X_0=x_1$
3	-0.618	0	-2.236	-0.618	$X_0=x_1$

CODE:

```
# Defining Function def
f(x):
   return x**2 - x - 1

# Defining derivative of function
def g(x):     return 2*x - 1

# Implementing Newton Raphson Method def newton Raphson (x0, e, N):
   print ('\n\n*** NEWTON RAPHSON METHOD IMPLEMENTATION ***')
   step = 1
flag = 1
   condition = True
while condition:
if g(x0) == 0.0:
         print ('Divide by zero error!')
         break
```

```
        x1 = x0 - f(x0)/g(x0)         print ('Iteration-%d, x1 = %0.6f and f(x1)
= %0.6f' % (step, x1, f(x1)))
        x0 = x1
step = step + 1

        if step > N:
flag = 0
break

        condition = abs(f(x1)) > e

    if flag==1:
        print ('\nRequired root is: %0.8f' % x1)
    else:
        print ('\nNot Convergent.')

# Input Section
x0 = input ('Enter Guess: ') e
= input ('Tolerable Error: ')
N = input ('Maximum Step: ')

# Converting x0 and e to float
x0 = float(x0) e = float(e)

# Converting N to integer
N = int(N)

#Note: You can combine above three section like this
```

```
# x0 = float (input ('Enter Guess: '))
# e = float (input ('Tolerable Error: '))
# N = int (input ('Maximum Step: '))

# Starting Newton Raphson Method newton
Raphson(x0,e,N)
```



➢ SECANT METHOD:

This method is also used to find the roots of the non-linear equation f(x)=0.

Formula for secant method:

$$x_{i+1}=x_i-\frac{f(x_i)(x_i-x_{i-1})}{f(x_i)-f(x_{i-1})}$$

❖ ALGORITHM:

1)Choose i=1

2)take two initial guesses x_{i-1},x_i

3)use formula

4)Find absolute error by using formula,

$$|\in_a| \leq \left|\frac{x_{i+1}-x_i}{x_{i+1}}\right| * 100$$

Check $|\in_a| \leq |\in_s|$, If true stop the algorithm.

If false go to the step 2 with estimate x_{i+1},x_i here we take i=i+1. Increment in I by i+1 as the new initial guess.

- **ADVANTAGES AND DRAWBACKS OF SECANT METHOD:** ❖ **ADVANTAGES:**

- It converges at faster than a linear rate, so that it is more rapidly convergent than the bisection method.
- It does not require use of the derivative of the function.
- It requires only one function evaluation per iteration, as compared with Newton's method which requires two.

❖ **DRAWBACKS OF SECANT METHOD:**

- There is no guaranteed error bound for the computed iterates.
- It may not converge.

Find a root of an $\boldsymbol{equation\ f(x) = x^3 - x - 1}$ using secent method.

Solution:

$$Here\ x^3 - x - 1 = 0$$

$$let\ f(x) = x^3 - x - 1$$

Here

X	0	1	2
F(x)_	-1	-1	5

- **First Iteration:**

$x_0 = 1 \ \ and \ x_1 = 2$

$F(x_0) = f(1) = -1 \ and \ f(2) = 5$

$x_2 = c = a - f(a).\frac{b-a}{f(b)-f(a)}$

$x_2 = 1 - (-1).\frac{2-1}{5-(-1)}$

$x_2 = 1.166$

$f(x_2) = f(1.166) = -0.5787$

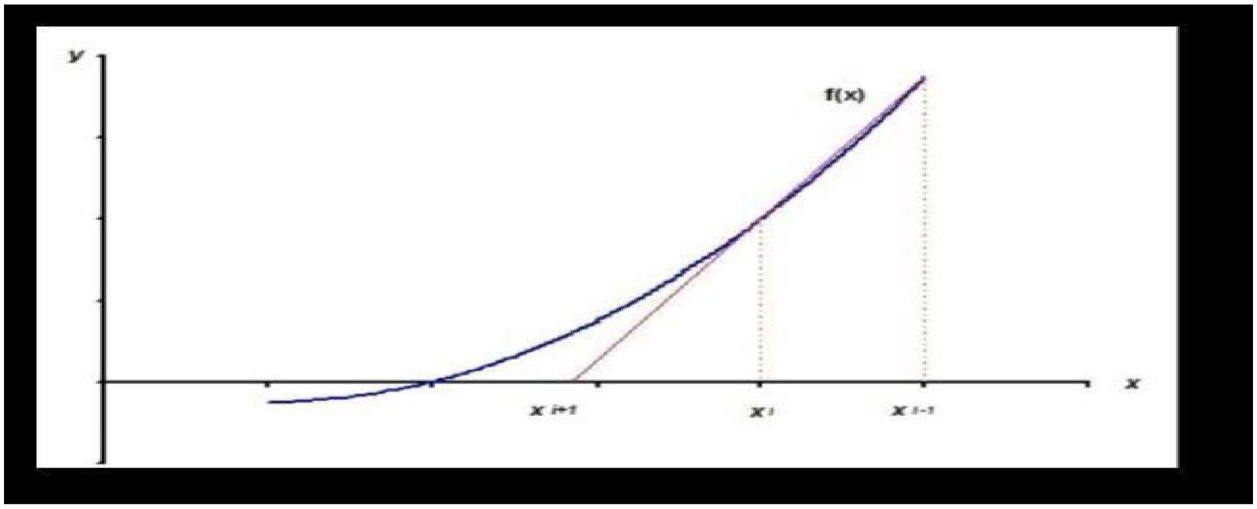
NO	x_0	$f(x_0)$	x_1	$f(x_1)$	x_2	$f(x_2)$	Update
1	1	-1	2	5	1.667	-0.578	$x_0=x_1$ $x_1=x_2$
2	2	5	1.166	-0.578	1.2531	-0.285	$x_0=x_1$ $x_1=x_2$
3	1.166	-0.578	1.253	-0.285	1.337	0.0538	$x_0=x_1$ $x_1=x_2$
4	1.253	-0.285	1.337	0.0538	1.323	-0.003	$x_0=x_1$ $x_1=x_2$
5	1.337	0.0538	1.323	-0.003	1.324	-0.004	$x_0=x_1$ $x_1=x_2$

CODE:

```
# Defining Function def
f(x):
   return x**3 - x - 1

# Implementing Secant Method

def secant (x0, x1, e, N):
   print ('\n\n*** SECANT METHOD
IMPLEMENTATION ***')
   step = 1
condition = True
```

```
while condition:
if f(x0) == f(x1):
print ('Divide by
zero error!')
           break

       x2 = x0 - (x1-x0) *f(x0)/(f(x1) - f(x0))
print ('Iteration-%d, x2 = %0.6f and f(x2) =
%0.6f' % (step, x2, f(x2)))
x0 = x1        x1 = x2
step = step + 1

       if step > N:            print
('Not Convergent!')
           break

       condition = abs(f(x2)) > e     print ('\n
Required root is: %0.8f' % x2)

# Input Section x0 = input ('Enter
First Guess: ') x1 = input ('Enter
Second Guess: ') e = input
('Tolerable Error: ') N = input
('Maximum Step: ')

# Converting x0 and e to float
x0 = float(x0) x1 = float(x1)
e = float(e)

# Converting N to integer
N = int(N)

#Note: You can combine above three section like this
# x0 = float (input ('Enter First Guess: '))
# x1 = float (input ('Enter Second Guess: '))
# e = float (input ('Tolerable Error: '))
# N = int (input ('Maximum Step: '))
```

# Starting Secant Method Secant
(x0, x1, e, N)



➢ FIXED POINT METHOD:

A non-linear equation of the form f(x)=0 can be written to obtain an equation of the form g(x)=x, in which case the solution is a fixed point of the function g. This formulation of the original problem f(x) = 0 will leads to a simple solution method known as fixed-point iteration. This method is used to solve and to find the approximate root of algebraic and transretinal equation.

ALGORITHM FOR FIXED POINT METHOD:

Let we have equation f(x)= 0… (1)

Xo is the initial root of f(x)=0

The root lies between two points at which function changes its sign. We write equation (1) as

X=Φ(x) … (2) Equation one can be written in different form. This is one of the forms.

If $|\frac{d\Phi_x}{dx}|<1$

If it is true at Xo. Then equation (2) can be used as iteration formula.

x_{n+1}=Φ (x_n), n=1,2,3….

We will repeat the process until Xn=x_{n+1}

➢ DRAWBACKS:

It requires a starting interval containing a change of sign. Therefore, it cannot find repeated roots. It has a fixed rate of convergence, which can be much slower than other method, requiring more iterations to find the root to a given degree of precision.

Find a root of an $\boldsymbol{equation\ f(x)\ =\ 2x3 - 2x - 4}$ using Fixed Point Iteration Method.

X	0	1	2
f(x)	-4	-4	8

$$Here\quad f(1) = -4 < 0 \quad and\ f(2) = 8 > 0$$

$$Roots\ are\ lies\ between\ 1\ and\ 2$$

$$x_0 \;=\frac{1+2}{2} =\; 1.5$$

$$x_1 \;=\; \phi\ (x0) = 1.518$$

No	x_0	$x_1=\phi(x_0)$	Update	Difference x_1-x_0
2	1.5	1.5183	$x_0=x_1$	0.0183
3	1.518	1.5209	$x_0=x_1$	0.0026
4	1.520	1.5213	$x_0=x_1$	0.0004
5	1.531	1.5331	$x_0=x_1$	0.0005

CODE:

# Fixed Point Iteration Method #

Importing math to use sqrt function

import math

```
def f(x):
   return 2X**3 + 2*x -4

# Re-writing f(x)=0 to x = g(x) def
g(x):
   return 1/math.sqrt(1+x)

# Implementing Fixed Point Iteration Method def
fixed Point Iteration (x0, e, N):
   print '\n\n*** FIXED POINT ITERATION ***')
step = 1     flag = 1     condition = True

   while condition:         x1 = g(x0)         print 'Iteration-%d, x1 = %0.6f
and f(x1) = %0.6f' % (step, x1, f(x1)))         x0 = x1

      step = step + 1

      if step > N:
flag=0             break

      condition = abs(f(x1)) > e

   if flag==1:
      print ('\nRequired root is: %0.8f' % x1)
else:
      print ('\nNot Convergent.')
```

# Input Section x0 = input

('Enter Guess: ') e = input

('Tolerable Error: ')

N = input ('Maximum Step: ')

# Converting x0 and e to float x0

= float(x0)

e = float(e)

# Converting N to integer

N = int(N)

#Note: You can combine above three section like this

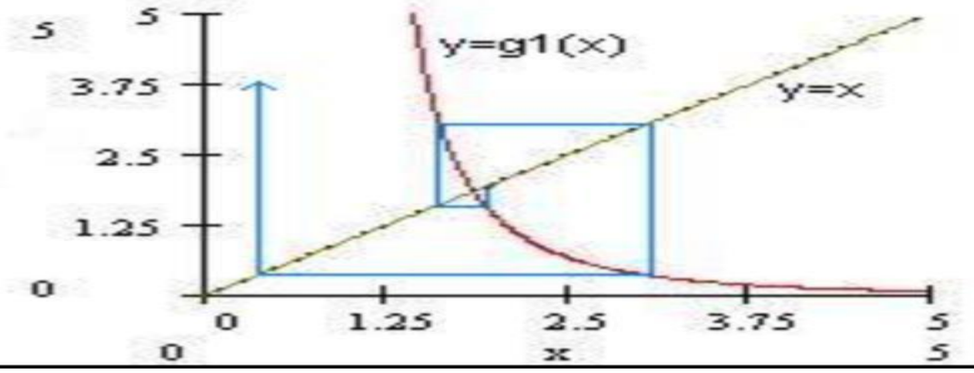
# x0 = float (input ('Enter Guess: '))

# e = float (input ('Tolerable Error: '))

# N = int (input ('Maximum Step: '))

# Starting Newton Raphson Method

Fixed Point Iteration (x0, e, N)



➢ JACOBI METHOD:

It is the method for determining the solutions of the strictly diagonally dominant system of linear equations. Each diagonal element is solved for, and an approximate value is plugged in. This process is iterated until it convergent.

Let

A_x=b be a square system of *n* linear

equations, where:

A=$\begin{bmatrix} a_{11} & a_{12} & \cdots \\ a_{21} & a_{22} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$, X=$\begin{bmatrix} X_1 \\ X_2 \\ X_N \end{bmatrix}$,B=$\begin{bmatrix} B_1 \\ B_2 \\ B_N \end{bmatrix}$

Then A can be decomposed into a diagonal component D, a lower triangular part L and an upper triangular part U.

A=D+L+U, Where D=$\begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{nn} \end{bmatrix}$,L+U=$\begin{bmatrix} 0 & a_{12} & a_{1n} \\ a_{21} & 0 & a_{2n} \\ a_{n1} & a_{n2} & 0 \end{bmatrix}$

The solution is then obtained via,

X^{k+1}=D^{-1}(b-(L+U)X^{K})

Here X^K is the K^{th} iteration. And X^{k+1} is k+1 iteration on x.

❖ ALGORITHM:

1) Start

2)Arrange the given system of equation in diagonally dominant form.

3)Convert the first equation in term of first variable, and second equation in term of second variable and so on.

4) Set initial guess as x_0, y_0, z_0 and so on.

5)Substituting the value of x_0, y_0, z_0,… from step 4 in step 3, to calculate the new value of x_1 y_1 z_1 , , and so on.

6)Calculate error.

7) If |x_0-x_1|> error and |y_0-y_1|> error and |z_0-z_1|>error and so on then go to step. 8) Set x_0=x_1, y_0=y_1, z_0= z_1 and so on and go to the step 5 9)Print values of x_1, y_1, z_1. 10)Stop

ADVANTAGES AND DRAWBACKS OF JACOBI METHOD:

❖ ADVANTAGES

Fallowing are the advantages of Jacobi method:

1) It is the simple method.

2)Each iteration quite fast.

3)It is highly desirable for many applications.

4) This method has applications in Engineering also as it is one of the efficient methods for solving systems of linear equations, when approximate solutions are known. This significantly reduces the number of computations required.

❖ DRAWBACKS OF JACOBI METHOD:

1) The method may not always converge on the set of solutions.

2) Jacobian problems and solutions have many significant disadvantages, such as low numerical stability and incorrect solutions, particularly if diagonal entries are small.

Jacobi Method Example:

$$8x + y - z = 8$$

$$x - 7y + 2z = -4$$

$$2x + y + 9z = 12$$

Now

$$x = \frac{1}{8}[8 - y + z]$$

$$y = \frac{1}{7}[4 + x + 2z]$$

$$z = \frac{1}{9}[12 - 2x - y]$$

$$x^k = \frac{1}{a_{11}}[b_1 - a_{12}y^{k-1} + a_{13}z^{k-1}$$

$$y^k = \frac{1}{a_{22}}[b_2 - a_{21}x^{k-1} + a_{23}\, z^{k-1}$$

$$z^k = \frac{1}{a_{23}}[b_3 - a_{31}x^{k-1} + a_{32}y^{k-1}$$

First Iteration:

$$x^0 = 0, \qquad y^0 = 0, \;\; z^0 = 0$$

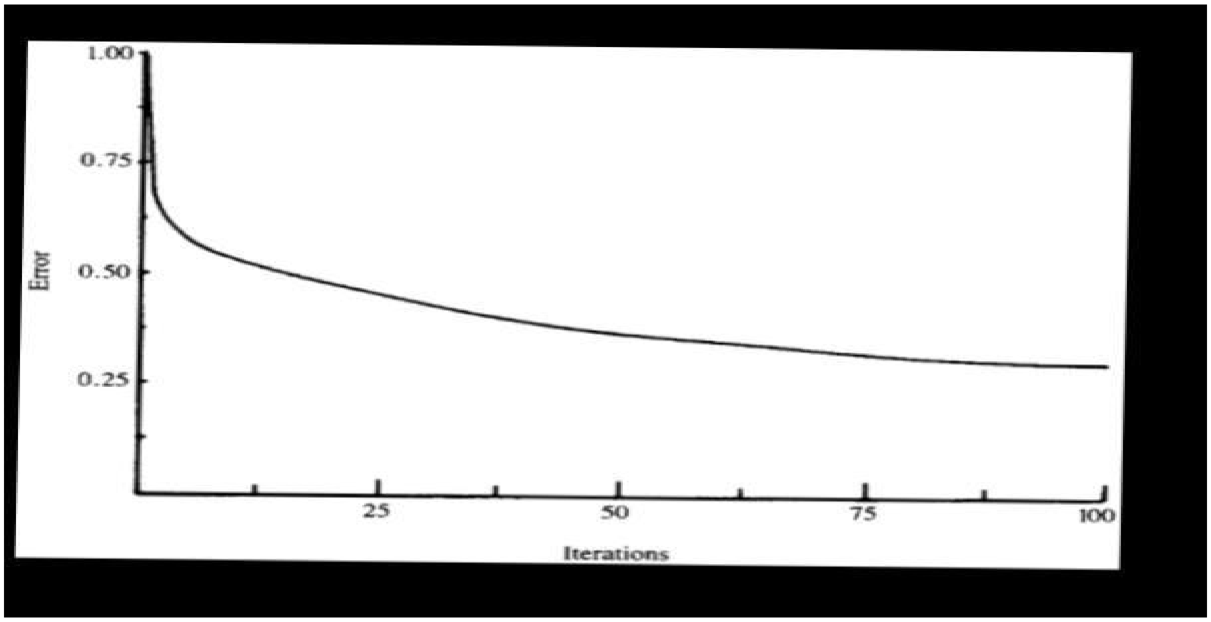
$$x^1 = \frac{1}{8}[8 - 0 + 0] = 1$$

$$y^1 = \frac{1}{7}[4 + 0 + 2(0) = 0.571$$

$$z^1 = \frac{1}{9}[12 - 2(0) - 0] = 1.333$$

No	x	y	z
1	$x^1 = 1$	$y^1 = 0.571$	$z^1 = 1.33$
2	$x^2 = 1.09$	$y^2 = 1.795$	$z^2 = 1.047$
3	$x^3 = 0.994$	$y^3 = 1.027$	$z^3 = 0.968$
4	$x^4 = 0.993$	$y^4 = 0.990$	$z^4 = 0.998$

Code:



➢ GAUESS SEIDEL METHOD:

This method is the improvement of Jacobi method. This method is used to find the simultaneous linear equations. We use this method because; 1) It may be computationally more efficient for large n.

2) Can control roundoff error

A simultaneous linear equation can be written as;

$$[A]_{n\times n}\,[X]_{n\times 1} = [C]_{n\times 1}$$

We have to find our X"s. We can write down the first equation in regular and expended form.

$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = c_1$

$x_1 = \frac{c_1 - a_{12}x_2 - \cdots - a_{1n}x_n}{a_{11}}$

ith equation:(1≤i≤n)

$a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{ii}x_i + \cdots + a_{in} = c_i$

$X_i = \frac{c_i - \Sigma a_{ij}x_j}{a_{ii}}$

❖ ALGORITHM:

1) Rewrite each equation.

$X_i = \frac{c_i - \Sigma a_{ij} x_j}{a_{ii}}$ …(A)

2)Assume an initial solution.

$[x_1^0, x_2^0, \dots, x_n^0]$,

3)Substitute the solution in equation (A), but use the most recent value.

4) Continue iterating till $|\epsilon_a| \leq |\epsilon_s|$ For

ith iteration error will be

$|\epsilon_a| = \left|\frac{x_i^{new} - x_i^{old}}{x_i^{old}}\right| \times 100$, i=1,2, 3…, n

Find max $|\epsilon_a^i|$, i=1…, n

Check $[\max |\epsilon_a^i|] \leq |\epsilon_s|$

❖ ADVANTAGES AND DRAWBACKS:

ADVANTAGES:

1) Computation time per iteration is less.

2)It has linear convergence characteristics.

3)Less memory requirements.

4) The fact that the same array can be used in the Gauss–Seidel method to store both previous and current iteration values is an additional advantage of the Gauss–Seidel method over the Jacobi method.

DRAWBACKS:

1) Requires large number of iterations to reach converge.

2) Not suitable for large systems.

3) Convergence time increases with size of the system.

$$x + 2y + z = 13$$

$$x + 3y + z = 13$$

The coefficient of matrix is not diagonally dominant, Hence we re arrange the equation as follows, such that the elements in coefficient matrix are diagonally dominant.

$$x + y + z = 7$$

$$x + 3y + z = 13$$

$$x + 2y + 2z = 13$$

First iteration

$$x = [7 - 0 - 0] = 7$$

$$y = \frac{1}{3}[13 - 7 - 0] = 2$$

$$z = \frac{1}{2}[13 - 7 - 2] = 1$$

No	x	y	Z
1	7	2	1
2	4	2.66	1.833
3	2.5	2.88	2.631
4	1.75	2.96	2.66

Code:

```
import NumPy as np
A= np. array ([4.,-1.,1.,-1.,4.,-2.,1.,-2.,4.]).reshape(3,3)
B= np. array ([12., -1.,5.]). Reshape (3,1)
N= Len(B)
print (N)
x=np. zeros(N)
xold= np. array([10.,10.,10.]).reshape(3,1)
Tol = 0.01
x=np. array ([5.,5.,5.]). reshape (3,1)
#For i in range (N):
#    print ("start at i= “, i, "and xi=“, x[i])
temp=0
while (abs(x[0]-xold[0])>Tol):
   xold=x
   print ("abstract=“, abs(x-xold))
```

```
for i in range (N):
   for j in range (N):
      if j! = i:
         temp+=A[ij]*x[j]
   #print ("temp=“, temp)
   x[i]=(1/A[i,i])*(B[i]-temp)
   #Print ("end at x", i,"=", x[i])
   #Print ("abs= “, abs(x-xold))
print (x)
print (xold)
```

❖ SOR METHOD:

SOR is one of the most important methods for solution of large linear system equations. This method is used to speed up the convergence.

Let the given system of equation,

AX=B … (1)

The coefficient of system can be expressed as

A=D-L-U

Here‟

„D‟ is the diagonal part of „A‟

„L‟ is the strictly lower triangular part of „A‟

„U‟ is the strictly upper triangular part of „A‟

$$A=\begin{bmatrix} a_{11} & a_{12} & \cdots \\ a_{21} & a_{22} & \ldots \\ \vdots & \vdots & \ddots \end{bmatrix}, X=\begin{bmatrix} X_1 \\ X_2 \\ X_N \end{bmatrix}, B=\begin{bmatrix} B_1 \\ B_2 \\ B_N \end{bmatrix}$$

$$D=\begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{nn} \end{bmatrix}, L=\begin{bmatrix} 0 & 0 & 0 \\ a_{21} & 0 & 0 \\ a_{n1} & a_{n2} & 0 \end{bmatrix}, U=\begin{bmatrix} 0 & a_{12} & a_{1n} \\ 0 & 0 & a_{2n} \\ 0 & 0 & 0 \end{bmatrix}$$

The system of linear equation can be written as,

$(D+\omega l)\, x=\omega b-[\omega u+(\omega-1)\, D]x$

$x^{k+1}=(D+\omega l)^{-1}(\omega b-[\omega u+(\omega-l)\, D]x^{k})=L_{\omega}+c$

$x_i^{k+1}=(1-\omega)x_i^{k}+\frac{\omega}{a_{ii}}(b_i-\Sigma_{j<i}a_{ij}x_j^{k+1}-\Sigma_{j>i}a_{ij}x_j^{k})$, i=1,2,3…n

The choice of relaxation factor ω is not necessarily easy, and depends upon the properties of coefficient matrix. If A is symmetric and positive define then $\rho\,(l_{\omega})<1$ for $0<\omega<2$

❖ ALGORITHM:

Inputs: A, b, ω

Output: the approximate solution $x_1,\ldots,x_n$

```
Step 1:      Set k=1 Step 2:

      While (k≤N) do step 3-6   Step
3:
      For i= 1…, n
     Set $x_i=(1-\omega)\, X0_i+\frac{\omega}{a_{ii}}(b_i-\Sigma_{j<i}a_{ij}x_j-\Sigma_{j>i}a_{ij}XO_j+b_i)$, i=1,2,3…n
Step 4:
        if ||x-X0||< TOL then outputs ( $x_1,\ldots,x_n$) STOP
STEP 5:
         Set k= k+1  Step
6:
        For i= 1…, n set X$O_i$  Step
7:
       OUTPUT.
       Stop.
```

ADVANTAGES AND DRAWBACKS OF SOR METHOD:

ADVANTAGES:

1) Convergence is faster than GAUSS-SEIDEL METHOD.

Succesive Over Relaxation

$$5x + y = 10$$

$$2x + 3y = 4$$

$$X = (1 - w).xk \ + 1 \div 5(10 - yk)$$

$$yk + 1 = (1 - w)yk + w.1 \div 3(4 - 2xk + 1)$$

Initial guess (x,y)=(0,0) and 1.25

1st iteration

$$x = (1 - 1.25).0 + 1.25\ 1(1 - 1.25).0 + 1.25\ 1 \div 5[10 - 0] = 0 + 0.25 = 2.5$$

$$y = (1 - 1.25).0 + 1.25.1 \div 3[4 - 2(2.5)] = -0.416$$

Iteration	x	Y
1	2.5	-0.4167
2	1.9792	0.1215
3	2.0086	-0.0094
4	1.9991	-0.0049
5	1.9997	0.002

CODE:

```
import NumPy as np
import math

x = np.array([[ 2.5, 1.9792, 2.0086,1.9991,1.9997]])
b = np. Array ([-0.4167 , 0.1215, -0.0094, -0.0049 , 0.002]])
```

```
x0 = np. Array ([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
tol = 10 ** (-15)
mixite = 20
w = 1.5

def SOR (A, b, x0, tol, mixite, w):
   if (w<=1 or w>2):
      print ('w should be inside [1, 2)');
      step = -1;
      x = float('nan')
return
   n = b.shape
   x = x0

   for step in range (1, max_iter):
      for i in range(n[0]):
         new_values_sum = np.dot(A[i, 1 : (i - 1)], x[1 : (i - 1)])
         for j in range(i + 1, n[0]):
            old_values_sum = np.dot(A[i, j], x0[j])
         x[i] = b[i] - (old_values_sum + new_values_sum) / A[i, i]
         x[i] = np.dot(x[i], w) + np.dot(x0[i], (1 - w))

      if (np.linalg.norm(x - x0) < tol):
         print(step)
         break

     x0 = x
```

❖ FINITE DIFFERENCE:

A **finite difference** is a mathematical expression of the form $f(x + b) - f(x + a)$. If a finite difference is divided by $b - a$, one gets a difference quotient. The approximation

of derivative by finite differences plays a central role in finite difference method for the numerical solution of differential equations, especially boundary value problems. In numerical analysis, finite differences are widely used for approximating derivative, and the term "finite difference" is often used as an abbreviation of "finite difference approximation of derivatives".

BASIC TYPES:

Three basic types are commonly considered:

forward, backward, and central finite differences.

❖ FORWARD FINITE DIFFERENCE:

A forward difference, denoted $\Delta_h[f]$ of a function f is defined as,

$\Delta_h[f](x) = f(x+h) - f(x)$

Depending on the application, the spacing h may be variable or constant. When omitted, h is taken to be 1; that is, $\Delta_1[f](x) = f(x+1) - f(x)$.

• ALGORITHM:

We use this method to find the derivative of the function. We have to choose a point Δx ahead, so that means that the distance between the two points is Δx. by definition,

$$f'(x) = limit_{\Delta x \to 0} \frac{\mathbf{f(x+\Delta x)-f(x)}}{\Delta_x}$$

In forward difference,

$f'(x) \approx \frac{\mathbf{f(x+\Delta x)-f(x)}}{\Delta_x}$, here Δx is the finite number.

• Advantages:

Forward difference method is defined by the slope of secant line between current data value and future data value as approximation of the first order derivative. This is useful for single step prediction. And the truncation error is of first order.

❖ BACKWORD FINITE DIFFERENCE:

First order backward difference of 'y'is the change in 'y' when x is decreased by a positive difference 'h'.

$$\nabla f(x) = f(x) - f(x-h)$$

ALGORITHM:

Take two points on the graph. subtract point of interest from a point behind the point of interest.

$$\nabla f(x) = f(x) - f(x-h)$$

ADVANTAGES:

Backward difference method is defined by the slope of secant line between previous data value and current data value as approximation of the first order derivative. This is useful when future data isn't available. This method too has a truncation error of first order.

❖ CENTRAL FINITE DIFFERENCE:

It is the average of both forward and backward finite difference.

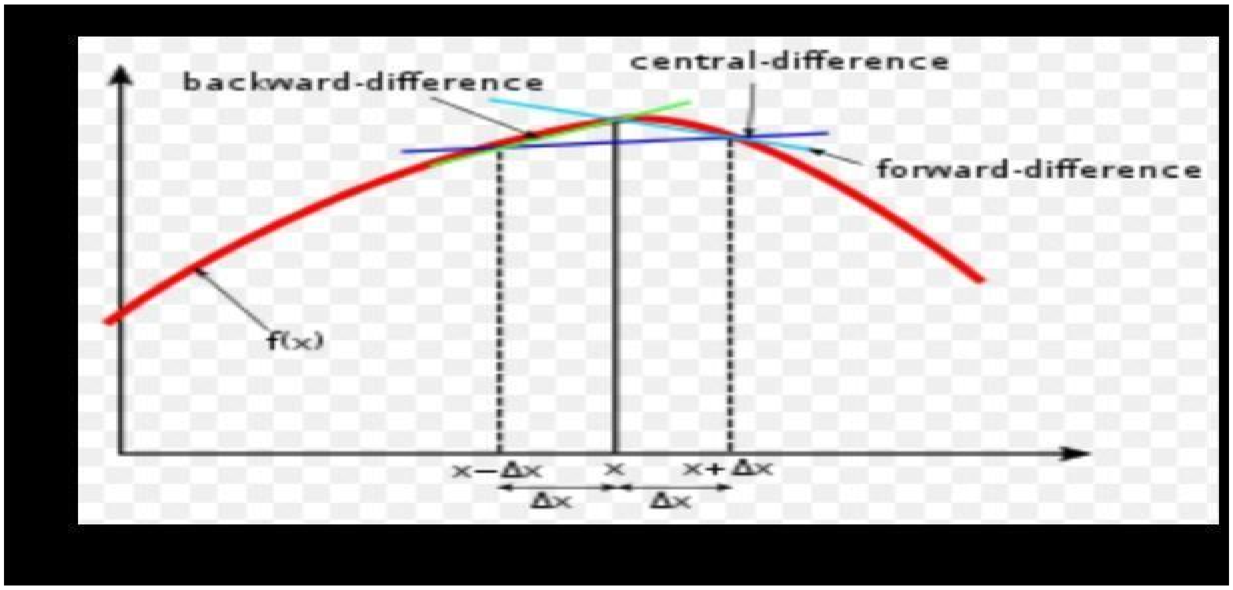
δf(x)=f(x+$\frac{h}{2}$)-f(x-$\frac{h}{2}$)

Algorithm:

δf(x)=f(x+$\frac{h}{2}$)-f(x-$\frac{h}{2}$)

Advantages:

Central difference method is equivalent to the average of forward and backward difference method when the data points are equally spaced. This method gives a truncation error of second order which provides more accuracy in approximation of the first derivative.



INTERPOLATION:

Interpolation can be described as the mathematical procedure applied in order to derive value in between two points having a prescribed value. In simple words, we can describe it as a process of approximating the value of a given function at a given set of discrete points. It can be applied in estimating varied concepts of cost, mathematics, statistics, etc.

Interpolation can be said as the method of determining the unknown value for any given set of functions with known values.

NETWON FORWORD INTERPOLATION:

Newton's forward interpolation formula contains y0 and the forward differences of y0. This Formula is used for interpolating the values of y near the beginning of a set of tabulated values and extrapolation the values of y a little backward (i.e., to the left) of y0.
The differences y1 – y0, y2 – y1, y3 – y2, ……, yn – yn–1 when denoted by dy0, dy1, dy2, ……, dyn–1 is respectively, called the first forward differences. Thus, the first forward differences are:

$\Delta Y_r = Y_{r+1} - Y_r$

FARMULA FOR NEWTON FORWORD INTERPOLATION:

$Y_r = Y_0 + r\Delta Y_0 + \frac{r(r-1)}{2!}\Delta^2 Y_0 + \frac{r(r-1)(r-2)}{3!}\Delta^3 Y_0 + \dots$

Here $r = \frac{x - x_0}{h}$

Newton Forward Interpolation:

x	100	150	200	250	300	350	400
y	10.63	13.03	15.04	16.81	18.42	19.90	21.27

Solution:

The difference table is as under:

x	y	Δ	Δ^2	Δ^3	Δ^4
100	10.63				
		2.40			
150	13.03		-0.39		
		2.01		0.15	

200	15.04		-0.24		-0.07
		1.77		0.08	
250	16.81		-0.16		-0.05
		1.61		0.03	
300	18.42		-0.13		-0.01
		1.48		0.02	
350	19.90		-0.11		
		1.37			
400	21.27				

If we take $x_0 = 160$, then $y_0 = 13.03$,$\Delta y_0 = 2.01, \Delta^2 y_0 = -0.24$

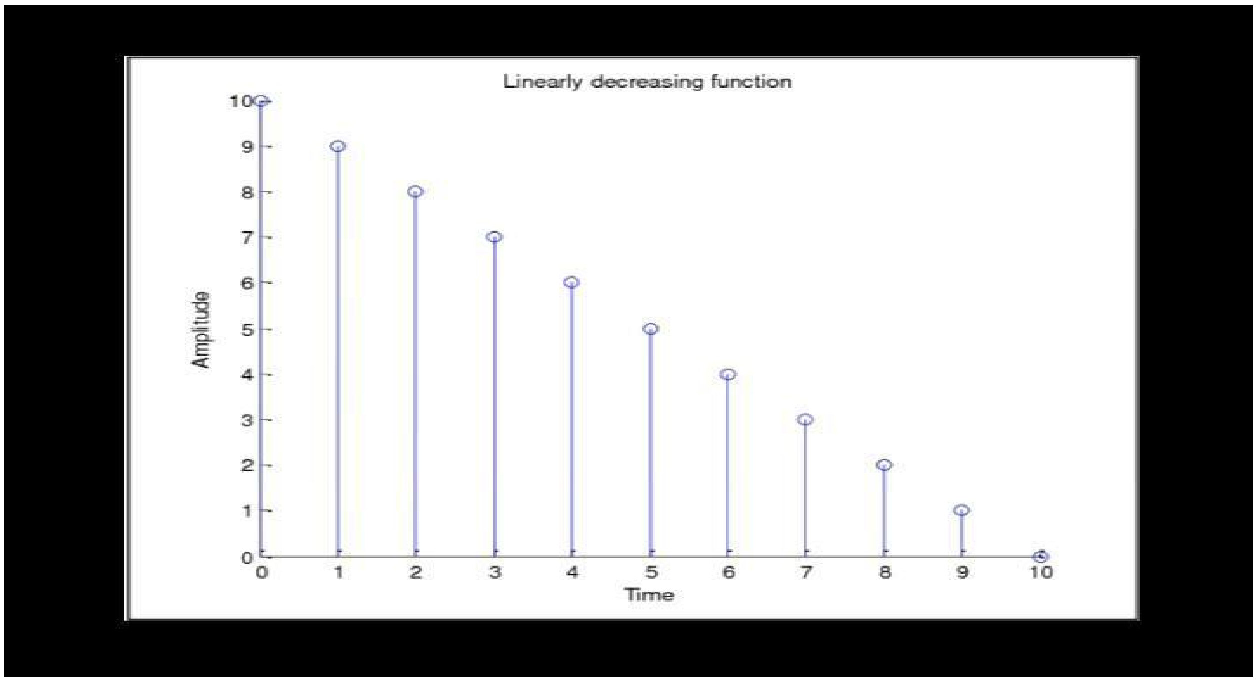
Since x=160 and h=50

$$p = \frac{x - x_0}{h} = \frac{10}{50} = 0.2$$

Using Newton's Forward Interpolation formula, we get

$$y = y_p = y_0 + p\Delta y_0 + \frac{p(p-1)}{2!}\Delta^2 y_0 + \frac{p(p-1)(p-2)}{3!}\Delta^3 y_{0\ldots}$$

$$y = 13.03 + 0.402 + 0.192 + 0.0384 + 0.00168 = 13.46$$



NETWON BACKWORD INTERPOLATION:

The differences y1 – y0, y2 – y1, ……, yn – yn–1 when denoted by dy1, dy2, ……, dyn, respectively, are called first backward difference. Thus, the first backward differences are: ∇Y_r=Y_r-Y_{r-1}

FARMULA FOR NEWTON BACKWORD DIFFERENCE:

Y_n=Y_n+r∇Y_n+$\frac{r(r+1)}{2!}\nabla^2 Y_n$+$\frac{r(r+1)(r+2)}{3!}\nabla^3 Y_n$+…

Here r=$\frac{x-x_0}{h}$

Newton Backward Difference Interpolation:

Given $sin45° = 0.7071$ $sin50° = 0.7660, sin55° = 0.8192$ and $sin60° = 0.8660$. Find $sin57°$ by using an appropriate interpolation formula.

We have to find the value of $f(x) = sinx$ at $x = 57°$ which is near the end value is $x = 60$ using Newton backward Interpolation Formula.

x	y	∇y	$\nabla^2 y$	$\nabla^3 = y$
45	0.7071	0.0589	-0.0057	-0.0007
50	0.7660	0.0532	-0.0064	
55	0.8192	0.0468		
60	0.8660			

We have Newton backward interpolation formula

$$y_p = y_n + p\nabla y_n + \frac{p(p+1)}{2!}\nabla^2 y_n + \frac{p(p+1)(p+2)}{3!}\nabla^3 y_n \ldots$$

$$f(57) = 0.8660 + (-0.6)(0.0468) + \frac{(-0.6)(-0.6+1)}{2}(-0.0064)$$

sin57= 0.8387

NETWON CENTRAL DIFFERENCE:

Y(x)=Y_0+ ($\frac{\Delta y_0+\Delta y_{-1}}{2}$)× p+$\frac{p^2}{2!}\Delta^2 y_{-1}$+… here p=$\frac{x-x_o}{h}$

Use central interpolation to find "y" for x=9 from following:

x	0	4	8	12	16
y	14	24	32	35	40

Solution:

The central difference table for given data is :

x	$f(x)$	$\delta f(x)$	$\delta^2 f(x)$	$\delta^3 f(x)$	$\delta^3 f(x)$
0	14				
		10			
4	24		-2		
		8		-3	
8	32		-5		10
		3		7	
12	35		2		
		5			
16	40				

LAGRANGE INTERPOLATION:

Lagrange interpolation is the way of crafting a function from a set of data pairs. The resulting function passes through all the data points. So, we can use to find the function to interpolate between these points. In other words, interpolation is the technique to estimate the value of a mathematical function, for any intermediate value of the independent variable.

ALGORITHM:

Here we can apply the Lagrange's interpolation formula to get our solution. The Lagrange's Interpolation formula:
If, y = f(x) takes the values y0, y1, …, yn corresponding to x = x0, x1, …, xn then,

The N-th order formula can be written in the form: f(x)

= f0δ0(x) + f1δ1(x) + ... + fN δN (x)

Lagrange First Order Interpolation Formula:

f(x) = $\frac{x-x_1}{x_0-x_1}$ f0 + $\frac{x-x_0}{x_0-x_1} f_1$

ADVANTAGES OF LAGRANGE INTERPOLATION:

- Used in simultaneous optimization of norms of derivatives of Lagrange polynomials

- The answers for higher order polynomials will be more accurate.
- For higher order polynomials the approximate result converges to the exact solution very quickly.

DISADVANTAGES OF LAGRANGE INTERPOLATION:

It is difficult to deal when the polynomial of higher degree involve. because the number of points increase and we have to find the approximate solution to each point.

Apply Lagrange's formula inversely to obtain a root of the equation $f(x) = 0\ given\ that\ f(30) = -30, f(34) = -13, f(38) = 3$ and $f'(42) = 18$

Solution :

Here $x_0 = 30, x_1 = 34, x_2 = 38, x_3 = 42$ and $y_0 = -30\ , y_1 = -30, y_2 = 3, y_3 = 18$ it is required to find $x$corresponding to $y = f(x) = 0$.

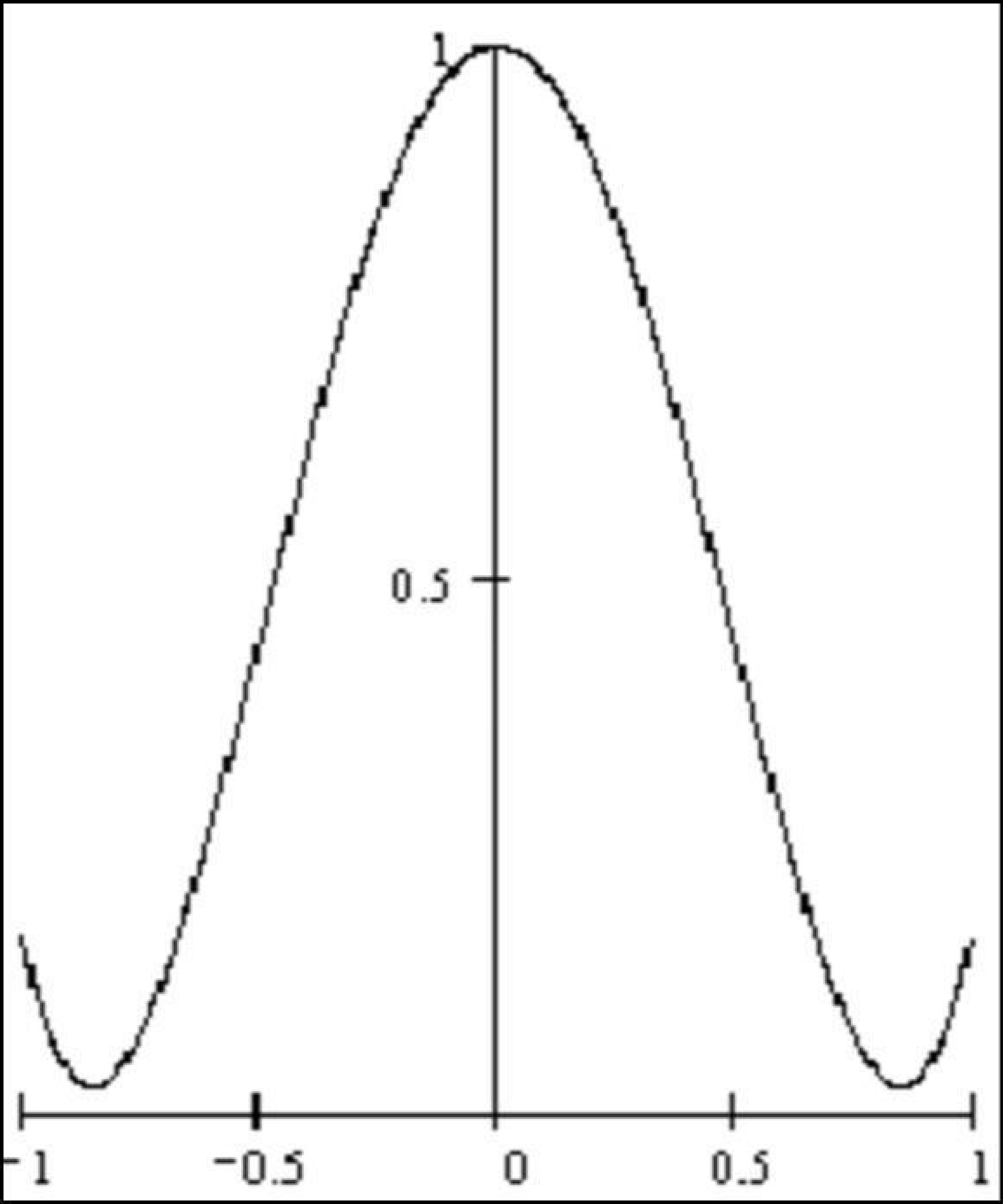
Taking y=0 Lagrange's formula gives

$$x = \frac{(y - y_1)(y - y_2)(y - y_3)}{(y_0 - y_1)(y_0 - y_2)(y_0 - y_3)} x_0 + \frac{(y - y_0)(y - y_2)(y - y_3)}{(y_1 - y_0)(y_1 - y_2)(y_1 - y_3)} x_2 \dots$$

$$x = \frac{13(-3)(-18)}{(-17)(-33)(-48)} \times 30 + \frac{30(-3)(-18)}{17(-16)(-31)} \times 34 + \frac{30(13)(-18)}{33(16)(-15)} \times 38 + \frac{30(13)(-3)}{48(31)(15)} \times 42$$

$$x = -0.782 + 6.532 + 33.682 - 2.202 = 37.23$$

Hence the desired root of $f(x) = 0\ is\ 37.23$


1
0.5
-1
-0.5
0
0.5
1

NEWTON DIVIDED DIFFERENCE:

Newton's divided difference interpolation formula is an interpolation technique used when the interval difference is not same for all sequence of values.

Suppose $f(x_0)$, $f(x_1)$, $f(x_2)$ ………$f(x_n)$ be the (n+1) values of the function y=f(x) corresponding to the arguments $x=x_0, x_1, x_2 \ldots x_n$, where interval differences are not same. Formula for newton divided difference is given as,

The kth degree polynomial approximation to f(x) can be written as

$f(x) = f[x_0] + (x - x_0) f[x_0, x_1] + (x - x_0)(x - x_1) f[x_0, x_1, x_2]$

$+ \ldots + (x - x_0)(x - x_1) \ldots (x - x_{k-1}) f[x_0, x_1, \ldots, x_k]$.

ADVANTAGES:

- These are useful for interpolation.
- Through difference table, we can find out the differences in higher order.
- Differences at each stage in each of the columns are easily measured by subtracting the previous value from its immediately succeeding value.
- The differences are found out successively between the two adjacent values of the y variable till the ultimate difference vanishes or become a constant. Find $f(x)$ as a polynomial in x for the following data by Newton's divided difference formula

x	1	3	4	5	7	10
$f(x)$	3	31	69	131	351	1011

Solution:

We form the divided difference table for the given data.

x	$f(x)$	1st D.D	2nd D.D	3rd D.D	4th D.D	5th D.D
1	3					
		14				
3	31		8			
		38		1		
4	69		12		0	
		62		1		0
5	131		16		0	
		110		1		
7	351		22			
		220				

10	1011					

SPLINE INTERPOLATION:

In the mathematical field of numerical analysis, spline interpolation is a form of interpolation where the interpolant is a special type of piecewise polynomial called a spline.

LINEAR SPLINE INTERPOLATION:

The linear spline represents a set of linear segments between the two adjacent data points. The equation for each linear segment can be immediately found in a simple form.

ALGORITHM:

Linear interpolation is the most basic type of interpolations. It works remarkably well for smooth functions with sufficient number of points. However, because it is such a basic method, interpolating more complex functions requires a little bit more work.

Let we have the data points $(x_0, x_1, \ldots, x_n)$. We have to arrange the data points in ascending order. $x_0 < x_1 < x_2, \ldots, x_n$.

We draw a straight line from one point to the next point, so we have to draw the consecutive data points. Similarly draw the straight line between x_i and x_{i+1} also draw the straight line between x_{i-1}, y_{n-1} and x_n ,y_n and find the value of x at particular y.

General formula:

$f_1(x)=y_i+\frac{y_{i+1}-y_i}{x_{i+1}-x_i}(x-x_i)$, $x_i \le x \le x_{i+1}$

DRAWBACKS:

The drawbacks of linear spline interpolation are that we are finding the value of the function which we are trying to find, is only depend on this straight line. there is no consideration given to the values of the y at any other data point, because we are finding out this linear spline by just taking the values at x=x_0 and x=x_1

Second one is that at interior knot derivative is suddenly changing, derivative is piecewise continuous. even the first derivative is not continuous, because we get a piecewise continuous derivative at interior points

Linear Spline Interpolation:

Example:

T	0	10	20	15	27.5
V	0	227.04	517.35	362.78	602.97

$$v_1(t) = v(15) + \frac{v(20) - v(15)}{20 - 15}(t - 15)$$

$$= 362.78 + \frac{517.35 - 362.78}{20 - 15}(t - 5)$$

$$= 362.78 + 30.913(t - 15)$$

$$15 \le t \le 20$$

$$v_1(16) \approx 362.78 + 30.913(16 - 15)$$

$$= 393.7m/s$$

QUADRATIC SPLINE INTERPOLATION:

Let we have given $(x_0,y_0)\ldots,(x_n,y_n)$ and we conduct quadratic spline interpolation. In quadratic spline interpolation we simply draw a second order polynomial going through the consecutive data points.

$a_1x^2+b_1(x)+c_1$

We have n splines, but in each spline, we have three unknowns. so, that means we have $3n$ unknowns. For this we have to find the $3n$simultaneous equations.

Algorithm:

1) Each spline goes through 2 consecutive data points.

$a_1x_0^2+b_1(x_0)+c_1=y_0$

$a_1x_1^2+b_1(x_1)+c_1=y_1$ $\therefore \{2n\ \ equations\}$

$a_ix_{i-1}^2+b_i(x_{i-1})+c_i=y_{i-1}$

$a_i x_i^2 + b_i(x_i) + c_i = y_i$ ∴i=1,2,3, …

We have already $2n$equation so we have to find $3n$ more equation.

$3n - 2n = n$

n-equations are needed.

2) Each spline has slope at the interior data points. We have got $2n$equations from each spline that is going through each consecutive data points. Then first derivative is continuous at the interior points, and n-1 equations are coming from that the slope is same at the interior points.

(2n) +(n-1) =$3n - 1$

3nth equation:$a_1 = 0$

ADVANTAGES:

1)The derivative is continuous.

2)In order to overcome the problems that we face in linear spline interpolation we find the method of quadratic spline interpolation.

CUBIC SPLINE INTERPOLATION:

Cubic spline interpolation is a way of finding a curve that connects the data points with a degree of three or less. Spline are polynomial that are smooth and continuous across a given plot and also continuous fist and second derivative where they join.

We take a set of points [x_i, y_i] for i = 0, 1, …, n for the function y = f(x). The cubic spline interpolation is a piecewise continuous curve.

- **INTERGRATION:**
  - **Trapezoidal Rule:**

Trapezoidal Rule is a rule that evaluates the area under the curves by dividing the total area into smaller trapezoids rather than using rectangles. This integration works by approximating the region under the graph of a function as a trapezoid, and it calculates the area. This rule takes the average of the left and the right sum.

ADVANTAGES:

In numerical analysis, this rule is a technique for approximating the definite integral. In this rule, the approximation is done on the region under the graph of f(x) as a trapezoid and then the area of that trapezoid is calculated. The result of the trapezoidal rule tends to be more accurate than the other methods.

DRAWBACKS:

One drawback of the trapezoidal rule is that the error is related to the second derivative of the function. More complicated approximation formulas can improve the accuracy for curves - these include using (a) 2nd and (b) 3rd order polynomials.

Trapezoidel Rule

$$\int_2^3 (4u^2 + 6)du, n = 4$$

$$\frac{h}{2}[f(x_0 + 2(fx_1) + 2f(x_2) + 2f(x_3) + f(x_4)]$$

$$\approx \frac{1}{8}\left[f(2) + 2f\left(\frac{9}{4}\right) + 2f\left(\frac{5}{2}\right) + 2f\left(\frac{11}{4}\right) + f(3)\right]$$

$$\approx \frac{1}{8}\left[22 + 2\left(\frac{105}{4}\right) + 2(31) + 2\left(\frac{145}{4}\right) + 42\right]$$

$$\approx \frac{1}{8}\left[22 + \frac{105}{2} + 62 + \frac{145}{2} + 42\right]$$

$$\approx \frac{1}{8}[251]$$

$$\approx 31.375$$

SIMPSON'S RULE:

Simpson's Rule is a numerical method that approximates the value of a definite integral by using quadratic functions. This method is named after the English mathematician Thomas Simpson

Simpson's 1/3 Rule:

Find the approximate value of $I = e^{\frac{1}{2}}$dx using simson's rule.

Solution:

x	1	1.25	1.50	1.75	2.0
y	0.607	0.535	0.472	0.417	0.36

$$[(0.607 + 0.368) + 4(0.535 + 0.417) + 2(0.472)]$$

$$= \frac{0.25}{3} = 0.083[0.975 + 3.808 + 0.944]$$

$$= 0.083 \times 5.727$$

$$= 0.4753$$

Simpson 3/8 rule:

Evaluate I = R 2 1 dx 5 + 3x with 3 and 6 subintervals using Simpson's 3/8 rule. Compare with the exact solution.

Solution:

With N = 3 and 6, we have the following step lengths and nodal points. N = 3, h = b − a N = 1 3 . The nodes are 1, 4/3, 5/3, 2.0. We have the following tables of values.

X	1	4/3	5/3	2.00
f(x)	0.125	0.1111	0.1000	0.9091

Now, we compute the value of the integral.

$I_1 = \int_1^2 \frac{dx}{5+3x} = [f(1.0) + f(2.0) + 3\left\{f\left(\frac{4}{3}\right) + f\left(\frac{5}{3}\right)\right\}]$

= 0.125 [0.125 + 0.09091 + 3 {0.11111 + 0.10000}]

= 0.10616.