Submitted by:

Syeda Sharqa Ghazanfar	SP20 BSM 037
Aimen Abassi	SP20 BSM 004

A Project Report submitted to

Dr. Umair Umer

In partial fulfilment of the requirements for the course of

Numerical Computation

Faculty of Mathematics
Comsats University,
Islamabad

June 22, 2022

Chapter 1

METHOD TO SOLVE NONLINEAR EQUATION

1.1 Bisection method

Bisection method is also called Bolzano method. Bisection method is simplest among all the numerical scheme to solve the transcendental equation. This scheme is based on intermediate value theorem for continuous function.

Bisection method start with two initial guesses says x_l and x_u. consider transcendental equation f(x)=0. Bisection method is based on the fact that if f(x) is real and continuous function and for two initial guesses x_l and x_u bracket the roots such that f(x_l)f(x_u)<0 then there exist at least one root and x_u. if we have two initial guesses than we between $x_l$compute the new root as :

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

we have three cases:

If f(x_u)=0 than root is x_m.

if f(x_l)(x_u)<0 than roots lie between x_l and x_m.

if f(x_l)f(x_u)>0, then root lie between x_u and x_m .

1.1.1 Algorithm

1 . start

2. Define function f(x)

3. Choose initial guesses x_l and $x_u$such that f(x_l)f(x_m)< 0

4. Choose pre specified tolerable error e.

5. calculate new approximated root as x_m = $\frac{x_l+x_m}{2}$

6. Calculate f(x_l) f(x_m)

   a. if f(x_l) f(x_m) < 0 then $x_l = x_l$ and $x_u = x_m$

   b. if f(x_l) f(x_m) > 0 then $x_l = x_m$ and$x_u = x_u$

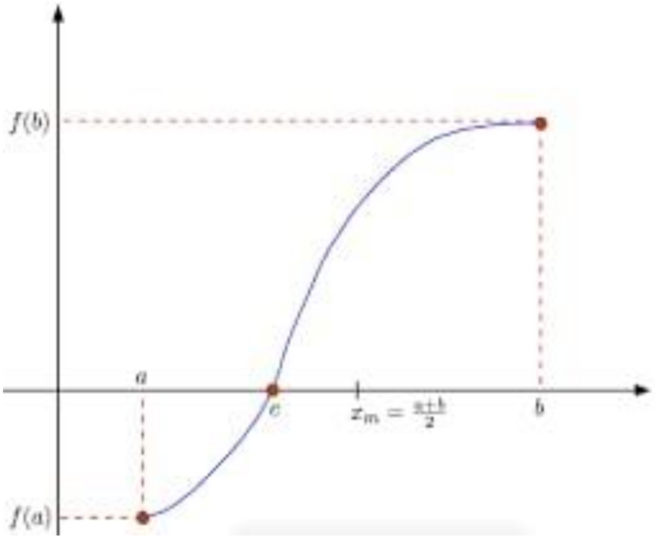
   c. if f(x_l) f(x_m) = 0 then go to step 8.

7. If | f(x_m) > e then go to step (5) otherwise go to step (8)

8.Display x_m as root .

9. Stop.

1.1.2 Example



$F(x) = 2x^2-4$

First iteration

X	0	1	2
F(x)	-4	-2	28

$x_{l=}$ 1, x_u =2

f (x_l)=-2<0 and f(x_u)= 28> 0

f(x_l)f(x_u)<0

Now roots are: $x_m = \frac{x_l - x_u}{2}$

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

f(x_m)=2.1.5^4-4 = 6.124 >0

Now x_l =-2 x_m= 1.5 for 2nd iteration

n	x_l	**F(x_l)**	x_u	**f (x_u)**	**x_m**	**f(x_m)**	**Update value**
1	1	-2	2	28	1.5	6.125	x_u=x_m
2	1	-2	1.5	6.125	1.25	0.8828	x_u=x_m
3	1	-2	1.25	0.8828	1.125	-0.7964	x_l=x_m
4	1.125	-0.7964	1.25	0.8828	1.1875	-0.0229	x_l=x_m
5	1.1875	-0.0229	1.25	0.8828	1.2188	0.4125	x_u=x_m
6	1.1875	-0.0229	1.2188	0.4125	1.2031	0.1906	x_u=x_m

Error

$\in=|\frac{x^{new}-x^{old}}{x^{new}}|$

Absolute error

$\in=|\frac{x^{new}-x^{old}}{x^{new}}|* 100$

Code:

```
# guess1 = 1, guess2 = 2
from math import sin
def bisection(x0, x1,e):
step = 1
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('root is :%0.8f '%x2)
#    Return x2

def f(x):
return 2*x**2-4

x0 = float (input('first guess: '))
x1 = float (input('second guess: '))
e = float (input('tolerance: '))

if f(x0) * f(x1) > 0.0:
print ('given guess values do not bracket the root') else:
root = bisection (x0, x1,e)
```

output:

```
first guess: 1
second guess: 2
tolerance: 0.001
iteration 1, x2 = 1.500000 and f(x2)= 0.500000
iteration 2, x2 = 1.250000 and f(x2)= -0.875000
iteration 3, x2 = 1.375000 and f(x2)= -0.218750
iteration 4, x2 = 1.437500 and f(x2)= 0.132812
iteration 5, x2 = 1.406250 and f(x2)= -0.044922
```

iteration 6, x2 = 1.421875 and f(x2)= 0.043457
iteration 7, x2 = 1.414062 and f(x2)= -0.000854
root is :1.41406250

Drawbacks:

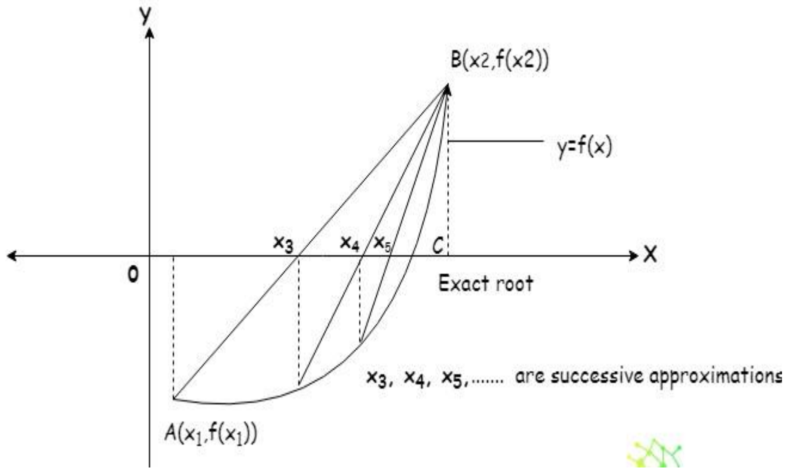
i. Rate of convergence is slow
ii. Relies on sign change
iii. Cannot detect multiple roots
iv. We need a lot of iteration for convergence

1.2 False Position Method

The procedure of false position method and bisection method is similar. The only difference is the formula used to calculate the new estimate of the root x_2.

$$x_2 = \frac{x_1 f(x_0) - x_0 f(x_1)}{f(x_0) - f(x_1)}$$

In this method we again choose two initial guesses x_0 and x_1 such that $f(x_0)\,f(x_1) < 0$



1.2.1 Algorithm

1. 1. start
2. Define function f(x)
3. Choose initial guesses x_0 and $x_1$such that f(x_0)f(x_1)< 0
4. Choose pre specified tolerable error e.
5. calculate new approximated root as $x_2 = \frac{x_1 f(x_0) - x_0 f(x_1)}{f(x_0) - f(x_1)}$
6. Calculate f(x_0) f(x_2)
   a. if f(x_0) f(x_2) < 0 then $x_0 = x_0$ and $x_1 = x_2$
   b. if f(x_0) f(x_2) > 0 then $x_0 = x_2$ and$x_1 = x_1$
   c. if f(x_0) f(x_2) = 0 then go to step 8.

7. If | f (x_2) > e, then go to step (5) otherwise go to step (8).
8. Display x_2 as root
9. Stop.

1.2.2 Example

$2x^2 - 2x - 6$ =0

$2x^2$ -2x -6 =0

Let f(x) =$2x^2 - 2x - 6$ =0

Here

X	0	-1	-2	-3
F(x)	-6	-2	6	18

First iteration

x_0=-2 ; x_1= -1

f(x_0)=-2 <0 ; f(x_1) = 6 > -0 f(x_0)f(x_1) < 0

Roots lie between x_0 = -2 and x_1 =-1

Formula:

as $x_2 = \frac{x_1 f(x_0) - x_0 f(x_1)}{f(x_0) - f(x_1)}$

x_2=-2-6.$\frac{-1-(-2)}{-2-6}$

x_2 = -1.25

f(x_2) = f(-1.25) = 2. (-1.25.-1.25)-2(-1.25) -6 = -0.375 <0

x_2= -2 x_1 =-1.25

Now f(x_0)= 6 and f(x_1)= -0.375

N	$\boldsymbol{x_0}$	**F($\boldsymbol{x_0}$)**	$\boldsymbol{x_1}$	**F($\boldsymbol{x_1}$)**	$\boldsymbol{x_2}$	**F($\boldsymbol{x_2}$)**	**Update value**
1	-2	6	-1	-2	-1.25	-0.375	$x_1 = x_2$
2	-2	6	-1.25	-0.375	-1.2491	-0.0623	$x_1 = x_2$
3	-2	6	-1.2941	-0.0623	-1.3014	-0.0101	$x_1 = x_2$
4	-2	6	-1.3014	-0.0101	-1.3025	-0.0016	$x_1 = x_2$
5	-2	6	-1.3025	-0.0016	-1.3027	-0.0003	$x_1 = x_2$

After 5 iteration the approximate root of equation is -1.3027

Code:

```
import math import sin

def reg_falsi(f, x1,x2,tol=1.0e-6,maxfpos=100):
    if f(x1) * f(x2)<0:
   for fpos in range(1, maxfpos+1):
     Xu = x2 - (x2-x1)/(f(x2)-f(x1)) * f(x2)
       if abs(f(xh)) < tol:
           break
         elif f(x1) * f(xh) < 0:
             x2 = xh
           else:
              x1 = xh
    else:
     print ('No roots exists within the given interval')

    return xh, fpos

y = lambda x: 2*x**2 -2*x -6

x1 = float (input('enter x1: '))x2 = float (input('enter x2: '))r, n = reg_falsi(y,x1,x2)print
('The root = %f at %d false position'%(r,n
enter x1: -2
enter x2: -1
```

Output:
the root = -1.302776 at 9 false positions

Drawbacks:
1. It has linear rate of convergence.
2. It fails to determine complex root.
3. It cannot apply over an interval where the function takes values of the same sign.

1.3 Newton Raphson method

The newton Raphson is used for solving equation of form f(x) = 0 we make an initial guess for the root. To implement automatically we need formula for each iteration in term previous one. we need x_{n+1} in term of x_n. The equation of tangent line to the graph y = f(x) at a point

$(x_0$,f (x_0) is
y-f$(x_0) = f'(x_0)(x - x_0)$
The tangent line intersects the x- axis when y=0 and x=x_1 , so -f(x_0) = f'$(x_0)(x - x_0)$

Solving this for x_1 gives
$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$
Generally,
$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$

1.3.1 Algorithm
1 Start.
2. Define function as f(x) .
3. Define first derivative of f(x) as g(x).
4. Input initial guess (x_0), tolerable error (e) and maximum iteration (N) .
5. Initialize iteration counter i = 1
6. If g (x_0) = 0 then print "Mathematical Error" and go to (12) otherwise go to (7)
7. Calculate $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$

8. Increment iteration counter i = i + 1
9. If i >= N, then print "Not Convergent" and go to (12) otherwise go to (10)
10. If |f (x_1)| > e, then set $x_0 = x_1$ and go to (6) otherwise go to (11)
11. Print root as x_1
12. Stop.

1.3.2 Example

$2x^3 - 2x - 5$
Let f(x) = $2x^3 - 2x - 5$

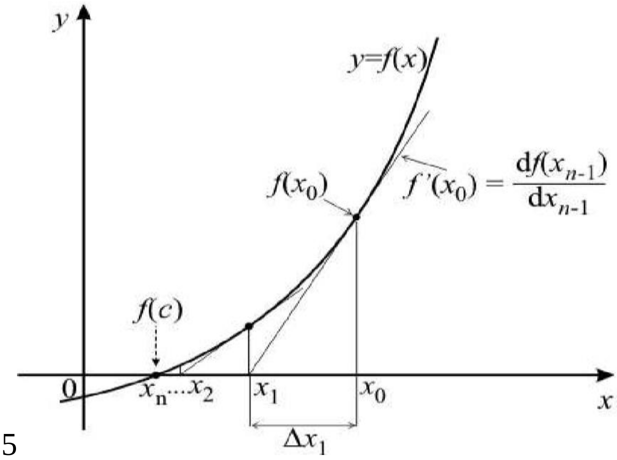
f'(x) = $6x^2 - 2$

x_s =1 x_g =2
f(1)= -5 <0 f(2) = 7 >0

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

First iteration

f(x_0) =2.(1.5^3) -2.(1.5) -5 =-1.25



$f'(x) = 6(1.5^2) - 2 = 11.5$

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

$x_1 = \frac{-1.25}{11.5}$

$x_1 = 1.6087$

Table for next iteration:

n	x_0	**F(x_0)**	**F'(x_0)**	x_1	**Update value**
1	1.5	-1.25	11.5	1.6087	$x_0 = x_1$
2	1.6087	0.1089	13.5274	1.6006	$x_0 = x_1$
3	1.6006	0.0006	13.3724	1.6006	$x_0 = x_1$
4	1.6006	0	13.3715	1.6006	$x_0 = x_1$

Code:

```
def newton(fn,dfn,x,tol,maxiter):

    for i in range(maxiter):
        xnew = x - fn(x)/dfn(x)
        if abs(xnew-x)<tol:
            break
        x = xnew
    return xnew, i

y = lambda x: 2*x**3-2*x-5
dy = lambda x : 6*x**2-2

x, n = newton(y, dy, 5, 0.0001, 100)
print('the root is %.3f at %d iterations.'%(x,n))
```

Output

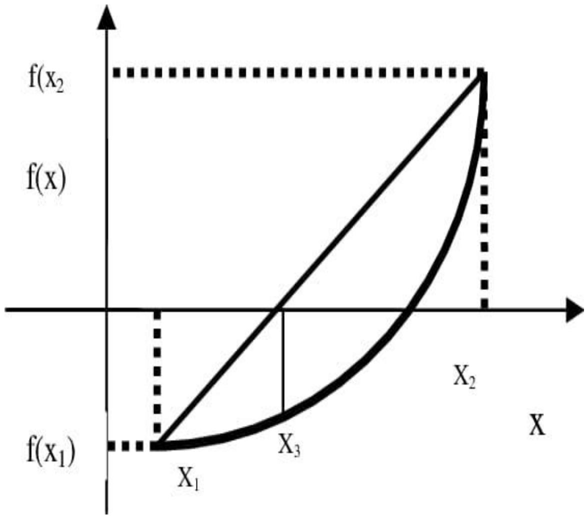
The root is 1.601 at 6 iterations.

1.4 Secant Method

Secant method is also a iterative technique for finding the root for the polynomials by consecutive approximation. It's like the Regular-falsie method but here we don't need to check f(x_0)f(x_1)<0 again and again after every approximation. In this method, the neighbourhoods' roots are approximated by secant line or chord to the function **f(x)**. It's also advantageous of this method that we don't need to differentiate the given function **f(x)**, as we do in **Newton-Raphson** method.

Secant method start with a two initial approximation x_0 and x_1 and then calculate the x_2 by the same formula as in regular falsie method but proceed to the next iteration. Consider employing an approximating line based on interpolation. let's we have two roots say x_0 and x_1 then we have linear function. Formula for secant method.

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



1.4.1 Algorithm

1. Start
2. Define function f(x)
3. Choose initial guess x_0 and x_1 tolerable error and maximum iteration (N)
4. Initialize iteration counter I =1
5. If f(x_0)=f(x_1) then print and go to step 11 otherwise go to step 6
6. Calculate x_2 =
7. Increment iteration counter i=i+1
8. In then print " not convergent" and go to step 11 otherwise go to step 9
9. If(x_2) is greater than e then set x_0 =x_1,x_1 =x_2 and go to step 5 otherwise go to step 10
10. Print root as x_2
11. Stop

1.4.2 Example

$x^3 + x^2 - 3x - 3$
f(x) =$x^3 + x^2 - 3x - 3$

Initial guess

x_0 = 1; x_1= 2
f (x_0) = 1^3 +1^2 -3(1) -3 = -4
f () = 2^3 + 2^2 -3(2)-3 = 3

First iteration

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

2-$\frac{3(2-1)}{(3-(-4)}$= 1.57142

f(x) = $(1.5714)^3$ +(1.5714) -3(1.5741) -3 = -1.3644

approximate value after 4 iterations

Table

N	x_0	x_1	f(x_0)	F(x_1)	x_2	F(x_2)
1	1	2	-4	3	1.57143	-1.3644
2	2	1.70541	3	-1.36443	1.70541	-0.24775
3	1.57143	1.73514	-1.364431	-0.247745	1.73514	0.02926
4	1.70541	1.732	-0.247745	0.02925554	1.732	-0.00052

Absolute error $|\frac{x^{new}-x^{old}}{x^{new}}|$ *** 100**

Code:

```
from math import sin
def secant(fn,x1,x2,tol,maxiter):
    for i in range(maxiter):
        xnew  = x2 - (x2-x1)/(fn(x2)-fn(x1))*fn(x2)
        if abs(xnew-x2) < tol:
            break
        else:
            x1 = x2
            x2 = xnew
    else:
        print('warning: Maximum number of iterations is reached')
    return xnew, i

f = lambda x: x**3 -x**2- 3*x - 3
```

```
x1 = float(input('enter x1: '))
x2 = float(input('enter x2: '))

r, n = secant(f,x1,x2,1.0e-6,100)

print('Root = %f at %d iterations'%(r,n))
enter x1: 1
enter x2: 2
```

Output:
Root = 2.598675 at 8 iterations

Drawbacks:
The convergence in secant method is not always certain.
At any stage of iteration this method fails. Newton approaches is more easily generalized to new ways for solving non-linear same equation.

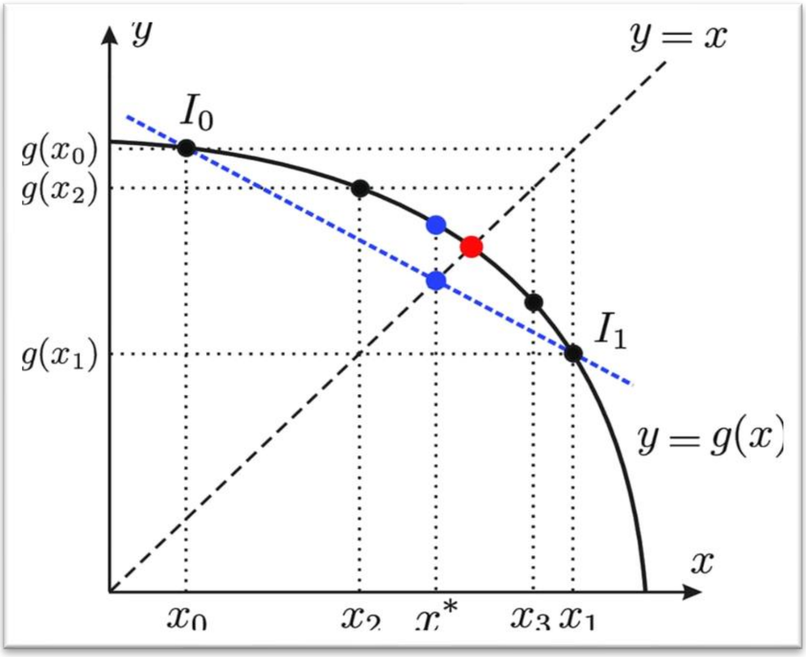
1.5 Fixed Point Iteration Method

Fixed point iteration method is simple method for finding real root of nonlinear equation. it required only one initial guess to stop. this method is also known as iterative method.
A point says b is called fix position if it satisfies the equation x=g(x). The equation f(x) =0 can be converted algebraically into the x= g(x) and then using the iterative scheme with the recursive relation

$$\boldsymbol{x_{i+1} = g(x_i)} \qquad\qquad \mathbf{i= 0,1,2......}$$

With some initial guess x_0 is called fixed point iteration method. choose g(x) such that |g'(x) |<1

If we have lower value of g'(x) then less iteration is required. The rate of convergence is more if g(x) is less.



1.5.1 Algorithm

1: start

2: Define function f(x)

3:Define function g(x) which is obtained from f(x) =0 such that x= g(x) and |g'(x)<1
4: Choose initial guess x_0 tolerable error e and maximum iteration N
5: Set iteration counter : step =1
6: Calculate x_1 = g(x_0)
7: Increment iteration counter
8: if step >N then print "not convergent" and go to step(12)otherwise go to step (10)
9: set x_0 = x_1 for next iteration
10: if |f(x)|> e then go to step (6) otherwise go to step (11).
11: Display x_1 as root .
12: stop.

1.5.2 Example

$x^3 - 4x^2 - x = 10$

f(x) = $x^3 - 4x^2 - x = 10$

x	0	1	2	4	5
f(x)	-10	-12	-16	-6	20

So, interval in which root lies is (4,5)

Formula

$x_{i+1} = g(x_i)$

Now by rearranging

x = -$x^3 + 4x^2 + 10$

g(x)= 10 +4$x^2 - x^3$

|g'(x)| = | 0+ 8x -3x^2|> 1 for all x belong to (4,5)

Now g(x) =

Clearly, |g'(x)|<1

Choosing x_0 =4

x_1= $\frac{4(16)-4+10}{4}$ = 4.375

N	x_i	x_{i+1} = g (x_i)
1	4	4.3754
2	4.375	4.293
3	4.293	4.309
4	4.309	4.306
5	4.306	4.307
6	4.307	4.307

Error = $|\frac{x_{i+1}-x_i}{x_{i+1}}|$

Drawbacks

1. t requires a starting interval containing a change of sign. Therefore, it cannot find repeated roots.

2. It has a fixed rate of convergence, which can be much slower than other methods, requiring more iterations to find the root to a given degree of precision.

Chapter 2

METHOD TO SOLVE LINEAR EQUATION

2.1 Jacobi Method

The first iterative technique is called the Jacobi method, named after Carl Gustav Jacob Jacobi (1804-1851) to solve the system of linear equations.

In numerical analysis, the Jacobi method is an iterative algorithm for determining the solution of a strictly diagonally dominant system of linear equations. In this method, an approximate value is filled in for each diagonal element. Until it converges, the process is repeated.

The given system of equation has unique solution.

$$a_{11}x_1 + a_{12}x_2 + \cdots . a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \cdots . a_{2n}x_n = b_2$$
$$a_{31}x_1 + a_{32}x_3 + \cdots . a_{3n}x_n = b_3$$

The coefficient matrix A has no zeros on its main diagonal namely. = $a_{11}, a_{22} \ldots . a_{nn}$ if any diagonal entries are zero, then we interchange the rows and column to make entries non-zeros.

Jacobi method can be defined as:

$$\boldsymbol{x^{k+1} = D^{-1}(L + U)x^k + D^{-1}\, b}$$

where the matrices **D, L, U** represent the diagonal, strictly lower triangular, and strictly upper triangle parts of A, respectively.

Convergence

1. Matrix should be strictly dominant.

2. Spectral Radius: The Spectral radius, p(A), provides a valuable measure of the eigenvalues, which helps determine if a numerical scheme will converge.

The spectral radius, p(A), of a matrix A is defined by

p(A)= max |λ|

where λ is an eigenvalue of A.

If p(A)< 1 then our system is converged.
Eigen values defined as
det (A- λi)

2.1.1 Algorithm

1. start
2. Arrange given system of linear equation in diagonally dominate form.
3. Read tolerable error. (e)
4. covert 1^{st} 2^{nd} and 3^{rd} equation in term of 1^{st} 2^{nd} and 3^{rd} variable and so one.

5.set initial guesses for x_0, y_0, z_0

6. substitute the value of , x_0 , y_0 $z_0$from 5 in equation obtained in step 4 to calculate new values of| x_1, y_1 , z_1| and so on.
7.if | $x_0 - x_1| >$ e and $|y_0$- y_1 | > e and | z_0-z_1| > e and so on then go to step 9
8.set x_0 =x_1 ,y_0 =y_1 ,$z_0 = z_1$
9. print value of x_1 , y_1, z_1.
10.stop

2.1.2 Example

20x +y -2z =17
3x + 20y – z = -18
2x – 3y + 20z = 25
We rewrite the equation in the form of
x = $\frac{1}{20}$ (17 -y + 2z)
y = $\frac{1}{20}$ (-18-3x-z)
z= $\frac{1}{20}$ (25 – 2x+3y)

For first iteration (x_0 , y_0 , z_0) = 0
x^1 = $\frac{1}{20}$(17-y_0 + 2z_0) =$\frac{17}{20}$=0.85
y^1 = $\frac{1}{20}$ $(-18 – 3x_0 – z_0)$ =$\frac{-18}{20}$ = -0.9
z^1 = $\frac{1}{20}$(25-2x_0+3y_0) = $\frac{25}{20}$ = 1.25
Now x^1 = 0.85 , y^1 = -0.9 , z^1 = 1.25

Table for next four iteration:

	K=0	K=1	K=2	K=3	K=4	K=5
$x^{(k+1)}$	0.85	1.02	1.0134	1.0009	1.000	1.0000
$y^{(k+1)}$	-0.9	-0.965	-0.9954	-1.0018	-1.0002	-1.0000
$z^{(k+1)}$	1.25	1.1575	1.0032	0.9993	0.9996	1.0000

As values in last two iteration are same so we stop.

Hence the solution of is (1, -1 ,1)

Code:

```
 # in diagonally dominant form
f1 = lambda x,y,z: (17-y+2*z)/20
f2 = lambda x,y,z: (-18-3*x+z)/20
f3 = lambda x,y,z: (25-2*x+3*y)/20

# Initial setup
x0 = 0
y0 = 0
z0 = 0
```

```python
count = 1

# Reading tolerable error
e = float(input('Enter tolerable error: '))

# Implementation of Jacobi Iteration
print('\nCount\tx\ty\tz\n')

condition = True

while condition:
    x1 = f1(x0,y0,z0)
    y1 = f2(x0,y0,z0)
    z1 = f3(x0,y0,z0)
    print('%d\t%0.4f\t%0.4f\t%0.4f\n' %(count, x1,y1,z1))
    e1 = abs(x0-x1);
    e2 = abs(y0-y1);
    e3 = abs(z0-z1);

    count += 1
    x0 = x1
    y0 = y1
    z0 = z1
```

Entre tolerable error: 3

Output:

Count	x	y	z
1	0.8500	-0.9000	1.2500
2	1.0200	-0.9650	1.0300
3	1.0012	-1.0015	1.0032
4	1.0004	-1.0000	0.9996
5	1.0000	-1.0001	1.0000

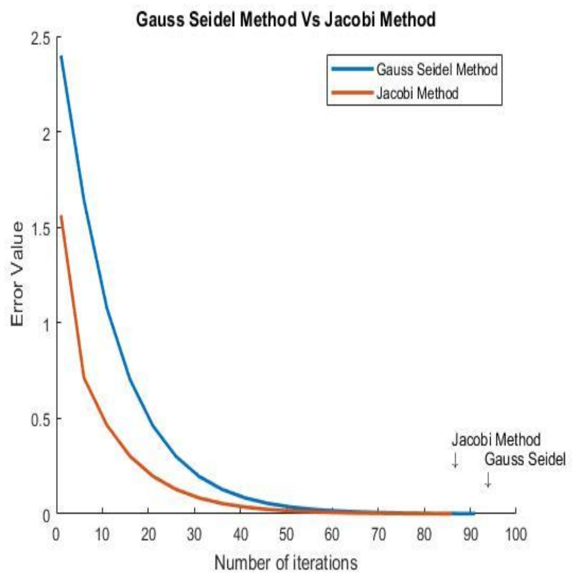
Disadvantage of Jacobi method

1. The Jacobi iterative method works fine with well-conditioned linear systems. If the linear s ystem is ill-conditioned, it is most probably that the Jacobi method will fail to converge.

2 The Jacobi method can generally be used for solving linear systems in which the coefficient matrix is diagonally dominant.

2.2 Guess Seidel Method

This method is one step further of Jacobi method. Where the better solution is x =($x_{1,}$ $\boldsymbol{x_2 \ldots x_n}$) **if**$x_{1,}$(k+1) is a better approximation to the value of $x_{1,}$ then $x_{1,}$(k) then it would be better that we have found the new value $x_{1,}$ (k+1)to use it rather than to use the old value of x_1(k) in finding $\boldsymbol{x_2}$(k+1),…x_n (k+1). So $x_{1,}(k+1)_{,}$ is found as in Jacobi method, but in finding $\boldsymbol{x_2(k+1)}$ **instead** of using the old value of x_1 (k) and old value of …. $x_{3,}$ (k)….. $x_{n,}$ (k) and similarly, for finding $x_{3\,,}(k+1)$ …. $x_{n,}$ $(k+1)$ This process to find the solution of given linear equation is known to be guess seidel method. the guess seidel method is iterative technique for solving a system of n(n=3) linear equation with unknown x. Although the three-resulting value for both guess seidel and guess Jacobi method is same in the first step, you should be able to find the difference between these two methods. in Jacobi method no updates are applied until next step .in guess seidel method new $x_{3\,,}$ is calculated from new $x_{2\,,}$ in 2nd equation in guess seidel method spectral radius can be found by $(\mathrm{D}-\mathrm{L})^{-1}U$



2.2.1 Algorithm

1 start.

2 Arrange given system of equation in diagonally dominate form.

3 Read tolerable error (e).

4 Convert first equation in term of first variable and 2nd equation in term of second variable and so on..

5 set initial guesses for x_0 ,y_0 $and\ z_0$ and so on.

6 substitute value of y_0 $and\ z_0$….. from step 5 in first equation obtained from step 4 to calculate new value of$x_{1,}$ $use\ x_{1,}$ $and\ z_0$,u_0 in second equation obtained from step 4 to calculate new value of y_1 . similarly use x_1 , y_1 , u_0 to find z_1 and

so on.

7 if$|x_{0\,,} - x_{1,}|$>e and $|y_{0\,,} - y_{1,}|$> e $|z_{0\,,} - z_{1,}|$> e and so on then go to step 9 .

8 set $x_{0\,,} = x_{1,}, y_{0\,,} = y_{1,}$ and $z_{0\,,} = z_{1,}$ and go to step 6

9 print value of $x_{1,}$ $y_{1,}z_{1,}$

10 stop

2.2.2 Example

4x + y + 2z=4
3x +5y + z =7
x +y +3z =3
Solving equation by using gauss seidel method
Solution:

x = $\frac{1}{4}$ (4 -y -2z)

y = $\frac{1}{5}$(7 - 3x -z)

z=$\frac{1}{3}$ (3-x-y)

Formula

x^{k+i} =$\frac{1}{4}$ (4 -y^k -2z^k)

y^{k+i} = $\frac{1}{5}$(7 - 3x^{k+i} –z^k)

z^{k+i} =$\frac{1}{3}$ (3-x^{k+i} -y^{k+i})

i=1 2 3 …..
For k =0
For first iteration (x, y, z) = (0 ,0, 0)

x^1= $\frac{1}{4}$ (4 -0-0) = 1

y^1= $\frac{1}{5}$(7 - 3 -0) = 0.8

z^1= $\frac{1}{3}$ (3-1-0.8) = 0.4

x^1 = 1 , y^1 = 0.8 , z^1 =0.4
By using calculator next iteration table is below:

N	**K=0**	**K=1**	**K=2**	**K=3**	**K=4**
x^{k+i}	1	0.6	0.5	0.508	0.5004
y^{k+i}	0.8	0.96	O.992	0.9984	0.99984
z^{k+i}	0.4	0.48	0.496	0.4992	0.49984

Code:

```
# # Gauss Seidel Iteration
# Defining equations to be solved
# in diagonally dominant form
f1 = lambda x,y,z: (4-y-2*z)/4
f2 = lambda x,y,z: (7-3*x-z)/5
f3 = lambda x,y,z: (3-x-y)/3

# Initial setup
```

```python
x0 = 0
y0 = 0
z0 = 0
count = 1

# Reading tolerable error
e = float(input('Enter tolerable error: '))

# Implementation of Gauss Seidel Iteration
print('\nCount\tx\ty\tz\n')

condition = True

while condition:
    x1 = f1(x0,y0,z0)
    y1 = f2(x1,y0,z0)
    z1 = f3(x1,y1,z0)
    print('%d\t%0.4f\t%0.4f\t%0.4f\n' %(count, x1,y1,z1))
    e1 = abs(x0-x1);
    e2 = abs(y0-y1);
    e3 = abs(z0-z1);

    count += 1
    x0 = x1
    y0 = y1
    z0 = z1

    condition = e1>e and e2>e and e3>e

print('\nSolution: x=%0.3f, y=%0.3f and z = %0.3f\n'% (x1,y1,z1))
```

Enter tolerable error: 0.001

Output:

Count	x	y	z
1	1.0000	0.8000	0.4000
2	0.6000	0.9600	0.4800
3	0.5200	0.9920	0.4960
4	0.5040	0.9984	0.4992
5	0.5008	0.9997	0.4998

Solution: x=0.501, y=1.000 and z = 0.
500

Advantage of guess seidel method.

1. Advantages: Calculations are simple and so the programming task is lessees.
2 The memory requirement is less.
3 Useful for small systems.

Disadvantage of guess seidel method

1. Disadvantages: Requires large no. of iterations to reach converge.
2. Not suitable for large systems.
3. Convergence time increases with size of the system.

2.3 SOR method:

Method involving $\boldsymbol{x_i^{(k)}} = \boldsymbol{x_i^{(k-1)}}$+ ω $\frac{r_{ii}^k}{a_{ii}}$ is relaxation technique. and ω is between 0 and 1. Is called under relaxation method.

Greater than 1 is called over relaxation method.

It is used to speed up the convergence for the system which is already converge by gauss seidel method. Abbreviation of this method is successive over relaxation method. And used to solve the system of linear equation. formula for sor method is

$\boldsymbol{x_i^{(k)}}$ =**(1-** ω) $\boldsymbol{x_i^{(k-1)}}$+ $\frac{\omega}{a_{ii}}$[$b_{i-}\sum_{j=1}^{i-1} a_{ij}$ $\boldsymbol{x_j^{(k)}}$ - $\sum_{j=i+1}^{n} a_{ij}$ $\boldsymbol{x_j^{(k-1)}}$**]**

ω can be found by formula

ω $\frac{2}{\sqrt{1-\rho(T_j)2}}\rho$

p(T_j)2 is spectral Radius in case of sor method. Can be find by

$$D^{-1}(L+U)$$

Special case:

1. if 0< $\rho(T_g)$< $\rho(T_j)$<1

2.1< $\rho(T_j)$< $\rho(T_g)$

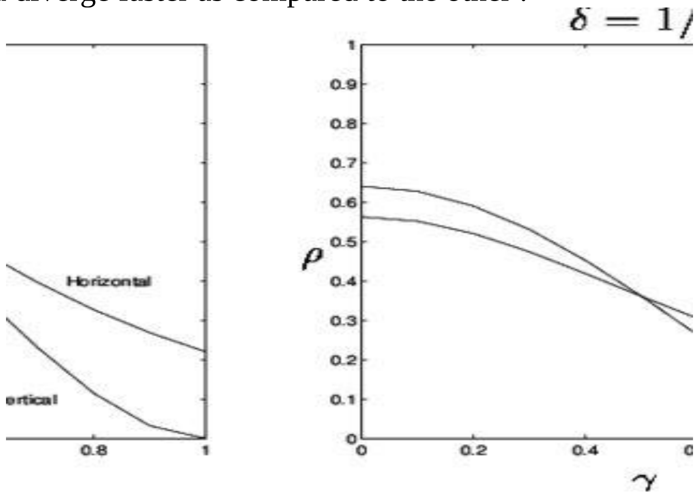
3. $\rho(T_g)$= $\rho(T_j)$=0

4. $\rho(T_j) = \rho(T_g)$=*0*

from 1 we conclude that when one method converge then both coverage and guess seidel method converge faster as compared to the other. From 2nd part we can conclude that when one method diverge other also diverge

and

guess seidel method diverge faster as compared to the other .



4.3.1 Example

$3x_1 - x_2 + x_3 = 1$

$3x_1 + 6x_2 + 3x_3 = 0$

$3x_1 + 3x_2 + 7x_3 = 4$

Solution:

Formula:

$x^{k+i} = (1\text{-}w)x_i^k + \frac{w}{a_{ii}}(b_i - \sum_{j=1}^{i=1} aij\ xj - \sum_{j=i+1}^{n} aijxj$

i=1   2 3 …..

$x_1 = \frac{1}{3}(1+x_2 - x_3)$

$x_2 = \frac{1}{6}(0 - 3x_2 - 2x_3)$

$x_3 = \frac{1}{7}(4 - 3x_2 - 3x_1)$

putting value of w as w =1.1

$x_1 = (1.1-1)x_1(1.1)\ \frac{1}{3}(1+x_2 - x_3)$

$x_2 = (1.1-1)x_2(1.1)\frac{1}{6}(0 - 3x_2 - 2x_3)$

$x_3 = (1.1-1)x_3(1.1)\ \frac{1}{7}(4 - 3x_2 - 3x_1)$

First iteration:

$(x_1^0, x_2^0, x_3^0) = (0,0,0)$

K=0

$x_1^{(1)} = \frac{1.1}{3}(1+0-0) - 0.1(0)$

=0.3666

$x_2^{(1)} = \frac{1.1}{6}(0-3(0.366)-2(0)) - 0.1(0) = -0.21$

$x_3^{(1)} = \frac{1.1}{7}(4-3(0.366) - 3(-0.21) - 0.1(0) = 0.5507$

Table for next iterations

n	**k=0**	**k=1**	**k=2**	**k=3**	**k=4**
$x_1^{(k+i)}$	0.3666	0.0541			
$x_2^{(k+i)}$	-0.21	-0.2115			
$x_3^{(k+i)}$	0.5507	0.6477			

Code:

```
# Successive over-relaxation (SOR)
# Defining equations to be solved
# in diagonally dominant form
f1 = lambda x,y,z: (1+y-z)/3
```

```
f2 = lambda x,y,z: (-3*y -2*z)/6
f3 = lambda x,y,z: (4-3*y-3*x)/7

# Initial setup
x0 = 0
y0 = 0
z0 = 0
count = 1

# Reading tolerable error
e = float(input('Enter tolerable error: '))

# Reading relaxation factor
w = float(input("Enter relaxation factor: "))

# Implementation of successive over-relaxation
print('\nCount\tx\ty\tz\n')

condition = True

while condition:
    x1 = (1-w) * x0 + w * f1(x0,y0,z0)
    y1 = (1-w) * y0 + w * f2(x1,y0,z0)
    z1 = (1-w) * z0 + w * f3(x1,y1,z0)
    print('%d\t%0.4f\t%0.4f\t%0.4f\n' %(count, x1,y1,z1))
    e1 = abs(x0-x1);
    e2 = abs(y0-y1);
    e3 = abs(z0-z1);

    count += 1
    x0 = x1
    y0 = y1
    z0 = z1

    condition = e1>e and e2>e and e3>e

print('\nSolution: x = %0.3f, y = %0.3f and z = %0.3f\n'% (x1,y1,z1))
```

Enter tolerable error: 0.002
Enter relaxation factor: 1.1

Output:

Count	x	y	z
1	0.3667	0.0000	0.4557

Solution: x = 0.367, y = 0.000 and z = 0.456

Advantage of guess SOR

1. They are often easy to use.

2 They can produce results quickly.

3 They can solve equations where an analytic solution is impossible.

Disadvantage of SOR

1. A disadvantage is all of the usual convergence criteria may increase or may decrease from round to round, whereas with other iterative methods for equations guaranteed to converge, convergence criteria generally decrease from round to round.
2. They are not as elegant as analytic solutions.
3. They do not provide any insight into generalizations. An exact value may not be clear.

Chapter 3

INTERPOLATION

3.1 Newton Forward Difference Interpolation

The method which is used to estimate the values of function for any intermediate values of independent variables. The process of finding the value of function outside the given output is called extrapolation.

forward difference:

The difference $y_1 - y_0$, y_2 -y_1 , y_3 -, y_2 ……., y_n-, y_{n-1} is denoted by dy_1 dy_0 , dy_2 dy_1 ,d y_3 -, ….,d y_n are called first forward difference . $\Delta y_r = y_{r+1} - y_r$

Formula for newton forward interpolation:

f(a+hu)= f(a) +u$\Delta f(a) + \frac{u(u-1)\Delta^2}{2!}$f(a) +…….$\frac{u(u-1)(u-2)(u-n+1)\Delta^3}{n!}$f(a) this formula is useful for interpolating the value of f(x) near the beginning of the set of values. h is difference between the interval's and a is first term in table as shown in example and u=x-a\h

3.1.1 Example

x	1	2	3	4	5
f(x)=y	1	-1	1	-1	1

Table:

x	y	Δy	$\Delta^2 y$	$\Delta^3 y$	$\Delta^4 y$
1	y_0=1				
2	y_1 = -1	$y_1 - y_0$= -2 =Δy_0	$\Delta^2 y_0$ = Δ y_1 -Δy_0= 4		
3	y_2= 1	$y_2 - y_1$= 2 = Δy_1	$\Delta^2 y_1$ = Δ $y_2 - y_1$= -4	$\Delta^3 y_0$ = $\Delta^2 y_1$-$\Delta^2 y_0$= -8	$\Delta^4 y_0$ = $\Delta^3 y_1 - \Delta^3 y_0$= 16
4	y_3=-1	$y_3 - y_2$ = -2 =Δy_2		$\Delta^3 y_1$= $\Delta^2 y_2 - \Delta^2$ y_1 = 8	
5	y_4= 1	$y_4 - y_3$ =2 = Δy_3	$\Delta^2 y_2$ = Δ y_3- Δ y_2= 4		

F(x) = y_0 +$\frac{P}{1!}$ Δy_0 + $\frac{P(P-1)}{2!}$ $\Delta^2 y_0$ +$\frac{P(P-1)(P-2)}{3!}$ $\Delta^3 y_0$ + $\frac{P(P-1)(P-2)(P-3)}{4!}$ $\Delta^4 y_0$

P = $\frac{x-x0}{h}$ p= x-1

h=1

1 + $\frac{x-1}{1}$(-2) + $\frac{(x-1)(x-2)}{2}$(4) + $\frac{(x-1)(x-2)(x-3)}{6}$(-8) + $\frac{(x-1)(x-2)(x-3)(x-4)}{24}$(16)

$= \ 1-2x+2+2(x^2-3x+2)+\frac{4}{3}(x^3-6x^2+11x-6)+\frac{2}{3}(x^4-10x^3+35x^2-50x+24$

$\frac{2}{3}x^4-8x^3+\frac{100}{3}x^2-56x+31$

This is required polynomial.
Code:

```
import numpy as np
import matplotlib.pyplot as plt
f= lambda x: 0.666*x**4 -8*x**3 + 33.3*x**2 -56*x +31
x = 0.1
h = 0.01
df1 = 0.09405
df2 = -0.118
print("\t f'(x)\t\t err\timport numpy as np
import matplotlib.pyplot as plt\t f"(x)\t\t err")

            f'(x)                  err                     f"(x)                  err
dff1 = (f(x+h)-f(x))/h
dff2 = (f(x+2*h)-2*f(x+h)+f(x))/h**2
print("FFD\t% f\t% f\t% f\t% f"%(dff1, dff1-df1, dff2 ,dff2-df2))
```

Output:

```
FFD        0.093441              -0.000609             -0.129327             -0.011327
```

3.2 Newton Backward Interpolation

This technique or formula is useful to find the values of f(x)at the end of the table the difference $y_1 \ - y_0$, y_2 -y_1 , y_3 -, y_2 ……., y_n-, y_{n-1} is denoted by $dy_1\, dy_0$, $dy_2\, dy_1$,d y_3 -, ….,d y_n are called first forward difference .$\nabla y_r = y_r - \ y_{r-1}$
Formula ;

f(a+nh +uh) = f(a+nh) +u∇ f(a+nh) + $\frac{u(u+1)}{2!}\nabla^2$ f$(a+nh)$…….$\frac{u(u+1)\ (u+2)u(u+n-1)}{n!}\nabla^n$ f$(a+nh)$……. Here h is difference between intervals and u=x-a_n\h here and is last term.

3.2.1 Example

x	10	11	12	13
F(x)	22	24	28	34

Table

x	y	Δy	Δ^2y	Δ^3y
x_0 =10	y_0 = 22	24-22=2		

x_1 =11 x_2 =12 x_3 =13	y_1 = 24 y_2 =28 y_3 =38	28-24=4 34-28=6	4-2=2 6-4=2	2-2=0

Formula is given by:

F(x) = $y_n + \frac{P}{1!}\nabla y_0 + \frac{P(P+1)}{2!}\nabla^2 y_n + \frac{P(P+1)(P+2)}{3!}\nabla^3 y_n \quad + \frac{P(P+1)(P+2)(P+3)}{4!}\nabla^2 y_n \ldots\ldots$

H= $x_0 - x_1$ = 1

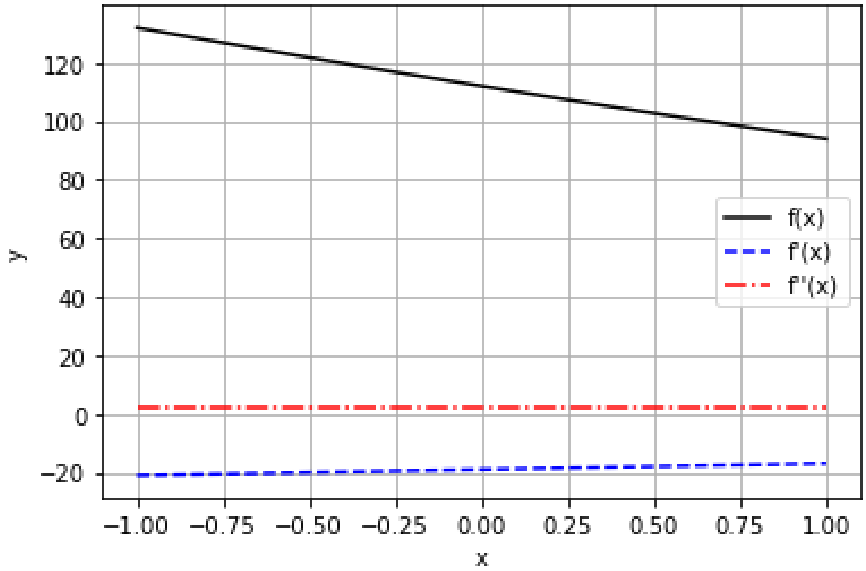
P = $\frac{x-x_n}{h}$ = x-13

F(x) = 34 + (x-13)$\frac{6}{1}$ + $\frac{x-13(x-12)}{2!}$ (2) + 0

Code:

```
import numpy as np
import matplotlib.pyplot as plt
= lambda x: x**2 - 19*x +112
x = 0.1
h = 0.01
df1 = 0.09405
df2 = -0.118
print("\t f'(x)\t\t err\t\t f"(x)\t\t err")

            f'(x)                err                  f"(x)                err
dff1 = (f(x+h)-f(x))/h
dff2 = (f(x+2*h)-2*f(x+h)+f(x))/h**2
print("FFD\t% f\t% f\t% f\t% f"%(dff1, dff1-df1, dff2 ,dff2-df2))
FFD       -18.790000          -18.884050          2.000000          2.118000

dff1 = (f(x)-f(x-h))/h
dff2 = (f(x)-2*f(x-h)+f(x-2*h))/h**2
print("BFD\t% f\t% f\t% f\t% f"%(dff1,dff1-df1,dff2,dff2-df2))
BFD       -18.810000          -18.904050          2.000000          2.118000

dff1 = (f(x+h)-f(x-h))/(2*h)
dff2 = (f(x+h)-2*f(x)+f(x-h))/h**2
print("CED\t% f\t% f\t% f\t% f"%(dff1,dff1-df1,dff2,dff2-df2))
CED       -18.800000          -18.894050          2.000000          2.118000

import numpy as np
import matplotlib.pyplot as plt
```

```
f= lambda x: x**2 - 19*x +112
h=0.001
x= np.linspace(-1,1,60)
#backward differnce
dff1 = (f(x)-f(x-h))/h
dff2 = (f(x)-2*f(x-h)+f(x-2*h))/h**2
#plot
plt.plot(x,f(x),'-k',x,dff1,'--b',x,dff2,'-.r')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(["f(x)","f'(x)","f"(x)"])
plt.grid()
```



3.3 Newton Central Difference Interpolation

3.3.1 Example:

x	y	δy	$\delta^2 y$	$\delta^3 y$
20	512			
		-73		
30	439		-20	
		-93		10
40	346		-10	
		-103		
50	243			

Given

$x = 35\ , x_0 = 40\ , h = 10$

$$p = \frac{x - x_0}{h} = \frac{35 - 40}{10}$$

$$p = -\frac{5}{10} = -0.5$$

$$\boldsymbol{y(x) = y_0 + p\left(\frac{\delta y_0 + \delta y_{-1}}{2}\right) + \frac{p^2}{2!}\delta^2 y_{-1} \ldots\ldots\ldots}$$

$$y(35) = 346 + (-0.5)\left(\frac{-103 - 93}{2}\right) + \frac{(-0.5)^2}{2}(-10)$$

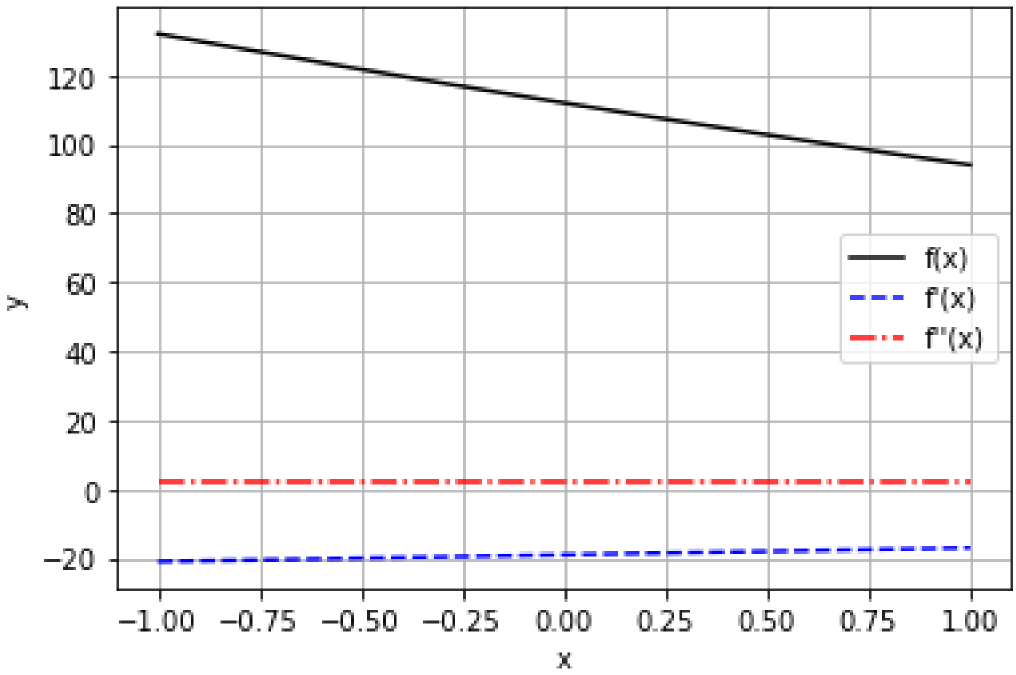
$$y(35) = 393.75$$

Code:

```
import numpy as np
import matplotlib.pyplot as plt

f= lambda x: x**2 - 19*x +112
x = 0.1
h = 0.01
df1 = 0.09405
df2 = -0.118
print("\t f'(x)\t\t err\t\t f''(x)\t\t err")

         f'(x)            err              f''(x)           err

dff1 = (f(x+h)-f(x-h))/(2*h)
dff2 = (f(x+h)-2*f(x)+f(x-h))/h**2
print("CED\t% f\t% f\t% f\t% f"%(dff1,dff1-df1,dff2,dff2-df2))
CED     -18.800000       -18.894050       2.000000         2.118000
import numpy as np
import matplotlib.pyplot as plt
f= lambda x: x**2 - 19*x +112
h=0.001
x= np.linspace(-1,1,60)
#backward differnce
dff1 = (f(x)-f(x-h))/h
dff2 = (f(x)-2*f(x-h)+f(x-2*h))/h**2
#plot
plt.plot(x,f(x),'-k',x,dff1,'--b',x,dff2,'-.r')
plt.xlabel('x')
plt.ylabel('y')
```

```
plt.legend(["f(x)","f'(x)","f"(x)"])
plt.grid()
```

Output:



Advantages and disadvantages.

1. Have a free parameter in conjunction with the fourth-difference dissipation, which is needed to approach a steady state.

2. More accurate than the first-order upwind scheme if the Peclet number is less than somewhat more dissipative

3. Leads to oscillations in the solution or divergence if the local Peclet number is larger than 2.

The general formula is very convenient to find the function value at various points if forward difference at various points are available. Similarly the polynomial approximations of functions of higher degree also can be expressed in terms of r and forward differences of higher order. Instead of using the method of solving the system as we did earlier it is convenient to use binomial formulae involving the difference operators to generate the higher order interpolation formulae.

3.4 Langrage Interpolation

Unequally spaced interpolation requires the use of the divided difference formula. It is defined as

f (x, x0) = f(x) − f(x0)

x − x0

(1)

f(x, x0, x1) = f(x, x0) − f(x0, x1)(x − x1)
f(x, x0, x1, x2) = f(x, x0, x1) − f(x0, x1, x2)(x − x2)
From equation (2), the formula can be rewritten as

(x − x1) f(x, x0, x1) + f(x0, x1) = f(x, x0) ,and the substitution of equation (1) yields,

(x − x0) (x − x1) f(x, x0, x1)+(x − x0) f(x0, x1) + f(x0) = f(x) .

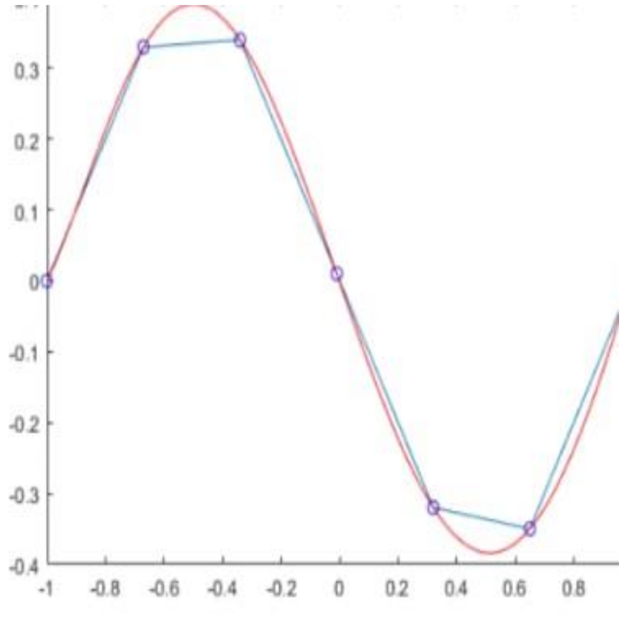
The first term is considered the remainder term as it is not in the difference table, so f(x) can be expressed approximately in terms of the divided differences as f(x) ≈ f(x0) +(x − x0) f(x0, x1)+(x − x0)(x − x1) f(x0, x1, x2) ,a second order formula. The first order formula can be written as

f(x) ≈ f(x0) +(x − x0) f(x0, x1) .

The above formulas are the most convenient for numerical computation when the divided differences are store in a matrix form. But actual explicit formulas can be written in terms of the sample function values. Lagrange First Order Interpolation Formula

Given

f(x) = f(x0) +(x − x0)



f(x0) − f(x1)

$$\mathbf{f(x)} = \frac{(x-x1)(x-x2)}{(x0-x1)(x0-x2)}\mathbf{f0} + \frac{(x-x0)(x-x2)}{x1-x0)(x1-x2)}\mathbf{f1} + \frac{(x-x0)(x-x1)}{(x-x0)(x-x1)}\mathbf{f2}$$

3.4.1 Algorithm

1.Start
2.read number of data (n)
3.read data x_i and y_i for i = 1 to n
4.set p =1
5.6.for j = 1 to n
7.if i is not equal to j then calculate p =p*(x_p -x_i)($x_{\ i} - x_j$)
8.end if next j
9.calculate y_p+p* y_i
next I
10display values of y_p as interpolated value
11.stop

3.4.2 Example

x		0	1	2	5
F(x)		2	3	12	147

$x_0 = 0$, $x_1 = 1$, $x_2 = 2$, $x_3 = 5$ $y_0 = 2$, $y_1 = 3$,
$y_2 = 12$, $y_3 = 147$

F(x) = $\frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)}(y_0) + \frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)}(y_1) + \frac{(x-x_0)(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_0)(x_2-x_3)}(y_2)$
+
$\frac{(x-x_0)(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)}(y_3)$

F(x) = $\frac{(x-1)(x-2)(x-5)}{(0-1)(0-2)(0-5)}(2) + \frac{(x-0)(x-2)(x-5)}{(1-0)(1-5)(1-2)}(3) + \frac{(x-0)(x-1)(x-5)}{(2-0)(2-1)(2-5)}(12) + \frac{(x-0)(x-1)(x-2)}{(5-9)(5-1)(5-2)}(147)$

F(x) = $\frac{-1}{5}(x-1)(x-2)(x-5) + \frac{3}{4}x(x-2)(x-5) + \frac{-2}{1}x(x-1)(x-5) + \frac{49}{20}x(x-1)(x-2)$

F(x) = $(20x^3+20^2 - 20x + 40)$ = $\boldsymbol{x^3+x^2-x-2}$ is required equation.

Advantages and disadvantages
1. Even when the arguments are not evenly spaced, this formula is used to find the function's value.
2. This formula is used to calculate the value of the independent variable x that corresponds to a given function value.

Disadvantages
1. In a LaGrange polynomial, changing the degree necessitates a thorough recalculation of all terms.

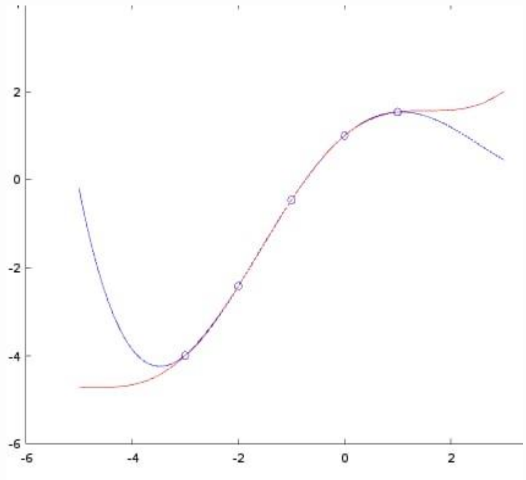
2. The formula for a polynomial of the high degree includes a significant number of multiplications, making the operation sluggish.

3. The degree of polynomial is chosen at the start of the Lagrange Interpolation. As a result, determining the degree of approximating a polynomial that is appropriate for a particular set of tabulated points is tricky.

3.5 Newton Dividend Difference Interpolation

Interpolation is an estimation of a value within two known values in a sequence of values.
Newton's divided difference interpolation formula is a 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
Then the first divided difference is given by

$$f[x_0, x_1] = \frac{f(x_1)-f(x_0)}{x_1-x_0}$$



Divided differences are symmetric with respect to the arguments i.e **independent of the order of arguments.**

$$f[x_0, x_1, x_2] = \frac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_0}$$

so,

$$\mathbf{f\,[x_0, x_1]=f[x_1, x_0]}$$

$$\mathbf{f\,[x_0, x_1, x_2]=f[x_2, x_1, x_0]=f[x_1, x_2, x_0]}$$

By using first divided difference, second divided difference as so on. A table is formed which is called the divided difference table.

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

3.5.1 Example

x	2	4	9	10
F(x)	4	56	711	980

Solution:

x	f(x)	F(x_0, x_1)	F(x_0, x_1, x_2)	F(x_0, x_1, x_2, x_3)
2	f(x_0)= 4	$\frac{56-4}{4-2}$ =26		
4	$f(x_1)$ = 56	$\frac{711-56}{9-4}$ = 131	$\frac{131-26}{9-2}$ =15	$\frac{23-15}{10-2}$ =1
9	$f(x_2)$ = 711	$\frac{980-711}{10-9}$ = 269	$\frac{269-131}{10-4}$ = 23	
10	$f(x_3)$ = 980			

F(x_0, x_1) = $\frac{f(x_1)-f(x_0)}{x_1-x_0}$

F(x_0, x_1, x_2) = $\frac{f(x_1,x_2)-f(x_0,x_1)}{x_2-x_0}$ …….. upto so on.

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) + $(x - x_0)$ $(x - x_1)$ $(x - x_2)$ f($x_0, x_1, x_2\ \ x_3$)

F(x) = 4 +$x - 2)(26) +\ (x-2)(x-4)(15) + (x-2)(x-4)(x-9)(1)$

F(x) = 4 + $(x-2)(x^2 +\ 2x + 2)$

F(x) = $x^3 - 2x$

This is required polynomial.

3.6 Spline Interpolation

spline interpolation is a form of interpolation where the interpolant is a special type of piecewise polynomial called a spline. That is, instead of fitting a single, high-degree polynomial to all of the values at once, spline interpolation fits low-degree polynomials to small subsets of the values, for example, fitting nine cubic polynomials between each of the pairs of ten points, instead of fitting a single degree-ten polynomial to all of them.

Linear interpolation

The idea is that we are given a set of numerical points and function values at these points. The task is to use the given set and approximate the function's value at some different points. That is, given x_i where $i = 0, \ldots, n-1$ our task is to estimate $f(x)$ for $x_0 \geq x \leq x_n$. Of course, we may require going outside of the range of our set of points, which would require extrapolation (or projection outside the known function values).

Almost all interpolation techniques are based around the concept of function approximation. Mathematically, the backbone is simple:

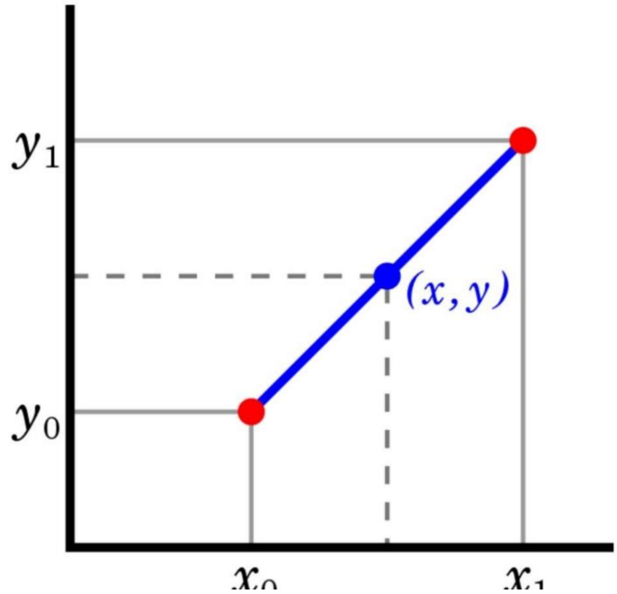
$f(x) = y_i + \frac{y_{i+1}-y_i}{x_{i+1}-x_i}(x - x_i)$

The above expression tells us that the value of the function we are approximating at point *x* will be around y_i and that $x_i \geq x \leq x_{i+1}$. We can re-write this expression as follows:

$$s_k(x) = a_k + b_k(x - x_k)$$

where we have substituted y_i for a_k and $\frac{y_{i+1}-y_i}{x_{i+1}-x_i}$ for b_k. Our linear interpolation is now taking a form of linear regression around a_k.

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



3.6.1 Quadratic spline

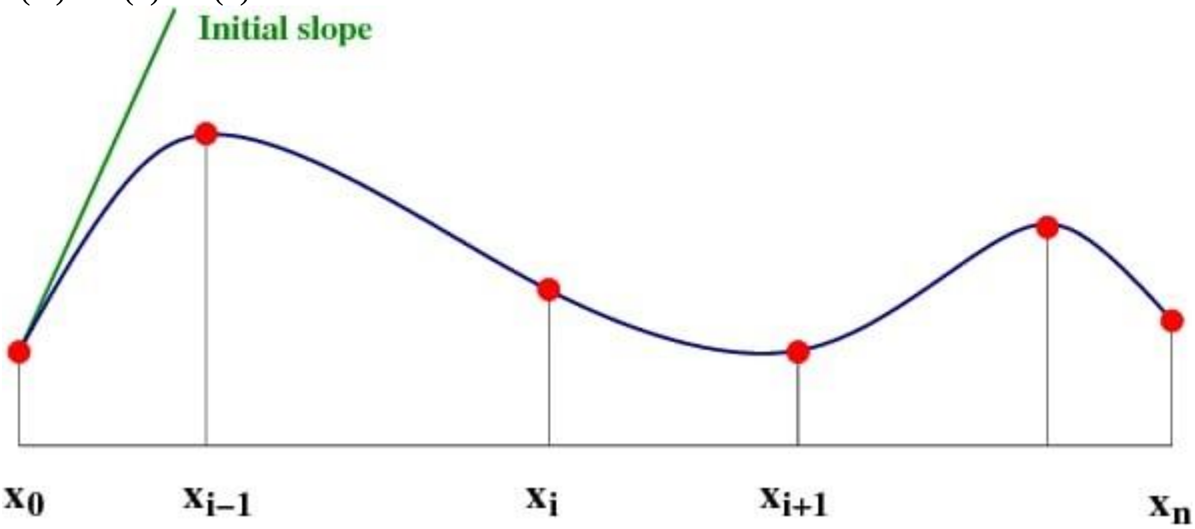
A Quadratic Spline is the creation of a set of polynomial functions that are quadratic, or, easier to understand, follow the format $f(x)=ax^2+bx+c$, where a, b and c are the values

To create the splines, it is .obtained while doing the Splines to create the desired functions necessary for the user to provide 2 or more points, as with 1 point it is impossible to calculate because the splines, as mentioned before, return a set of functions that contain n-1 functions. .Given 2 points, the set contains only 1 function, as with 1, it would contain 0

As Example, to create splines the User would Input the following set of point

4	3	1	0	**X**
2.3	3.6	1	2.5-	**Y**

To create a Linear Spline, we have 1 Possible method to find the desired equation Values**:**
A Matrix, which is Built using the values. This Matrix has the size of (n-1) *3
With the Matrix Method we have
The first (n-1) *2 equations correspond to the given values equations following the format
F(xi) = ai(x)²+bi(x)+ci



Example

x	y
2	3
3	6
4	9
7	18

We have three knows in this equation.

N +1` = 4

N= 3

Number of equation = 3n = 3(3) = 9

$a_1x^2 + b_1x + c_1$ $\qquad$ $2 \leq x \leq 3$

$a_2x^2 + b_2x + c_2$ $3 \leq x \leq 4$

$a_3x^2 + b_3x + c_3$ $4 \leq x \leq 7$

$4a_1 + 2b_1 + c_1 = 3$ **…………eq 1**

$9a_1 + 3b_1 + c_1 = 6$ **…………eq 2**

$9a_2 + 3b_2 + c_2 = 6$**…………eq 2**

$16a_2 + 3b_2 + c_2 = 9$ **…………eq 4**

$16a_3 + 4b_3 + c_3 = 9$**…………eq 5**

$49a_3 + 7b_3x + c_3 = 12$ **…………eq 6**

$\frac{d}{dx}(a_1x^2 + b_1x + c_1) = \frac{d}{dx}(a_2x^2 + b_2x + c_2)$ at x=3

$2xa_1 + b_1 = 2xa_2 + b_2$

$6a_1 + b_1 - 6a_2 - b_2 = 0$**…………eq 7**

$\frac{d}{dx}(a_2x^2 + b_2x + c_2) = \frac{d}{dx}(a_3x^2 + b_3x + c_3)$ at x=4

$8a_2 + b_2 - 8a_3 - b_3 = 0$**…………eq 8**

$|x_1 - x_0| \leq |x_4 - x_3|$

$|3 - 2| \leq |7 - 4|$

So $a_1 = 0$**…………eq 9**

Chapter 4

INTEGRATION

4.1 Trapezoidal rule

In mathematics, the trapezoidal rule, also known as the trapezoid rule or trapezium rule is a technique for approximating the definite integral in numerical analysis. The trapezoidal rule is an integration rule used to calculate the area under a curve by dividing the curve into small trapezoids. The summation of all the areas of the small trapezoids will give the area under the curve. Let us understand the trapezoidal rule formula and its proof using examples in the upcoming section.

We apply the trapezoidal rule formula to solve a definite integral by calculating the area under a curve by dividing the total area into little trapezoids rather than rectangles. This rule is used for approximating the definite integrals where it uses the linear approximations of the functions. The trapezoidal rule takes the average of the left and the right sum.

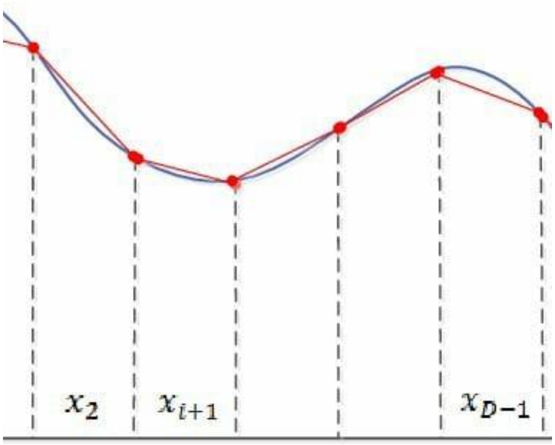
Let y = f(x) be continuous on [a, b]. We divide the interval [a, b] into n equal subintervals, **,**each of width, h = (b - a)/n

such that a = x0 < x1 < x2 < ⋯ < xn =

Área = (h/2) [y0 + 2 (y1 + y2 + y3 + …. + yn-1) + yn]

,were

y0, y1, y2…. are the values of function at x = 1, 2, 3….? Respectively



4.1.1 Example

$\int_0^1 \frac{1}{1+x^2}.$

Find error.

$\frac{upper-lower}{interval}$ =h

X	Y = F(x)

x_0 =0.2	F $(x_0) = \frac{1}{1-0}$=1
x_1=0.2	F$(x_0) = \frac{1}{1+0.2^2} = 0.9615$
x_2=0.4	F$(x_0) = \frac{1}{1+0.4^2}$=0.86209
x_3 =0.6	F$(x_0) = \frac{1}{1+0.6^2}$=0.7352
x_4 =0.8	F$(x_0) = \frac{1}{1+0.8^2}$=0.6097
x_5 = 1	F$(x_0) = \frac{1}{1+1} = 0.5$

$\int_0^1 \frac{1}{1+x^2} = \frac{h}{2}(f(x_0) + (fx_5) + 2(f(x_1) + f(x_2) + f(x_3) + f(x_4))$

$\frac{upper-lower}{interval}$ =h

h= $\frac{1-0}{5}$

= $\frac{0.2}{2}(1 + \frac{1}{2} + 2(0.9615 + 0.86209 + 0.7352 + 0.6097)$

= 0.783732

Actual value: $\int_0^1 \frac{1}{1+x^2}$ =$\tan^{-1} x]_0^1$ =0.785398

Error =actual value - approximate value

= 0.785398-0.783732 = 0.001666

4.1.2 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

Simpson s rule.

Simpson’s rule is one of the numerical methods which is used to evaluate the definite integral. Usually, to find the definite integral, we use the fundamental theorem of calculus, where we have to apply the antiderivative techniques of integration. However, sometimes, it isn’t easy to find the antiderivative of an integral, like in Scientific Experiments, where the function has to be determined from the observed readings. Therefore, numerical methods are used to approximate the integral in such conditions. Other numerical methods used are trapezoidal rule, midpoint rule, left or right approximation using Riemann sums.
Simpson’s rule methods are more accurate than the other numerical approximations and its formula for n+1 equally spaced subdivision is given by;

$\boldsymbol{\int_a^b f(x)dx \approx s_n = \frac{\Delta x}{3}[(f(x_o) + 4f(x_1) + 2f(x_2) + +4f(x_3) + \cdots .... 2f(x_{n-2}) + 4f(x_{n-1}) +}$

$\boldsymbol{f}((\boldsymbol{x_n})]$
Where n is the even number, △x = (b – a)/n and xi = a + i△x

If we have f(x) = y, which is equally spaced between [a, b] and if a = x0, x_1 = x_0 + h, x_2 = x_0 + 2h …., x_n = x_0 + nh, where h is the difference between the terms. Or we can say that y_0 = f (x_0), y_1 = f(x_1), y_n= f(x_n),……, = fx_n nh, are the analogous values of y with each value of x.

Simpson's one three rule
Simpson's one three rule is an extension of trapezoidal rule in which the integrand is approxi mated by a second order polynomial. Simpson rule can be derived from various ways using n ewton's divided difference polynomial, Lagrange polynomial and the method of coefficients. Simpsons one three rile is defined by

$$\int_a^b \boldsymbol{f(x)}\mathbf{dx} = \frac{3h}{8}[\boldsymbol{y_0}+\boldsymbol{y_n}) + \mathbf{4}(\boldsymbol{y_1} + \boldsymbol{y_3} + \boldsymbol{y_5} + \boldsymbol{y_7} \ldots .\boldsymbol{y_{n-1}}) + \mathbf{2}(\boldsymbol{y_2} + \boldsymbol{y_4} + \boldsymbol{y_6} + \boldsymbol{y_8} \ldots .. \boldsymbol{y_{n-2}})]$$

Example

x	1	2	3	4	5	6	7
F(x)	2.105	2.808	3.614	4.604	5.857	7.451	9.467

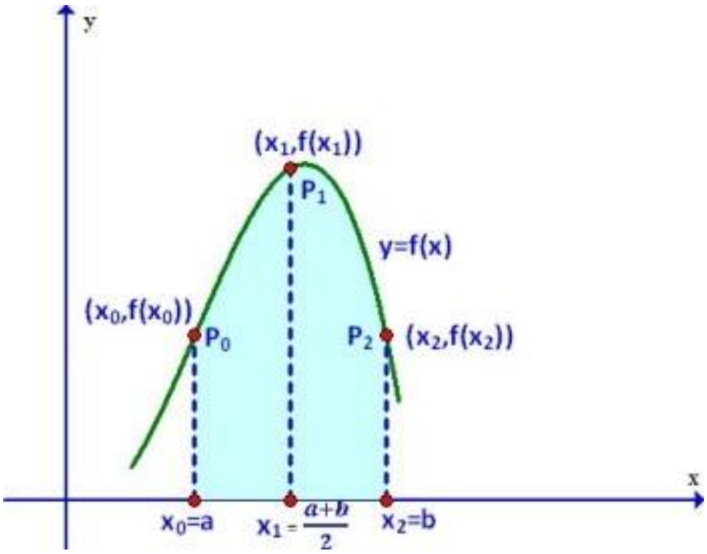
N = 6 h = $\frac{b-a}{n}$ =$\frac{7-1}{6}$ = 1
trapezoidal rule
$\int_1^7 f(x)\mathrm{dx} = \frac{h}{2}[(y_0+y_6) + 2(y_1 + y_2 + y_3 + y_4 + y_5)]$
$=\frac{1}{2}[60.24] = 30.12$

$\int_1^7 f(x)\mathrm{dx} = = \frac{h}{3}[(y_0+y_6) + 4(y_1 + y_3 + y_5) + 2(y_2+y_4)$
$= \frac{1}{3}[89.9660] = 29.9887$

Simpson rule $\frac{3}{8}$

Simpson's rule based on the cubic interpolation rather than the quadratic interpolation . Simp son's three by 8 rule is given by

$$\int_a^b \boldsymbol{f(x)}\mathbf{dx} = \frac{3h}{8}[\boldsymbol{y_0}+\boldsymbol{y_n}) + \mathbf{3}(\boldsymbol{y_1} + \boldsymbol{y_2} + \boldsymbol{y_4} + \boldsymbol{y_5} \ldots .\boldsymbol{y_{n-1}}) + \mathbf{2}(\boldsymbol{y_3} + \boldsymbol{y_6} + \boldsymbol{y_9} + \boldsymbol{y_{12}} \ldots .. \boldsymbol{y_{n-3}})]$$



Example:

x	1	2	3	4	5	6	7
F(x)	2.105	2.808	3.614	4.604	5.857	7.451	9.467

N = 6 h = $\frac{b-a}{n}$ =$\frac{7-1}{6}$ = 1

$\int_1^7 f(x)\text{dx} = \frac{3h}{8}[y_0+y_6) + 3(y_1 + y_2 + y_4 + y_5) + 2(y_3)]$

= 29.9887

Drawback:

1. It is obviously not accurate, i.e. there will always (except in some cases such as with the area under straight lines) be an error between it and the actual integral

2. Integrals allow you to get exact answers in terms of fundamental constants, this is not possible with Simpson's

3. It is necessary (often) to use a large number of ordinates to gain a good approximation to the real integral.