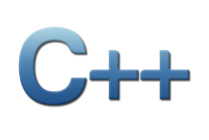
Scientific Computing Practical File[MC 204]

Name: **Ashish Gupta**

Roll Number: 2k16/MC/**023**

Department of Applied Mathematics

Delhi Technological University

File Content:

|  |  |  |  |
| --- | --- | --- | --- |
| S.no. | Date | Topic | Signature |
| 1 |  | Bisection Method |  |
| **2** |  | Secant Method |  |
| **3** |  | Regular Falsi Method |  |
| **4** |  | Newton Raphson Method |  |
| **5** |  | Gauss Jordan/Elimination Method |  |
| **7** |  | Newton Forward Interpolation |  |
| **8** |  | Newton Backward Interpolation |  |
| **9** |  | Lagrange Interpolation |  |
| **10** |  | Method of Numerical integration   * Trapezoidal Rule * Simpson 1/3rd * Simpson 3/8th |  |

**Practical 1**: Implement the **Bisection Method** on a given function.

**Theory:**

**Bisection method** is the simplest among all the numerical schemes to solve the transcendental equations.

Consider a transcendental equation f (x) = 0 which has a zero in the interval [a,b] and **f (a) \* f (b) < 0.** Bisection scheme computes the zero, say c, by repeatedly halving the interval [a,b]. That is, starting with

**c = (a+b) / 2**

the interval [a,b] is replaced either with [c,b] or with [a,c] depending on the sign of f (a) \* f (c) . This process is continued until the zero is obtained. Since the zero is obtained numerically the value of c may not exactly match with all the decimal places of the analytical solution of f (x) = 0 in the interval [a,b].

**C++ Code:**

#include<iostream>

#include<cmath>

using namespace std;

**float fun(float x)**

{

return ((x\*x\*x)-(4\*x)-9);

}

**int main()**

{

///cout<<fun(2)<<endl;

float error,inter\_a,inter\_b,iter,num;/\*the function is given in the program\*/

cout<<"Enter error interval pt a and interval point b"<<endl;

cin>>error>>inter\_a>>inter\_b;

cout<<"Enter Max Iterations : ";

cin>>iter;

if(fun(inter\_a)\*fun(inter\_b)>0){

cout<<"Invalid input intervals"<<endl;

return 0;

}

**while(iter>0)**

{

///cout<<inter\_a<<" "<<inter\_b<<endl;

num=(inter\_a+inter\_b)/2;

cout<<"For x = "<<num<<" ";

if(abs(fun(num))<error){

break;

}

if(fun(inter\_a)<0)

{

cout<<"Value of function at "<<num<<" is : "<<fun(num)<<endl;

if(fun(num)>0)

{

inter\_b=num;

iter--;

continue;

}

else{

inter\_a=num;

iter--;

continue;

}

}

else{

cout<<"Value of function at "<<num<<" is : "<<fun(num)<<endl;

if(fun(num)<0)

{

inter\_b=num;

iter--;

continue;

}

else{

inter\_a=num;

iter--;

continue;

}

} }

cout<<"The final root is : "<<num<<endl;

return 0

}

**Output:**

Enter error interval pt a and interval point b

.001

1 4

Enter Max Iterations : 6

For x = 2.5 Value of function at 2.5 is : -3.375

For x = 3.25 Value of function at 3.25 is : 12.3281

For x = 2.875 Value of function at 2.875 is : 3.26367

For x = 2.6875 Value of function at 2.6875 is : -0.339111

For x = 2.78125 Value of function at 2.78125 is : 1.38895

For x = 2.73438 Value of function at 2.73438 is : 0.506893

**The final root is : 2.73438**

Process returned 0 (0x0) execution time : 12.412 s

**Practical 2**: Implement the **Secant Method** on a given function.

**Theory:**

In numerical analysis, the **secant method** is a root-finding algorithm that uses a succession of roots of secant lines to better approximate a root of a function f. The secant method can be thought of as a finite difference approximation of Newton's method.


x_n
=x_{n-1}-f(x_{n-1})\frac{x_{n-1}-x_{n-2}}{f(x_{n-1})-f(x_{n-2})}
=\frac{x_{n-2}f(x_{n-1})-x_{n-1}f(x_{n-2})}{f(x_{n-1})-f(x_{n-2})}


where x0 and x1 are initial values.

**Matlab Code:**

syms x

y=x\*exp(x)-3;

x1=1;

x2=1.5;

x=x2;

f2=double(subs(y));

x=x1;

f1=double(subs(y));

count=0;

while count<6

**x3=calcFunction(y,x2,x1);**

fprintf('Value of x3 is : ')

x=x3

fprintf('\n')

fx3=double(subs(y));

if fx3\*f1<0

x2=x3;

else

x1=x2;

x2=x3;

end

count=count+1;

end

**fprintf('The Final Root is : ')**

**x3**

function **factor=calcFunction(fun,x2,x1)**

x=x2;

f2=double(subs(fun));

x=x1;

f1=double(subs(fun));

factor=x2-((x2-x1)./(f2-f1))\*f2;

end

**Output:**

Value of x3 is : =

0.2500

Value of x3 is : =

0.1864

Value of x3 is : =

0.2061

Value of x3 is : =

0.2003

Value of x3 is : =

0.2020

The Final Root is : =

0.2020

**Practical 3**: Implement the **Regula-Falsi Method** on a given function.

**Theory:**

In numerical analysis, the **Regula-Falsi method** is a root-finding algorithm that uses a succession of roots of secant lines to better approximate a root of a function f. The secant method can be thought of as a finite difference approximation of Newton's method.


x_n
=x_{n-1}-f(x_{n-1})\frac{x_{n-1}-x_{n-2}}{f(x_{n-1})-f(x_{n-2})}
=\frac{x_{n-2}f(x_{n-1})-x_{n-1}f(x_{n-2})}{f(x_{n-1})-f(x_{n-2})}


where x0 and x1 are initial values.

**Matlab Code:**

syms x

y=x\*exp(x)-3;

x1=1;

x2=1.5;

x=x2;

f2=double(subs(y));

x=x1;

f1=double(subs(y));

count=0;

while count<6

**x3=calcFunction(y,x2,x1);**

fprintf('Value of x3 is : ')

x=x3

fprintf('\n')

fx3=double(subs(y));

if fx3\*f1<0

x2=x3;

else

x1=x2;

x2=x3;

end

count=count+1;

end

**fprintf('The Final Root is : ')**

**x3**

function **factor=calcFunction(fun,x2,x1)**

x=x2;

f2=double(subs(fun));

x=x1;

f1=double(subs(fun));

factor=x2-((x2-x1)./(f2-f1))\*f2;

end

**Output:**

Value of x3 is : =

0.2500

Value of x3 is : =

0.1864

Value of x3 is : =

0.2061

Value of x3 is : =

0.2003

Value of x3 is : =

0.2020

The Final Root is : =

0.2020

**Practical 4**: Implement the **Newton Raphson Method** on a given function.

**Theory:**

In numerical analysis, **Newton's method** (also known as the **Newton**–**Raphson method**), named after Isaac **Newton** and Joseph **Raphson**, is a **method** for finding successively better approximations to the roots (or zeroes) of a real-valued function. The **method** can also be extended to complex functions and to systems of equations.

x_{1} = x_0 - \frac{f(x_0)}{f'(x_0)} \,.

The process is repeated as

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

Until a sufficiently accurate value is reached.

**Matlab Code:**

syms x

y=x-(23/10)\*(x^(10/23)-exp(1))/(x^(-13/23));

i=0;

x=8;

while i<10

xkplus1=eval(subs(y));

fprintf('the value of xk is %d and value of xk+1 is :%d ',x,xkplus1);

x=xkplus1;

i=i+1;

fprintf('\n')

eval(subs(x^(10/23)-exp(1)))

end

**Output:**

the value of xk is 8 and value of xk+1 is :9.851840e+00

ans =

-0.0145

the value of xk is 9.851840e+00 and value of xk+1 is :9.973756e+00

ans =

-5.0550e-05

the value of xk is 9.973756e+00 and value of xk+1 is :9.974182e+00

ans =

-6.1101e-10

the value of xk is 9.974182e+00 and value of xk+1 is :9.974182e+00

ans =

4.4409e-15

the value of xk is 9.974182e+00 and value of xk+1 is :9.974182e+00

ans =

1.3323e-15

the value of xk is 9.974182e+00 and value of xk+1 is :9.974182e+00

ans =

4.8850e-15

the value of xk is 9.974182e+00 and value of xk+1 is :9.974182e+00

ans =

4.8850e-15

**the value of xk is 9.974182e+00 and value of xk+1 is :9.974182e+00**

**ans =**

**1.3323e-15**

**>>**

**Practical 5**: Implement the **Gauss Jordan/Elimination Method** on a given function.

**Theory:**

In linear algebra, **Gaussian Jordan** (also known as row reduction) is an algorithm for solving systems of linear equations. It is usually understood as a sequence of operations performed on the corresponding matrix of coefficients.

$\displaystyle [A \;\; {\mathbf b}] = \left[\begin{array}{cccc\vert c}
a_{11} & ...
... \vdots & \vdots \\
a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{array}\right]$

*to an upper triangular form*

$\displaystyle \left[\begin{array}{cccc\vert c}
c_{11} & c_{12} & \cdots & c_{1n...
... \ddots & \vdots & \vdots \\
0 & 0 & \cdots & c_{mn} & d_m \end{array}\right].$

**Matlab Code:**

% A is a matrix and apply gauss jordan method

syms x y z

**eqn1= 2\*x+3\*y-z==2;**

**eqn2= 3\*x+ 7\*y+4\*z==11;**

**eqn3= x-5\*y-3\*z==-4;**

% covert the code to matrix with A and B

[A,B]=equationsToMatrix(eqn1,eqn2,eqn3);

%convert A,b to augmented matrix

augmat=[A,B];

%call the rref function

aug=rref(augmat);

%extract the elements

**ans = aug\*[0;0;0;1]**

**Output:**

**ans =**

**76/59**

**15/59**

**79/59**

>>

**Practical 6**: Implement the **Newton Forward Interpolation** on a given function.

**Theory[Newton Forward]:**

**Newton Forward Interpolation**

Newton's forward difference formula is a finite difference identity giving an interpolated value between tabulated points Description: {f_p} in terms of the first value Description: f_0and the powers of the forward differenceDescription: Delta. ForDescription: a in [0,1], the formula states

|  |  |
| --- | --- |
| Description:  f_a=f_0+aDelta+1/(2!)a(a-1)Delta^2+1/(3!)a(a-1)(a-2)Delta^3+.... |  |

When written in the form

|  |  |
| --- | --- |
| Description:  f(x+a)=sum_(n=0)^infty((a)_nDelta^nf(x))/(n!) |  |

with Description: (a)_n the falling factorial, the formula looks suspiciously like a finite analogue of a Taylor series expansion.

**Matlab Code:**

>> X = [-3; -2; -1; 1; 2; 3];

>> F = [18; 12; 8; 6; 8; 12];

>> newtonforward(X, F)

function [A] = newtonforward(X,F)

n = length(X);

syms x

h = X(2)-X(1);

u = (x-X(1))/h;

for I = 1:n

A(i,1)=F(i);

end

for j = 2:n

for i = j:n

A(i,j) = A(i,j-1)-A(i-1,j-1);

end

end

y = 0\*x+F(1);

%t = 0\*x+1;

for j = 2:n-1

t = A(j,j);

for i = 1:j-1

t = (t\*(x - X(i)))/(i\*h);

end

y = y + t;

end

end

**Output:**

ans =

18 0 0 0 0 0

12 -6 0 0 0 0

8 -4 2 0 0 0

6 -2 2 0 0 0

8 2 4 2 2 0

12 4 2 -2 -4 -6

2\*x + (2\*x - 6)\*(x - 2) + (x/3 - 1)\*(x - 1)\*(x - 2) + (x/12 - 1/4)\*(x - 1)\*(x + 1)\*(x - 2) + 6

ans=

12.1696

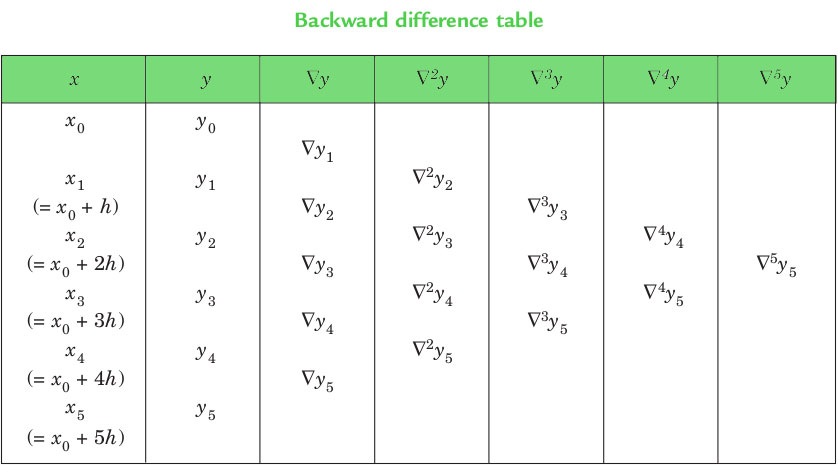
>>

**Practical 7**: Implement the **Newton Backward Interpolation** on a given function.

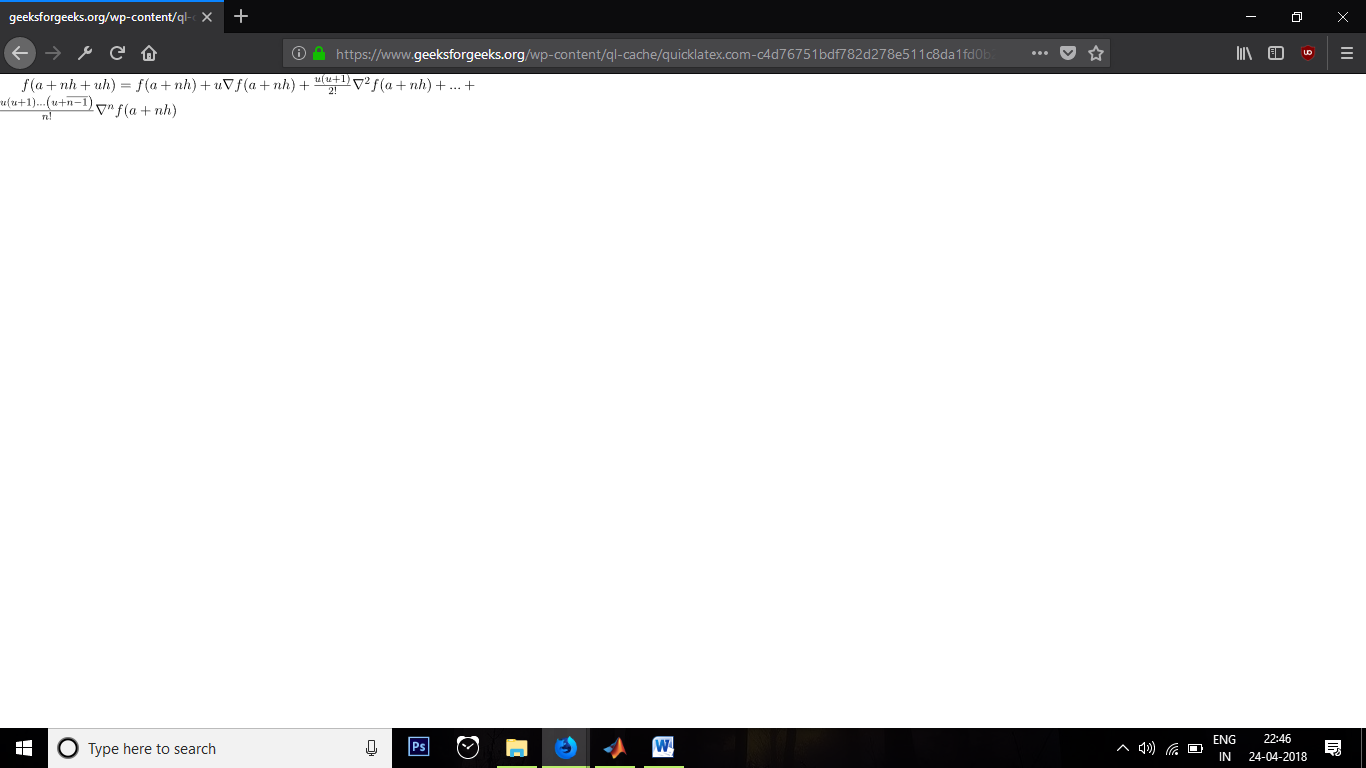
**Theory[Newton Backward]:**

**Newton Backward Interpolation**

**Backward Differences** : 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 :



**NEWTON’S GREGORY BACKWARD INTERPOLATION FORMULA** :



This formula is useful when the value of f(x) is required near the end of the table. h is called the interval of difference and **u = ( x – an ) / h**, Here an is last term.

**C++ Code:**

// CPP Program to interpolate using

// newton backward interpolation

#include <bits/stdc++.h>

using namespace std;

// Calculation of u mentioned in formula

float u\_cal(float u, int n)

{

    float temp = u;

    for (int i = 1; i < n; i++)

        temp = temp \* (u + i);

    return temp;

}

// Calculating factorial of given n

int fact(int n)

{

    int f = 1;

    for (int i = 2; i <= n; i++)

        f \*= i;

    return f;

}

int main()

{

    // number of values given

    int n = 5;

    float x[] = { 1891, 1901, 1911,

                  1921, 1931 };

    // y[][] is used for difference

    // table and y[][0] used for input

    float y[n][n];

    y[0][0] = 46;

    y[1][0] = 66;

    y[2][0] = 81;

    y[3][0] = 93;

    y[4][0] = 101;

    // Calculating the backward difference table

    for (int i = 1; i < n; i++) {

        for (int j = n - 1; j >= i; j--)

            y[j][i] = y[j][i - 1] - y[j - 1][i - 1];

    }

    // Displaying the backward difference table

    for (int i = 0; i < n; i++) {

        for (int j = 0; j <= i; j++)

            cout << setw(4) << y[i][j]

                 << "\t";

        cout << endl;

    }

    // Value to interpolate at

    float value = 1925;

    // Initializing u and sum

    float sum = y[n - 1][0];

    float u = (value - x[n - 1]) / (x[1] - x[0]);

    for (int i = 1; i < n; i++) {

        sum = sum + (u\_cal(u, i) \* y[n - 1][i]) /

                                     fact(i);

    }

    cout << "\n Value at " << value << " is "

         << sum << endl;

    return 0;

}

**Output:**

46

66 20

81 15 -5

93 12 -3 2

101 8 -4 -1 -3

Value at 1925 is 96.8368

Process returned 0 (0x0) execution time : 2.672 s

**Practical 8**: Implement the **Newton Divided Difference Formula** on a given table of values.

**Theory:**

**Newton Divided Difference Formula:**

The newton divided difference is that the function **f(x)** is linear then we have

Let

|  |  |
| --- | --- |
| pi_n(x)=product_(k=0)^n(x-x_k), | (1) |

then

|  |  |
| --- | --- |
| f(x)=f_0+sum_(k=1)^npi_(k-1)(x)[x_0,x_1,...,x_k]+R_n, | (2) |

where [x_1,...]is a[**divided difference**](http://mathworld.wolfram.com/DividedDifference.html), and the remainder is

|  |  |
| --- | --- |
| R_n(x)=pi_n(x)[x_0,...,x_n,x]=pi_n(x)(f^((n+1))(xi))/((n+1)!) | (3) |

for. x_0<xi<x_n

**Matlab Code:**

function Ndivided(X, Y, x0)

table = zeroes(length(Y), 5);

table(:,1)= Y;

for i=2:5

for j=1:length(Y)-i+1

table(j,i) = table(j+1, i-1) – table(j,i-1)

end

end

ind = max(find(x , x0))

h= X(2) – X(1)

p=x0 – X(ind)/h

val =Y(ind)+ p\*table(ind,2) + (p\*(p-1)\*table(ind, 3)/2) + (p\*(p-1)\*(p-2)\*table(ind,4)/6)

+(p\*(p-1)\*(p-2)\*(p-3)\*table(ind,5)/24);

disp(table)

disp(val)

end

X=input()

F=input()

V=input()

Ndivided(X,F,V)

**Input:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| >> X= [0 1 3 4 7 ]  >> F= [1 3 49 129 813] |  |  |  |  |  |
| >> V=0.3 |  |  |  |  |  |

**Output:**

ans=

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 |  |  |  |
|  |  | 2 |  |  |
| 1 | 3 |  | 7 |  |
|  |  | 23 |  | 3 |
| 3 | 49 |  | 19 |  |
|  |  | 80 |  | 3 |
| 4 | 129 |  | 37 |  |
|  |  | 228 |  |  |
| 7 | 813 |  |  |  |

ans =

1.831

>>

**Practical 9**: Implement **Lagrange Interpolation** on given table of values.

**Theory:**

The Lagrange interpolating polynomial is the [polynomial](http://mathworld.wolfram.com/Polynomial.html) P(x)of degree <=(n-1)that passes through the npoints (x_1,y_1=f(x_1)), (x_2,y_2=f(x_2)), ..., (x_n,y_n=f(x_n)), and is given by

|  |  |
| --- | --- |
| P(x)=sum_(j=1)^nP_j(x), | (1) |

where

|  |  |
| --- | --- |
| P_j(x)=y_jproduct_(k=1; k!=j)^n(x-x_k)/(x_j-x_k). | (2) |

Written explicitly,

|  |  |  |
| --- | --- | --- |
| P(x) | = | ((x-x_2)(x-x_3)...(x-x_n))/((x_1-x_2)(x_1-x_3)...(x_1-x_n))y_1+((x-x_1)(x-x_3)...(x-x_n))/((x_2-x_1)(x_2-x_3)...(x_2-x_n))y_2+...+((x-x_1)(x-x_2)...(x-x_(n-1)))/((x_n-x_1)(x_n-x_2)...(x_n-x_(n-1)))y_n. |

**Matlab Code:**

function[y] = lagrange(x,n)

ax=input(‘X');

ay=input('Y);

y=0;

for i=1:n

nr=1;

dr=1;

for j=1:n

if(j~=i)

nr=nr\*(x-ax(j));

dr=dr\*(ax(i)-ax(j));

end

end

y=y+(nr/dr)\*(ay(i));

end

fprintf('ans: %f',y);

**Input & Output:**

Lagrange(301,4)

X=[300 304 305 307]

Y: [2.4771 2.4829 2.4843 2.4871]

ans: 2.478597

**Practical 10**: Implement **Trapezoidal method, Simpson 1/3rd method, Simpson 3/8 method interpolation** on a given table of values.

**Theory;**

In numerical analysis, the trapezoidal rule (also known as the trapezoid rule or trapezium rule) is a technique for approximating the definite integral: The trapezoidal rule works by approximating the region under the graph of the function Description: f(x) as a trapezoid and calculating its area. It follows that

**Simpson's rule** is a method for numerical integration, the numerical approximation of definite integrals.

**Simpsons 1/3rd Rule :**

Description: \int_a^b f(x) \, dx\approx
\tfrac{h}{3}\bigg[f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+2f(x_4)+\cdots+4f(x_{n-1})+f(x_n)\bigg]
=\tfrac{h}{3}\sum_{j=1}^{n/2}\bigg[f(x_{2j-2})+4f(x_{2j-1})+f(x_{2j})\bigg].

with error as 190(*b*−*a*2)5|*f*4(*z*)|

**Simpsons 3/8th Rule:**

Description:  \int_{a}^{b} f(x) \, dx \approx \tfrac{3h}{8}\left[f(x_0) + 3f(x_1) + 3f(x_2) 
+ 2f(x_3) + 3f(x_4) + 3f(x_5) + 2f(x_6) + ... + f(x_n)\right] .

Note, we can only use this if Description: n is a **multiple** of **three**.

with error as ∣∣(*b*−*a*)56480*f*4(*z*)∣∣

**Matlab Code:**

function [ result ] = GenricSimpson( y, a, b, n )

syms x;

h = (b-a)/n;

if(h==1)

f0 = subs(f, x, a);

f1 = subs(f, x, b);

f2 = 0;

x0 = a;

for i = 1:n-1

x2 = x0+i\*h;

f2 = f2 + subs(f, x, x2);

end

disp('The answer for integration is: ');

result = h\*(f0+f1)/2+ h\*f2

end

if(h==2)

def = int(y,a,b);

f = 0;

g = -subs(y,b);

N = n/2;

for i = 1:N

f = f + subs(y, a+((2\*i)-1)\*h);

g = g + subs(y, a+((2\*i)\*h));

end

fprintf(' Result Error')

result(1) = h\*(subs(y, a) + subs(y, b) + 4\*f + 2\*g)/3;

result(2) = abs(def - result(1));

end

end

if(h = 3)

def = int(y,a,b);

f = 0;

g = 0;

N = n/3;

for i = 1:n-1

if (mod(i,3) == 0)

g = subs(y,a+i\*h) + g;

else

f = subs(y,a+i\*h) + f;

end

end

fprintf(' Result Error')

result(1) = 3\*h\*(subs(y,a) + subs(y,b) + 3\*f + 2\*g)/8;

result(2) = abs(def - result(1));

end

end

end

**Input & Output:**

>> f = exp(-x^2)

f =

exp(-x^2)

>> GenricSimpson(f, 0, 1, 20)

ans =

0.7467

>> GenricSimpson(f, 0, 0.6, 7)

ans =

0.4743 0.0608

>> GenricSimpson(f, 0, 0.6, 7)

ans =

0.5194 0.015