**MC204**

**SCIENTIFIC COMPUTING**

**PRACTICAL FILE**



**Submitted by**

Himanshu Singh

2K16/MC/036

Department of Applied Mathematics

Delhi Technological University

**Submitted to:-**

Faculty Name – C.P. Singh

Programmed on MATLAB

**NO INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| S.no. | Date | Topic | Remarks |
| 1 |  | To implement Bisection method |  |
| **2** |  | To implement Secant method |  |
| **3** |  | To implement Regular falsi |  |
| **4** |  | To implement Newton raphson |  |
| **5** |  | To implement Gauss elimination |  |
| **6** |  | To implement Gauss Jordan |  |
| **7** |  | To implement Newton Forward |  |
| **8** |  | To implement Newton divided difference |  |
| **9** |  | To implement lagrange method |  |
| **10** |  | To implement method of Numerical integration   * Trapezoidal * Simpson 1/3rd * Simpson 3/8th |  |
| **11** |  | To implement Runge Kutta method |  |

**Practical 1: To implement Bisection Method**

**Theory**

**Algorithm**

* Algorithm:

1. Take I0 = (a0 , b0).
2. Take C1 = (a0 + b0) / 2
3. I1 = (a0 , c1) if f(c1)\*f(a0) < 0 else I1 = (c1,b0) if f(c1)\*f(b0) < 0
4. Subsequently find other intervals.

**CODE**

function [ roots ] = bisection(a,b,fun,n)

syms c

syms fa

syms fb

syms fc

syms x

c=(a+b)/2;

fa = subs(fun,x,a);

fb = subs(fun,x,b);

if(fa\*fb>0)

disp('not possible');

else

for I = 1:n

c = (a+b)/2;

fa = subs(fun,x,a);

fb = subs(fun,x,b);

fc = subs(fun,x,c);

if ((fa\*fc)<0)

b = c;

else if ((fc\*fb)<0)

a = c;

end

end

end

roots = c;

end

end

**OUTPUT - COMMAND WINDOW**

>> syms x

>> fun = x^3 - 4\*x - 9;

>> a=2;

>> b=3;

>> n=10;

>> bisection(a,b,fun,n)

ans =

2.7061

**Practical 2: To implement secant method**

**RECURRENCE RELATION**

In [numerical analysis](http://en.wikipedia.org/wiki/Numerical_analysis), the secant method is a [root-finding algorithm](http://en.wikipedia.org/wiki/Root-finding_algorithm) that uses a succession of [roots](http://en.wikipedia.org/wiki/Root_of_a_function) of [secant lines](http://en.wikipedia.org/wiki/Secant_line) to better approximate a root of a [function](http://en.wikipedia.org/wiki/Function_(mathematics)) f. The secant method can be thought of as a [finite difference](http://en.wikipedia.org/wiki/Finite_difference) approximation of [Newton's method](http://en.wikipedia.org/wiki/Newton%27s_method). The secant method is defined by the [recurrence relation](http://en.wikipedia.org/wiki/Recurrence_relation).

Description: 
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.

**CODE**

function [ roots ] = secant(a,b,func,n)

syms fa;

syms fb;

syms x;

syms c;

fa = double(subs(func, x, a));

fb = double(subs(func, x, b));

if(fa\*fb>0)

disp('not possible');

else

for i = 1:n

c= ((a\*fb)-(b\*fa))/(fb-fa);

a=b;

b=c;

fa = double(subs(func, x, a));

fb = double(subs(func, x, b));

answer = c;

end

end

roots= answer;

end

**OUTPUT – COMMAND WINDOW**

>> secant(0,1,fun,5)

root =

0.2500

root =

0.1864

root =

0.2061

root =

0.2003

root =

0.2020

ans =

0.2020

**Practical 3: To implement Regula-Falsi method**

**RECURRENCE RELATION – SAME AS SECANT METHOD**

In [numerical analysis](http://en.wikipedia.org/wiki/Numerical_analysis), the regular falsi method is a [root-finding algorithm](http://en.wikipedia.org/wiki/Root-finding_algorithm) that uses a succession of [roots](http://en.wikipedia.org/wiki/Root_of_a_function) of [regular falsi lines](http://en.wikipedia.org/wiki/Secant_line) to better approximate a root of a [function](http://en.wikipedia.org/wiki/Function_(mathematics)) f. The secant method can be thought of as a [finite difference](http://en.wikipedia.org/wiki/Finite_difference) approximation of [Newton's method](http://en.wikipedia.org/wiki/Newton%27s_method). The secant method is defined by the [recurrence relation](http://en.wikipedia.org/wiki/Recurrence_relation)

Description: 
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.

**CODE**

function [ roots ] = regularfalsi(a,b,func,n)

syms fa;

syms fb;

syms x;

syms c;

syms fc;

fa = double(subs(func, x, a));

fb = double(subs(func, x, b));

if(fa\*fb>0)

disp('not possible');

else

for i = 1:n

c= ((a\*fb)-(b\*fa))/(fb-fa);

fc = double(subs(func, x, c));

if(fc<0)

a=c;

fa=fc;

else if(fc>0)

b=c;

fb=fc;

answer = c;

end

end

end

roots= answer;

end

end

**OUTPUT – COMMAND WINDOW**

>> regularfalsi(0,1,fun,5)

root =

0.2500

root =

0.1864

root =

0.2061

root =

0.2003

root =

0.2020

ans =

0.2020

**Practical 4: To implement Newton-Raphson Method**

**Theory:** Newton's method (also known as the Newton–Raphson method), named after [Isaac Newton](http://en.wikipedia.org/wiki/Isaac_Newton) and [Joseph Raphson](http://en.wikipedia.org/wiki/Joseph_Raphson), is a method for finding successively better approximations to the [roots](http://en.wikipedia.org/wiki/Root_of_a_function) (or zeroes) of a [real](http://en.wikipedia.org/wiki/Real_number)-valued [function](http://en.wikipedia.org/wiki/Function_(mathematics)).

x : f(x) = 0 \,.

**FORMULA**

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

The process is repeated as

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

Until a sufficiently accurate value is reached.

**CODE**

clear all; close all; clc;

% INPUTS: Enter the following

% GRAPH

syms x;

x= -2:0.5:6;

f = inline('(x.^3)-6\*(x.^2)+11\*x-6'); % Function in f(x)=0

f1 = inline('3\*(x.^2)-12\*x+11'); % Derivative of the function

plot(x,(x.^3)-6\*(x.^2)+11\*x-6,'--rs') % Plotting the graph of the function

grid on;

% FIRST ROOT

disp('------- ------- FIRST ROOT OF f(x) ------ -------')

xn=0; % Initial guess

tol=.000001; count=0;

disp(sprintf('The initial approximation of the first root is %g',xn));

fxn=f(xn);

f1xn=f1(xn);

xn1=xn-fxn/f1xn;

while( abs(xn1-xn) > tol)

count=count+1;

if fxn==0

xn1=xn

disp(sprintf('The root is x = %g',xn1));

else

xn=xn1;

fxn=f(xn);

f1xn=f1(xn);

xn1=xn-(fxn/f1xn);

end

end

disp(sprintf('\n The root is x = %g',xn1));

disp(sprintf('\n Number of iterations are = %g',count));

% SECOND ROOT

disp('------- ------- SECOND ROOT OF f(x) ------ -------')

xn=5;% Initial guess

tol=.000001; count=0;

disp(sprintf('\nThe initial approximation of the second root is %g',xn));

fxn=f(xn);

f1xn=f1(xn);

xn1=xn-fxn/f1xn;

while( abs(xn1-xn) > tol)

count=count+1;

if fxn==0

xn1=xn

disp(sprintf('The equation is cos(x)-x^2-x and the root is x = %g',xn1));

else

xn=xn1;

fxn=f(xn);

f1xn=f1(xn);

xn1=xn-(fxn/f1xn);

end

end

disp(sprintf('\n The root is x = %g',xn1));

disp(sprintf('\n Number of iterations are = %g',count));

**OUTPUT- COMMAND WINDOW**

------- ------- FIRST ROOT OF f(x) ------ -------

The initial approximation of the first root is 0

The root is x = 1

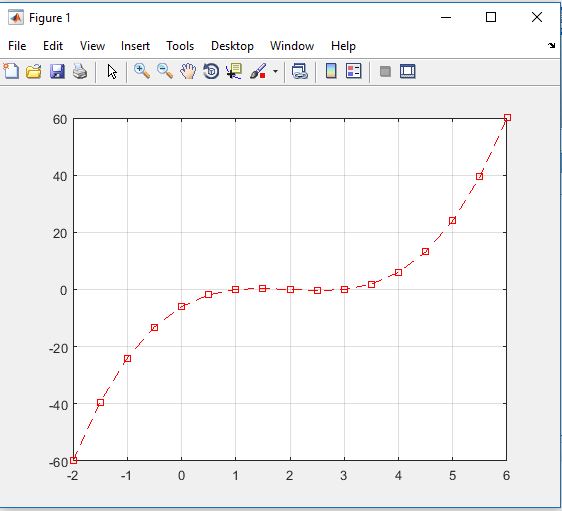
Number of iterations are = 6

------- ------- SECOND ROOT OF f(x) ------ -------

The initial approximation of the second root is 5

The root is x = 3

Number of iterations are = 7



**Precaution:**

1. **Important to draw a graph because it confirms us how many roots a equation can have so that we can find consecutive solutions**

**Practical 5 to implement Gauss Elimination Method**

**Theory:** To perform row reduction on a matrix, one uses a sequence of [elementary row operations](https://en.wikipedia.org/wiki/Elementary_row_operations) to modify the matrix until the lower left-hand corner of the matrix is filled with zeros, as much as possible. There are three types of elementary row operations:

1) Swapping two rows

2) Multiplying a row by a non-zero number

3) Adding a multiple of one row to another row.

Using these operations, a matrix can always be transformed into an [upper triangular matrix](https://en.wikipedia.org/wiki/Triangular_matrix), and in fact one that is in [row echelon form](https://en.wikipedia.org/wiki/Row_echelon_form). Once all of the leading coefficients (the left-most non-zero entry in each row) are 1, and every column containing a leading coefficient has zeros elsewhere, the matrix is said to be in [reduced row echelon form](https://en.wikipedia.org/wiki/Reduced_row_echelon_form). This final form is unique; in other words, it is independent of the sequence of row operations used. For example, in the following sequence of row operations (where multiple elementary operations might be done at each step), the third and fourth matrices are the ones in row echelon form, and the final matrix is the unique reduced row echelon form.

{\displaystyle \left[{\begin{array}{rrr|r}1&3&1&9\\1&1&-1&1\\3&11&5&35\end{array}}\right]\to \left[{\begin{array}{rrr|r}1&3&1&9\\0&-2&-2&-8\\0&2&2&8\end{array}}\right]\to \left[{\begin{array}{rrr|r}1&3&1&9\\0&-2&-2&-8\\0&0&0&0\end{array}}\right]\to \left[{\begin{array}{rrr|r}1&0&-2&-3\\0&1&1&4\\0&0&0&0\end{array}}\right]}**CODE**

function gausselimination()

A = [1 3 3 1; 1 -2 3 7; 4 3 1 5; 2 10 3 5];

B = [9 12 31 8];

% Reducing given matrix into upper triangular matrix

for k = 1:3

for i = k+1:4

f1 = A(i,k) / A(k,k);

for j = k:4

A(i,j) = A(i,j) - f1\*A(k,j);

end

B(i) = B(i) - f1 \* B(k);

end

end

% Solving the given system by backward substitution

disp('The given solution is: ');

for i = 4:-1:1

f2 = 0;

j = i+1;

while(j <= 4)

f2 = f2 + (s(j) \* A(i,j));

j = j+1;

end

s(i) = (B(i) - f2) / A(i,i);

fprintf('%f \n', s(i));

end

**OUTPUT – COMMAND WINDOW**

>> gausselimination()

The given solution is:

-0.454861

1.350694

-1.145833

8.840278

**Practical 6 To implement Gauss Jordan Method.**

**Theory:** The Gauss-Jordan method is also known as Gauss-Jordan elimination method. It was introduced by the mathematicians Carl Friedrich Gauss and Wilhelm Jordan, after their name it is called so. This method is very useful in solving a linear system of equations. It is a technique in which a system of linear equations is resolved by the means of matrices.   
  
This method allows the isolation of the coefficients of a system of linear equations. In Gauss-Jordan method, given matrix can be fetched to row echelon form and made simpler. We can say that the transformation of augmented matrix of the given system into reduced row-echelon form is done by the use of row operations. Sometimes, this method is also used for finding rank of a matrix as well as finding inverse of a square invertible matrix.  
In Gauss-Jordan method, elementary row operations are utilized in order to solve given system of linear equations.

**CODE**

% Function to implement Gauss Jordan Method

Function gaussjordan()

A = [1 3 3 1; 1 -2 3 7; 4 3 1 5; 2 10 3 5];

B = [9 12 31 8];

% Reducing A into upper triangular matrix

for k = 1:3

for i = k+1:3

f1 = A(i,k) / A(k,k);

for j = k:3

A(i,j) = A(i,j) - (f1 \* A(k,j));

end

B(i) = B(i) - (f1 \* B(k));

end

end

% Reducing A into diagonal matrix

for k = 3:-1:2

for i = k-1:-1:1

f2 = A(i,k) / A(k,k);

A(i,k) = A(i,k) - f2 \* A(k,k);

B(i) = B(i) - f2 \* B(k);

end

end

% Solving the given system

fprintf('\n The solution is: \n');

for i = 1:3

s(i) = (B(i) / A(i,i));

fprintf('%f \n', s(i));

end

**OUTPUT – COMMAND WINDOW:**

>> gaussjordan()

The solution is:

7.963636

-0.600000

0.945455

**Practical 7 to implement Newton forward interpolation.**

**Theory: Newton Forward Interpolation**

Newton's forward difference formula is a [finite difference](http://mathworld.wolfram.com/FiniteDifference.html) identity giving an interpolated value between tabulated points {f_p} in terms of the first value f_0and the [powers](http://mathworld.wolfram.com/Power.html) of the [forward difference](http://mathworld.wolfram.com/ForwardDifference.html) Delta. For a in [0,1], the formula states

|  |  |
| --- | --- |
| 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

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

with (a)_n the [falling factorial](http://mathworld.wolfram.com/FallingFactorial.html), the formula looks suspiciously like a finite analogue of a [Taylor series](http://mathworld.wolfram.com/TaylorSeries.html) expansion.

**CODE:**

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 :**

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

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

>> newtonforward(X, F)

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 8 to implement newton divided interpolation.**

**Theory:**

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

where **xi** and **xj** are any two tabular points, is independent of **xi** and **xj**. This ratio is called the first divided difference of **f(x)** relative to **xi** and **xj** and is denoted by **f [xi, xj]**. That is

|  |  |  |
| --- | --- | --- |
| **f [xi, xj] =** | **f(xi) - f(xj)** | **=  f [xj, xi]** |
|  |
| **(xi - xj)** |

Since the ratio is independent of  **xi** and **xj** we can write   **f [x0, x] = f [x0, x1]**

|  |  |  |
| --- | --- | --- |
| **f(x) - f(x0)** |  |  |
|  | = | **f [x0, x1]** |
| **(x - x0)** |  |  |

**f(x)** = **f(x0)** **+** **(x - x0) f [x0, x1]**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| = | **1** | | | **f(x0)** | **x0 - x** | | |  | **f1 - f0** |  | **f0x1 - f1x0** |
|  |  |  | = |  | **x +** |  |
| **x - x0** | **f(x1)** | **x1 - x** |  | **x1 - x0** |  | **x1 - x0** |

So if **f(x)** is approximated with a linear polynomial then the function value at any point **x** can be calculated by using **f(x) @ P1(x) = f(x0) + (x - x1) f [x0, x1]**

where **f [x0, x1]** is the first divided difference of  **f** relative to **x0** and **x1**.

Similarly if **f(x)** is a second degree polynomial then the secant slope defined above is not constant but a linear function of **x.** Hence we have

|  |
| --- |
| **f [x1, x2] - f [x0, x1]** |
|  |
| **x2 - x0** |

**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)

**Output:**

**Practical 9 to implement lagrange interpolation.**

**Theory:** The Lagrange interpolating polynomial is the [polynomial](http://mathworld.wolfram.com/Polynomial.html) P(x) of degree ≤(n-1) that passes through the n points  and is given by –

where

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.  **Code**  function[y] = lagrange(x,n)  ax=input('the given abscisssae');  ay=input('the given ordinates');  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('The required value of y is: %f',y);  **OUTPUT**  Lagrange(301,4)  The given abscissae: [300 304 305 307]  The given ordinates: [2.4771 2.4829 2.4843 2.4871]  The required value of y is: 2.478597  Ans= 2.4786 |

**Practical 10: to implement Trapezoidal method, Simpson 1/3rd method, Simpson 3/8 method interpolation.**

**Theory:**

1. 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 f(x) as a trapezoid and calculating its area. It follows that

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

Simpsons 1/3rd Rule :

\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].

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

Simpsons 3/8th Rule:

 \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 n is a multiple of three.

**Code**

function [ result ] = numint( 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

**Output**

>> f = exp(-x^2)

f =

exp(-x^2)

>> **numint(f, 0, 1, 20)**

The answer for integration is:

ans =

0.7467

>> **numint(f, 0, 0.6, 7)**

ans =

0.4743 0.0608

**>> numint(f, 0, 0.6, 7)**

ans =

0.5194 0.015

**Practical 11: to implement Runga Kutta method to solve differential equation**

**THEORY:**

A method of numerically integrating ordinary differential equations by using a trial step at the midpoint of an interval to cancel out lower-order error terms and the fourth-order Range-Kutta method formula is

**Code:**

function result = rangekutta(f,xo,yo,h,xval)

syms x y;

n = int((xval-xo)/h);

xn = xo;

yn = yo;

for i = 1:n

k1 = h\*subs(f,{x,y},{xn,yn});

k2 = h\*subs(f,{x,y},{xo+h/2,yo+k1/2});

k3 = h\*subs(f,{x,y},{xo+h/2,yo+k2/2});

k4 = h\*subs(f,{x,y},{xo+h,yo+k3});

yn = yn + (k1 + 2\*k2 + 2\*k3 + k4)/6;

xn = xn + h;

end

result = yn;

end

**Output**

rangekutta(x+y, 0, 1, 0.1, 0.3)

ans =

1.3427