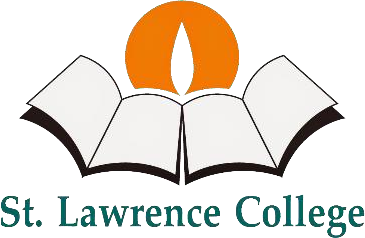
St. Lawrence College

Chabahil, Kathmandu



**LAB REPORT**

**“Numerical Method – CSC212” B.Sc.CSIT 3th Semester**

**Submitted by:**

**-----------------------**

### Roll No.: 00

**Submitted to:**

**Dipendra Basnet**

### Lecturer, B.Sc.CSIT

**Table of Contents**

[Bisection Method 3](#_bookmark0)

[Regular-Falsi Method 5](#_bookmark1)

[Secant Method 7](#_bookmark2)

[Newton’s Raphson Method 10](#_bookmark3)

[Fixed Point Iteration Method 13](#_bookmark4)

[Gauss Elimination Method 15](#_bookmark5)

[Gauss-Jordan Method 18](#_bookmark6)

[Matrix Inversion using Gauss-Jordan Method 20](#_bookmark7)

[Doo-Little LU Decomposition 22](#_bookmark8)

[Power Method 25](#_bookmark9)

[Least Square Method (Curve Fitting) 27](#_bookmark10)

* [Linear form (y=ax+b) 27](#_bookmark11)
* [Exponential Form(y=aebx) 29](#_bookmark12)

[Lagrange’s Interpolation Polynomial 31](#_bookmark13)

[Trapezoidal Rule 33](#_bookmark14)

[Simpson’s 1/3 Rule 34](#_bookmark15)

[Simpson’s 3/8 rule 35](#_bookmark16)

[Runge-Kutta Method 36](#_bookmark17)

# Bisection Method

One of the first numerical methods developed to find the root of a non-linear equation f(x)=0 was the bisection method also called half-interval method.

The method is based on the following theorem: Let f(x)= 0 eqn(i)

And also, a and b be the initial guesses; Such that f(a)<0 and f(b)>0 i.e., f(b)<0. If c be the midpoint of a and b,

𝑐 =

𝑎 + 𝑏

2

Then we have following conditions,

1. If f(c)=0, then c is exact root.
2. If f(c)<0, then root lies between c and b.
3. If f(c)>0, then root lies between a and c.

##### //C-program for Bisection Method.

#include <stdio.h> #include <math.h> float f(float x) {

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

}

int main() {

float xl, xu, E, xm, fxl, fxu, fxm, Era; int step = 1;

printf("Enter initial bracketing (xl and xu) and E: "); scanf("%f %f %f", &xl, &xu, &E);

if (f(xl) \* f(xu) > 0) {

printf("Invalid initial bracket. f(xl) and f(xu) must have opposite signs.\n"); return -1;

}

printf("\nStep\t\t xl\t\t xu\t\t xm\t\t f(xm)\n");

do {

fxl = f(xl); fxu = f(xu);

xm = (xl + xu) / 2; fxm = f(xm);

printf("%d\t\t %f\t %f\t %f\t %f\n", step, xl, xu, xm, fxm); if ((fxl \* fxm) < 0) {

xu = xm;

} else {

xl = xm;

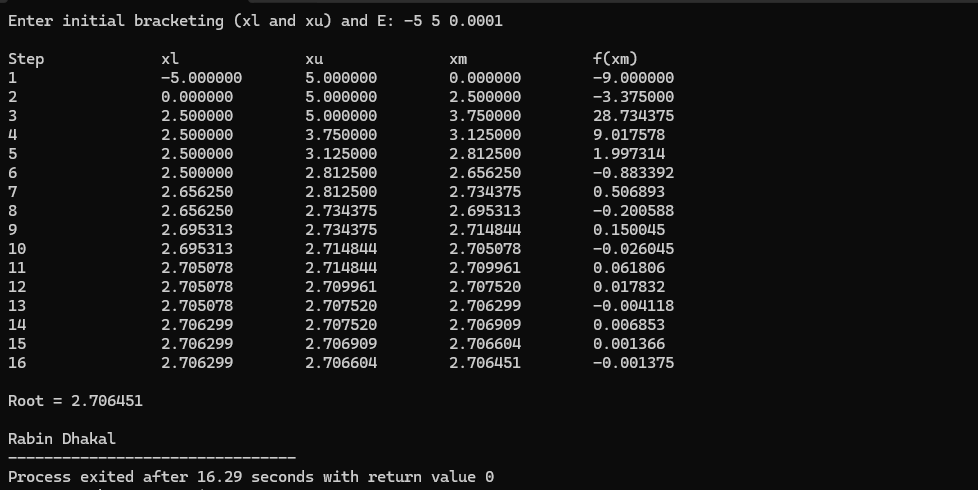
}

Era = fabs((xu - xl) / xm); step++;

} while (Era > E);

printf("\nRoot = %f\n", xm); printf("\n");

***Output: -***

******

# Regular-Falsi Method

Eqn of straight line joining between two points [a, f(a)] and [b, f(b)]

y-f(a)=𝑓(𝑏)−𝑓(𝑎) (𝑥 − 𝑎)

𝑏−𝑎

since, the line intersects the x-axis at c, when x=c, y=0

0-f(a)=𝑓(𝑏)−𝑓(𝑎) (𝑐 − 𝑎)

𝑏−𝑎

−𝑓(𝑎)

𝑓(𝑏) − 𝑓(𝑎) (𝑏 − 𝑎) = 𝑐 − 𝑎

𝑓(𝑎)(𝑏 − 𝑎)

𝑐 = 𝑎 − 𝑓(𝑏) − 𝑓(𝑎)

∴ 𝑐 =

𝑎𝑓(𝑏) − 𝑏𝑓(𝑎)

𝑓(𝑏) − 𝑓(𝑎)

##### //C-program for Regular-Falsi Method

#include<stdio.h> #include<math.h> float f (float x)

{

return 3\*x-cos(x) - 1;

}

void regula (float \*x, float x0, float x1, float fx0, float fx1, int \*itr)

{

\*x = x0 - ((x1 - x0) / (fx1 - fx0)) \*fx0;

++(\*itr);

printf ("Iteration no. %3d X = %7.5f \n", \*itr, \*x);

}

int main ()

{

int itr = 0, maxmitr;

float x0, x1, x2, x3, allerr;

printf ("\nEnter the values of x0, x1, allowed error and maximum iterations:\n"); scanf ("%f %f %f %d", &x0, &x1, &allerr, &maxmitr);

regula (&x2, x0, x1, f(x0), f(x1), &itr); do

{

if (f(x0) \*f(x2) < 0) x1=x2;

else

x0=x2;

regula (&x3, x0, x1, f(x0), f(x1), &itr); if (fabs(x3-x2) < allerr)

{

printf ("After %d iterations, root = %6.4f\n", itr, x3); printf("\n");

x2=x3;

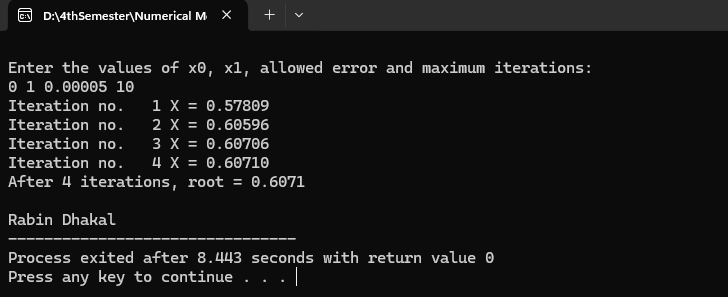
}

while (itr<maxmitr);

printf ("Solution does not converge or iterations not sufficient:\n"); return 1;

}

***Output: -***

******

# Secant Method

Let f(x)=0 eqn(i)

And x0 and x1 be the initial gueses

Then line joining between two points [x0,f(x0)] and [x1,f(x1)]

𝑦 − 𝑓(𝑥) =

𝑓(𝑥1) − 𝑓(𝑥0)

𝑥1 − 𝑥0

(𝑥 − 𝑥1)

𝑦 = 𝑓(𝑥1)−𝑓(𝑥0) (𝑥 − 𝑥1) + 𝑓(𝑥) eqn(ii)

𝑥1−𝑥0

The approximate root of the function that is value such that y =0. Then eqn(ii) becomes,

0 = 𝑓(𝑥1) +

𝑓(𝑥1) − 𝑓(𝑥0)

𝑥1 − 𝑥0

−𝑓(𝑥1)

𝑥 − 𝑥1

𝑓(𝑥1) − 𝑓(𝑥0)

=

𝑥1 − 𝑥0

∴ 𝑥 = 𝑥1 −

𝑓(𝑥1)(𝑥1 − 𝑥0)

𝑓(𝑥1) − 𝑓(𝑥0)

Now x= x2,

𝑥2 = 𝑥1 −

𝑓(𝑥1)(𝑥1 − 𝑥0)

𝑓(𝑥1) − 𝑓(𝑥0)

𝑥3 = 𝑥2 −

𝑓(𝑥2)(𝑥2 − 𝑥1)

𝑓(𝑥2) − 𝑓(𝑥1)

∴ 𝑥𝑛+1 = 𝑥𝑛 −

𝑓(𝑥𝑛)(𝑥𝑛 − 𝑥𝑛−1)

𝑓(𝑥𝑛) − 𝑓(𝑥𝑛−1)

#### //C-program for Secant Method

#include<stdio.h> #include<conio.h> #include<math.h> #include<stdlib.h>

#define f(x) cos(x)+2\*sin(x)+x\*x int main()

{

float x0, x1, x2, f0, f1, f2, e; int step = 1, N;

printf("\nEnter initial guesses:\n"); scanf("%f%f", &x0, &x1); printf("Enter tolerable error:\n"); scanf("%f", &e);

printf("Enter maximum iteration:\n"); scanf("%d", &N); printf("\nStep\t\tx0\t\tx1\t\tx2\t\tf(x2)\n"); do

{

f0 = f(x0);

f1 = f(x1); if(f0 == f1)

{

printf("Mathematical Error."); exit(0);

}

x2 = x1 - (x1 - x0) \* f1/(f1-f0); f2 = f(x2);

printf("%d\t\t%f\t%f\t%f\t%f\n",step,x0,x1,x2, f2);

x0 = x1;

f0 = f1; x1 = x2;

f1 = f2;

step = step + 1; if(step > N)

{

printf("Not Convergent.");

exit(0);

}

}

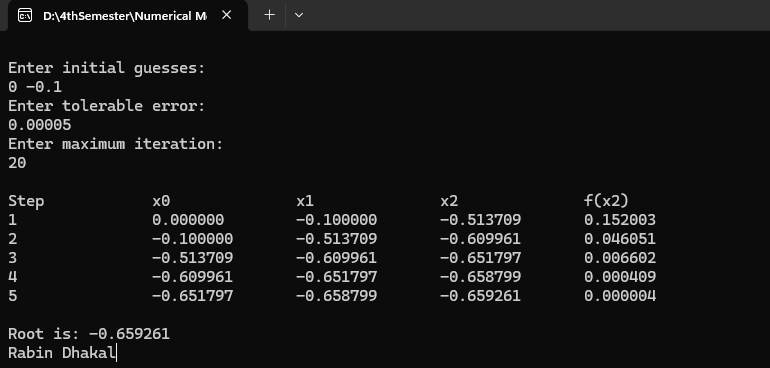
while(fabs(f2)>e);

printf("\nRoot is: %f", x2); printf("\n");

getch();

}

***Output:-***

******

# Newton’s Raphson Method

Let f(x)= 0 eqn(i)

Let x0 be the initial guesses to the root, x be the root corresponding exact root such that, x= x0+h (h= very small quantity) (ii)

f(x)= 0

f(x0+h) =0

𝑓(𝑥0 + ℎ) = 𝑓(𝑥0) +

ℎ𝑓′(𝑥0) 1!

ℎ2𝑓′′(𝑥0)

+

2!

ℎ3𝑓′′′(𝑥0)

+ + ⋯

3!

𝑓(𝑥0) +

ℎ𝑓′(𝑥0) 1!

ℎ2𝑓′′(𝑥0)

+ +

2!

ℎ3𝑓′′′(𝑥0) 3!

+ ⋯ = 0

Neglecting terms with h2 and higher power of h.

𝑓(𝑥0) +

ℎ𝑓′(𝑥0)

= 0

1!

𝑓(𝑥0) + ℎ𝑓′(𝑥0) = 0

𝑓(𝑥0)

𝑓′(𝑥0)

= −ℎ

−𝑓(𝑥0)

ℎ =

𝑓′(𝑥0)

∴ 𝑥 = 𝑥0 −

𝑓(𝑥0)

𝑓′(𝑥0)

##### //C-program for Newton Raphson’s Method

#include <stdio.h> #include <math.h> #include <stdlib.h>

#define f(x) (cos(x) + 2 \* sin(x) + x \* x) #define df(x) (-sin(x) + 2 \* cos(x) + 2 \* x) int main() {

float x0, x1, f0, df0, e; int step = 1, N;

printf("\nEnter initial guess:\n"); scanf("%f", &x0);

printf("Enter tolerable error:\n"); scanf("%f", &e);

printf("Enter maximum iterations:\n"); scanf("%d", &N); printf("\nStep\t\tx0\t\tf(x0)\t\tdf(x0)\t\tx1\n"); do {

f0 = f(x0); df0 = df(x0);

if (fabs(df0) < 1e-10) {

printf("Mathematical Error: Derivative is zero.\n"); exit(0);

}

x1 = x0 - f0 / df0;

printf("%d\t\t%f\t%f\t%f\t%f\n", step, x0, f0, df0, x1); if (fabs(x1 - x0) < e) {

printf("\nRoot is: %f\n", x1); return 0;

}

x0 = x1; step++;

if (step > N) {

printf("Not Convergent.\n"); exit(0);

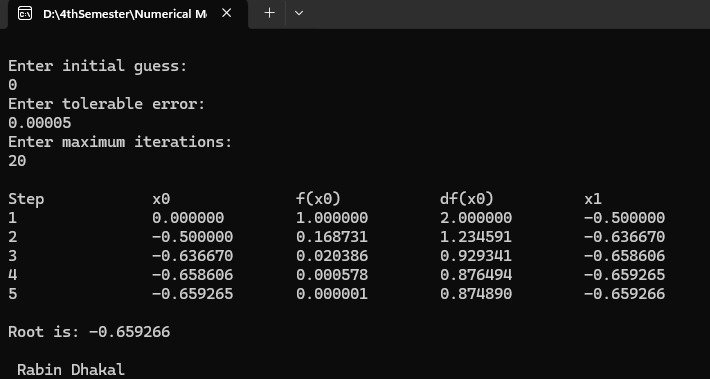
}

} while (1);

return 0;

}

***Output: -***

******

# Fixed Point Iteration Method

Let f(x)=0 eqn(i)

Equation I can be manipulated such that x is on the left-hand side of the eqn as, x=g(x) (ii)

let x0 be the initial guesses then x1= g(x0)

x2=g(x1) x3=g(x2)

xi+1= g(xi), where i= 1,2,3, 4, …

##### //C-program for Fixed Point Iteration Method

#include <stdio.h> #include <math.h>

#define G(x) ((a3 \* (x) \* (x) \* (x) + a2 \* (x) \* (x) + a0) / (-a1)) float a0, a1, a2, a3;

int main() {

float x0, x1, E, Er;

int max\_iterations = 100, step = 1;

printf("Enter coefficients a3, a2, a1, and a0:\n"); scanf("%f %f %f %f", &a3, &a2, &a1, &a0);

if (a1 == 0) {

printf("Error: Division by zero in the function G(x).\n"); return 1;

}

printf("Enter initial guess and tolerance E:\n"); scanf("%f %f", &x0, &E); printf("\nStep\t\tx0\t\tx1\t\tError\n");

while (step <= max\_iterations) { x1 = G(x0);

Er = fabs((x1 - x0) / x1); printf("%d\t\t%f\t%f\t%f\n", step, x0, x1, Er); if (Er < E) {

printf("\nRoot = %f\n", x1); break;

}

x0 = x1; step++;

}

if (step > max\_iterations) {

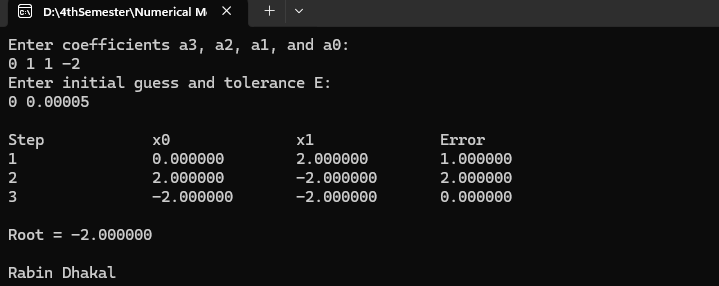
printf("The method did not converge within %d iterations.\n", max\_iterations);

}

return 0;

}

***Output: -***

******

# Gauss Elimination Method

##### //C-program for Gauss Elimination Method

#include<stdio.h> #include<conio.h> #include<math.h> int main ()

{

int n, i, k, j, p, q, row;

float pivot, temp, largest, term, sum = 0, a [10][10], b [10], x [10]; printf ("Enter Dimension of System of equations: \n");

scanf ("%d", &n);

printf ("Enter coefficients row-wise:\n"); for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

scanf ("%f", &a[i][j]);

}

}

printf ("Enter RHS vector: \n"); for (i = 0; i < n; i++)

{

scanf ("%f", &b[i]);

}

for (k = 0; k < n - 1; k++) ***// Forward Elimination***

{

largest = fabs(a[k][k]); row = k;

for (p = k + 1; p < n; p++)

{

if(fabs(a[p][k]) > largest)

{

largest = fabs(a[p][k]); row = p;

}

}

for (q = 0; q < n; q++)

{

temp = a[k][q]; a[k][q] = a[row][q]; a[row][q] = temp;

}

temp = b[k]; b[k] = b[row]; b[row] = temp; pivot = a[k][k];

for (i = k + 1; i < n; i++)

{

term = a[i][k] / pivot; for (j = 0; j < n; j++)

{

a[i][j] = a[i][j] - a[k][j] \* term;

}

b[i] = b[i] - b[k] \* term;

}

}

x [n - 1] = b [n - 1] / a [n - 1] [n - 1];

for (i = n - 2; i >= 0; i--) ***// Back substitution***

{

sum = 0;

for (j = i + 1; j < n; j++)

{

sum = sum + a[i][j] \* x[j];

}

x[i] = (b[i] - sum) / a[i][i];

}

printf("Solution:\n");

for (i = 0; i < n; i++) ***// Display Solution Vector***

{

printf ("x%d = %f\n", i + 1, x[i]);

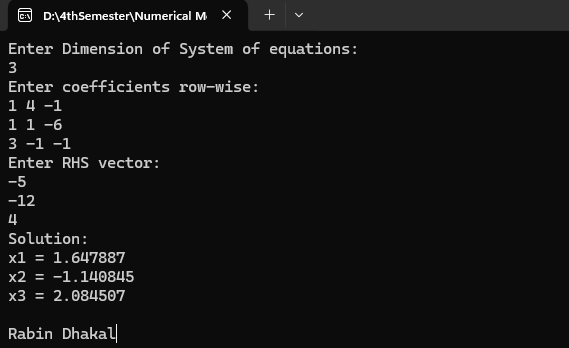
}

getch ();

return 0;

}

***Output: -***

******

# Gauss-Jordan Method

##### //C-program for Gauss-Jordan Method

#include <stdio.h> #include <math.h> int main ()

{

int n, i, k, j, p, q;

float pivot, term, a [10][10];

printf ("Enter Dimension of System of equations\n"); scanf ("%d", &n);

printf ("Enter coefficients Augmented Matrix\n"); for (i = 0; i < n; i++) {

for (j = 0; j < n + 1; j++) { scanf ("%f", &a[i][j]);

}

}

for (k = 0; k < n; k++) {***// Elimination***

pivot = a[k][k];

for (p = 0; p < n + 1; p++) {

a[k][p] = a[k][p] / pivot; ***// Normalization***

}

for (i = 0; i < n; i++) { term = a[i][k];

if (k != i) {

for (j = 0; j < n + 1; j++) {

a[i][j] = a[i][j] - a[k][j] \* term;

}

}

}

}

printf("Solution:\n");

for (i = 0; i < n; i++) {***// Display Solution Vector***

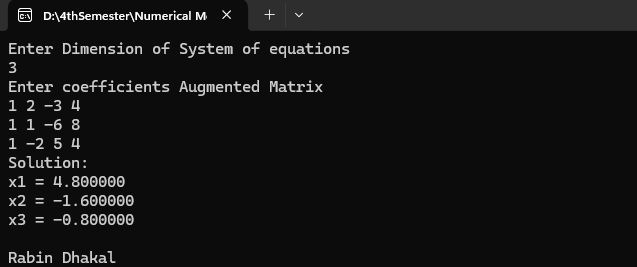
printf ("x%d = %f\n", i + 1, a[i][n]);

}

return 0;

}

***Output: -***

******

# Matrix Inversion using Gauss-Jordan Method

##### //C-program for Matrix Inversion using Gauss-Jordan Method

#include <stdio.h> #include <stdlib.h> #include <conio.h> #include <math.h> int main ()

{

int n, i, k, j, p, q;

float pivot, term, a [10][10];

printf ("Enter Dimension of System of equations: \n"); scanf ("%d", &n);

printf ("Enter coefficients Matrix:\n"); for (i = 0; i < n; i++) {

for (j = 0; j < n; j++) { scanf ("%f", &a[i][j]);

}

}

##### // Augment the matrix

for (i = 0; i < n; i++) {

for (j = n; j < 2 \* n; j++) { if (j - n == i)

a[i][j] = 1; else

a[i][j] = 0;

}

}

##### // Inverse Computation

for (k = 0; k < n; k++) { pivot = a[k][k];

for (p = 0; p < 2 \* n; p++) { a[k][p] = a[k][p] / pivot;

}

for (i = 0; i < n; i++) {

term = a[i][k];

if (k != i) {

for (j = 0; j < 2 \* n; j++) {

a[i][j] = a[i][j] - a[k][j] \* term;

}

}

}

}

printf ("Matrix Inverse is: \n"); for (i = 0; i < n; i++) {

for (j = n; j < 2 \* n; j++) { printf ("%f\t", a[i][j]);

}

printf("\n");

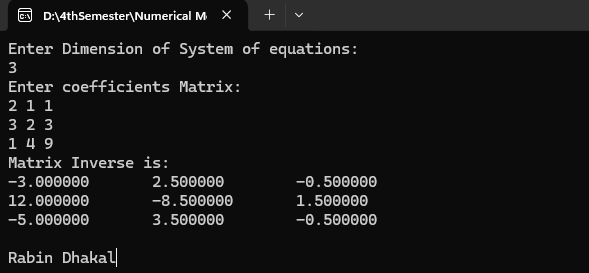
}

getch ();

return 0;

}

***Output: -***

******

# Doo-Little LU Decomposition

##### //C-program for solving system of linear equations using Doo-Little LU Decomposition

#include<stdio.h> #include<conio.h> #include<math.h> int main ()

{

int n, i, j, k;

float sum = 0, a [10][10], b [10], x [10], z [10], u [10][10], l [10][10];

printf ("Enter Dimension of System of equations\n"); scanf ("%d", &n);

printf ("Enter coefficients Matrix\n"); for (i = 0; i < n; i++) {

for (j = 0; j < n; j++) { scanf ("%f", &a[i][j]);

}

}

printf ("Enter RHS vector\n"); for (i = 0; i < n; i++) {

scanf ("%f", &b[i]);

}

##### // Compute Elements of L and U matrix

for (j = 0; j < n; j++) { u[0][j] = a[0][j];

}

for (i = 0; i < n; i++) { l[i][i] = 1;

}

for (i = 1; i < n; i++) { l[i][0] = a[i][0] / u [0][0];

}

for (j = 1; j < n; j++) { for (i =1; i <=j; i++) {

sum = 0;

for (k = 0; k <= i - 1; k++) {

sum = sum + (l[i][k] \* u[k][j]);

}

u[i][j] = a[i][j] - sum;

}

for (i = j + 1; i < n; i++) { sum = 0;

for (k = 0; k <= j - 1; k++) {

sum = sum + (l[i][k] \* u[k][j]);

}

l[i][j] = (a[i][j] - sum) / u[j][j];

}

}

##### // Solve for Z using Forward Substitution

z [0] = b [0];

for (i = 1; i < n; i++) { sum = 0;

for (j = 0; j < i; j++) {

sum = sum + (l[i][j] \* z[j]);

}

z[i] = b[i] - sum;

}

##### // Solve for X using Backward Substitution

x [n - 1] = z [n - 1] / u [n - 1] [n - 1]; for (i = n - 2; i >= 0; i--) {

sum = 0;

for (j = i + 1; j < n; j++) {

sum = sum + (u[i][j] \* x[j]);

}

x[i] = (z[i] - sum) / u[i][i];

}

printf("Solution:\n"); for (i = 0; i < n; i++) {

printf ("x%d = %f\n", i + 1, x[i]);

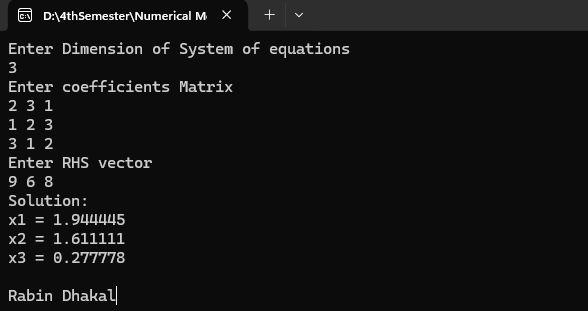
}

getch ();

return 0;

}

***Output: -***

******

# Power Method

##### //C-program for calculating eigenvalue and eigenvector of a matrix by using power method

#include<stdio.h> #include<conio.h> #include<math.h> int main ()

{

int n, i, j;

float el, k, E [10], a [10][10], x [10], nx [10], y [10];

printf ("Enter Dimension of System of equations\n"); scanf ("%d", &n);

printf ("Enter coefficients row-wise\n"); for (i = 0; i < n; i++) {

for (j = 0; j < n; j++) { scanf ("%f", &a[i][j]);

} }

printf ("Enter guess vector\n"); for (i = 0; i < n; i++) {

scanf ("%f", &x[i]);

}

printf ("Enter Accuracy Limit\n"); scanf ("%f", &el);

while (1) {

for (i = 0; i < n; i++) { y[i] = 0;

for (j = 0; j < n; j++) { y[i] += a[i][j] \* x[j];

}}

k = y [0];

for (i = 1; i < n; i++) { if (k < y[i]) {

k = y[i];

}

}

for (i = 0; i < n; i++) {

nx[i] = 1 / k \* y[i];

E[i] = fabs((nx[i] - x[i]) / nx[i]);

}

int converged = 1;

for (i = 0; i < n; i++) { if(E[i] > el) {

converged = 0; break;

}}

if(converged) {

printf ("Largest Eigenvalue is: %f\n", k); printf ("Eigenvector is:\n");

for (i = 0; i < n; i++) { printf ("%f\t", nx[i]);

}

break;

}

else {

for (i = 0; i < n; i++) { x[i] = nx[i];

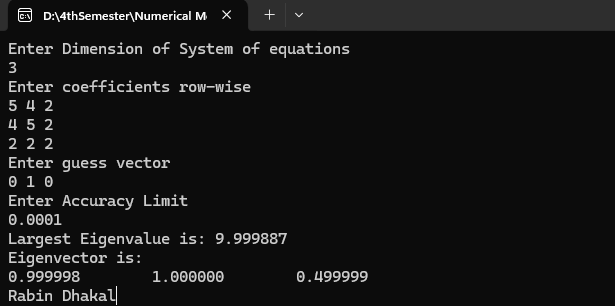
}}}

getch ();

return 0;

}

***Output: -***

******

# Least Square Method (Curve Fitting)

## Linear form (y=ax+b)

Let y = ax+b be the equation of straight line,

∴ 𝑦 = 𝑎𝑥 + 𝑏 eqn (i)

Normal equation is,

∑y=a∑x+nb (ii)

And,

∑xy=a∑x2+b∑x (iii)

##### //C program for Linear Regression

#include<stdio.h> #include<conio.h> int main()

{

int n,i,j,k;

float a=0,b=0,x[10],y[10],sx=0,sy=0,sxy=0,sx2=0; printf("Enter the number of points \n"); scanf("%d",&n);

printf("Enter the value of x and y\n"); for(i=0;i<n;i++)

{

scanf("%f%f", &x[i], &y[i]);

}

for(i=0;i<n;i++)

{

sx=sx+x[i]; sy=sy+y[i]; sxy=sxy+x[i]\*y[i]; sx2=sx2+x[i]\*x[i];

}

a=((n\*sxy) - (sx\*sy))/((n\*sx2) - (sx\*sx));

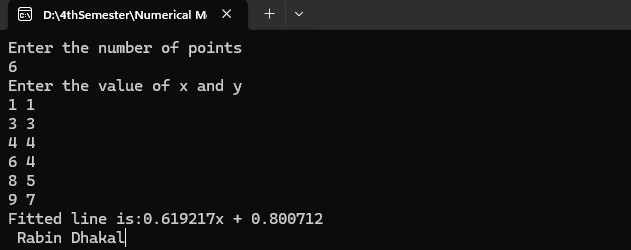
b=(sy/n) - (a\*sx/n);

printf("Fitted line is:%fx + %f",a,b);

getch();

}

***Output:-***

******

## Exponential Form(y=aebx)

Let the curve be, y=aebx eqn(i)

taking log on both sides,

logy = loga +logbx. logee [logee=1] or, logy= loga+bx eqn(ii)

let,

logy = Y loga= A

eqn(i) becomes

Y= A+bx eqn(iii)

Normal equations are:

∑Y= nA+b∑x (iv)

∑xY= A∑x+ b∑x2 (v)

##### //C-program for exponential regression

#include<stdio.h> #include<conio.h> #include<math.h> int main ()

{

int n, i, j, k;

float a=0, b=0, r, x [10], y [10], sx=0, slgy=0, sxy=0, sx2=0; printf ("Enter the number of points");

scanf ("%d", &n);

printf ("Enter the value of x and fx"); for (i=0; i<n; i++)

{

scanf ("%f%f", &x[i], &y[i]);

}

for (i=0; i<n; i++)

{

sx=sx+x[i]; slgy=slgy +log(y[i]);

sxy=sxy +x[i]\*log(y[i]); sx2=sx2+x[i]\*x[i];

}

b=((n\*sxy) - (sx\*slgy)) / ((n\*sx2) - (sx\*sx)); r= (slgy /n) - (b\*sx/n);

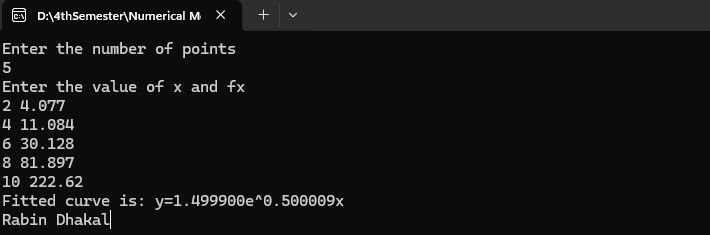
a=exp(r);

printf ("Fitted curve is: y=%fe^%fx", a, b);

getch ();

}

***Output: -***

******

# Lagrange’s Interpolation Polynomial

The formula for Lagrange’s Interpolation Polynomial is gives as,

∴ 𝑓(𝑥) = (𝑥−𝑥1)(𝑥−𝑥2)……(𝑥−𝑥𝑛) × 𝑦 + (𝑥−𝑥0)(𝑥−𝑥2)……(𝑥−𝑥𝑛) × 𝑦 +

(𝑥0−𝑥1)(𝑥0−𝑥2)……(𝑥0−𝑥𝑛) 0 (𝑥1−𝑥0)(𝑥1−𝑥2)……(𝑥1−𝑥𝑛) 1

(𝑥−𝑥0)(𝑥−𝑥1)(𝑥−𝑥3)……(𝑥−𝑥𝑛) × 𝑦 + ⋯ (𝑥−𝑥0)(𝑥−𝑥1)(𝑥−𝑥2)……(𝑥−𝑥𝑛−1) × 𝑦

(𝑥2−𝑥0)(𝑥2−𝑥1)(𝑥2−𝑥3)……(𝑥2−𝑥𝑛) 2 (𝑥𝑛−𝑥0)(𝑥𝑛−𝑥1)……(𝑥𝑛−𝑥𝑛−1) 𝑛

##### //C Program for Lagrange Interpolation

#include<stdio.h> #include<conio.h> int main ()

{

int n, i, j, k;

float x, l, v=0, ax [10], fx [10], L [10]; printf ("Enter the number of points \n"); scanf ("%d", &n);

printf ("Enter the value of x\n"); scanf ("%f", &x);

for (i=0; i<n; i++)

{

printf ("Enter the value of x and fx at i=%d \n", i); scanf ("%f%f", &ax[i], &fx[i]);

}

for (i=0; i<n; i++)

{

l=1.0;

for (j=0; j<n; j++)

{

} L[i]=l;

}

if (j != i)

l=l\*((x-ax[j])/(ax[i]-ax[j]));

for (i=0; i<n; i++)

{

v=v +fx[i]\*L[i];

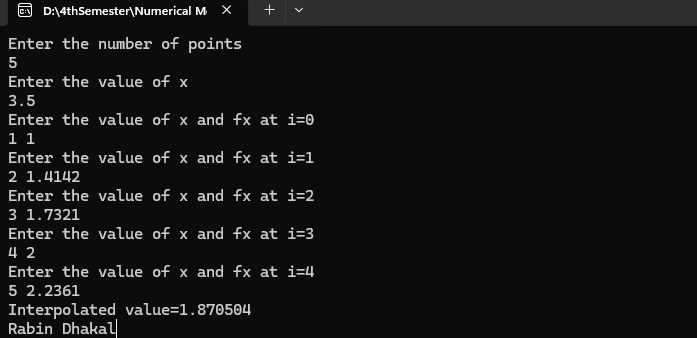
}

printf ("Interpolated value=%f", v);

getch (); return 0;

}

***Output: -***

******

# Trapezoidal Rule

The formula for trapezoidal rules is given as,

𝑏 ℎ

∴ ∫ 𝑦𝑑𝑥 =  [(𝑦0 + 𝑦𝑛) + 2(𝑦1 + 𝑦2 + 𝑦3 + ⋯ 𝑦𝑛−1)

2

𝑎

##### //C-program for Trapezoidal Rule

#include<stdio.h> #include<conio.h> #include<math.h> #define f(x) pow(x,3) +2 int main ()

{

float h, x0, x1, fx0, fx1, v;

printf ("Enter Lower & Upper Limit: \n"); scanf ("%f %f", &x0, &x1);

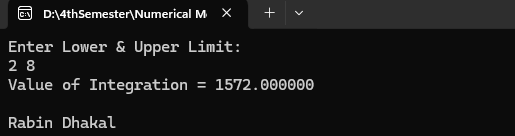
h = (x1 - x0); fx0=f(x0); fx1=f(x1); v=h/2\*(fx0+fx1);

printf ("Value of Integration = %f\n", v);

getch (); return 0;

}

***Output: -***

******

# Simpson’s 1/3 Rule

The formula for Simpson’s 1/3 is given as,

𝑏 ℎ

∴ ∫ 𝑦𝑑𝑥 =  [(𝑦0 + 𝑦𝑛) + 2(𝑦2 + 𝑦4 + 𝑦6 + ⋯ 𝑦𝑛−2) + 4(𝑦1 + 𝑦3 + +𝑦5 + ⋯ 𝑦𝑛−1)

3

𝑎

##### //C-program for Simpson’s 1/3 rule

#include<stdio.h> #include<conio.h> #include<math.h>

#define f(x) sqrt(1-pow(x,2)) int main ()

{

float h, x0, x1, x2, fx0, fx1, fx2, v; int n = 2;

printf ("Enter Lower & Upper Limit: \n"); scanf ("%f %f", &x0, &x2);

h = (x2 - x0) / n; x1 = x0 + h;

fx0 = f(x0); fx1 = f(x1); fx2 = f(x2);

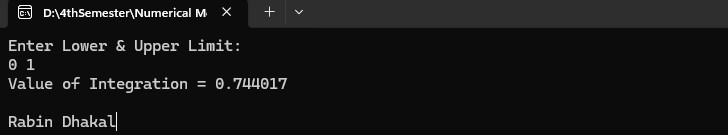
v = h / 3 \* (fx0 + 4 \* fx1 + fx2);

printf ("Value of Integration = %f\n", v);

getch (); return 0;

}

***Output: -***

******

# Simpson’s 3/8 rule

The formula for Simpson’s 3/8 rule is given as,

𝑏

∴ ∫ 𝑦𝑑𝑥 =

𝑎

3ℎ

8 [(𝑦0 + 𝑦𝑛) + 2(𝑦3 + 𝑦6 + 𝑦9 + ⋯ 𝑦𝑛−3) + 3(𝑦1 + 𝑦2 + 𝑦4 + ⋯ 𝑦𝑛−1)

##### //C-program for Simpson’s 3/8 rule

#include<stdio.h> #include<conio.h> #include<math.h>

#define f(x) ((x)\*(x)\*(x) + 1) int main () {

float h, x0, x1, x2, x3, fx0, fx1, fx2, fx3, v; int n = 3;

printf ("Enter Lower & Upper Limit: \n"); scanf ("%f%f", &x0, &x3);

h = (x3 - x0) / n; x1 = x0 + h;

x2 = x0 + 2 \* h; x3 = x0 + 3 \* h; fx0 = f(x0);

fx1 = f(x1); fx2 = f(x2); fx3 = f(x3);

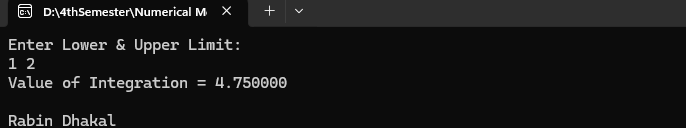
v = 3.0 / 8.0 \* h \* (fx0 + 3 \* fx1 + 3 \* fx2 + fx3); printf ("Value of Integration = %f\n", v);

getch ();

return 0;

}

***Output:-***

******

# Runge-Kutta Method

The formula for Runge-Kutta Method is given as, Given, the initial value problem

𝑑𝑥

= 𝑓(𝑥, 𝑦) , 𝑦(𝑥0) = 𝑦0

𝑑𝑦

For a fixed constant value of h; y(xn+h) can be approximated by

1

𝑦(𝑥0 + ℎ) = 𝑦𝑛+1 = 𝑦𝑛 + 6 ℎ(𝑚1 + 2𝑚2 + 2𝑚3 + 𝑚4)

Were,

𝑚1 = 𝑓(𝑥𝑖, 𝑦𝑖)

ℎ ℎ

𝑚2 = 𝑓(𝑥𝑖 + 2 , 𝑦𝑖 + 2 𝑚1)

ℎ ℎ

𝑚3 = 𝑓(𝑥𝑖 + 2 , 𝑦𝑖 + 2 𝑚2)

ℎ ℎ

𝑚4 = 𝑓(𝑥𝑖 + 2 , 𝑦𝑖 + 2 𝑚3)

##### //C-program for solving ordinary differential equations using Runge-Kutta Method

#include <stdio.h>

double f(double x, double y) { return x + y;

}

##### // Runge-Kutta 2nd Order Method

void rungeKutta2ndOrder(double x0, double y0, double h, double xn) { printf("\n2nd Order Runge-Kutta Method:\n");

printf("x\t\ty\n");

double x = x0, y = y0; while (x < xn) {

printf("%.4lf\t%.4lf\n", x, y);

double k1 = h \* f(x, y);

double k2 = h \* f(x + h, y + k1);

y = y + 0.5 \* (k1 + k2); x += h;

}

printf("%.4lf\t%.4lf\n", x, y);

}

##### // Runge-Kutta 4th Order Method

void rungeKutta4thOrder(double x0, double y0, double h, double xn) { printf("\n4th Order Runge-Kutta Method:\n");

printf("x\t\ty\n");

double x = x0, y = y0; while (x < xn) {

printf("%.4lf\t%.4lf\n", x, y); double k1 = h \* f(x, y);

double k2 = h \* f(x + 0.5 \* h, y + 0.5 \* k1); double k3 = h \* f(x + 0.5 \* h, y + 0.5 \* k2); double k4 = h \* f(x + h, y + k3);

y = y + (k1 + 2 \* k2 + 2 \* k3 + k4) / 6; x += h;

}

printf("%.4lf\t%.4lf\n", x, y);

}

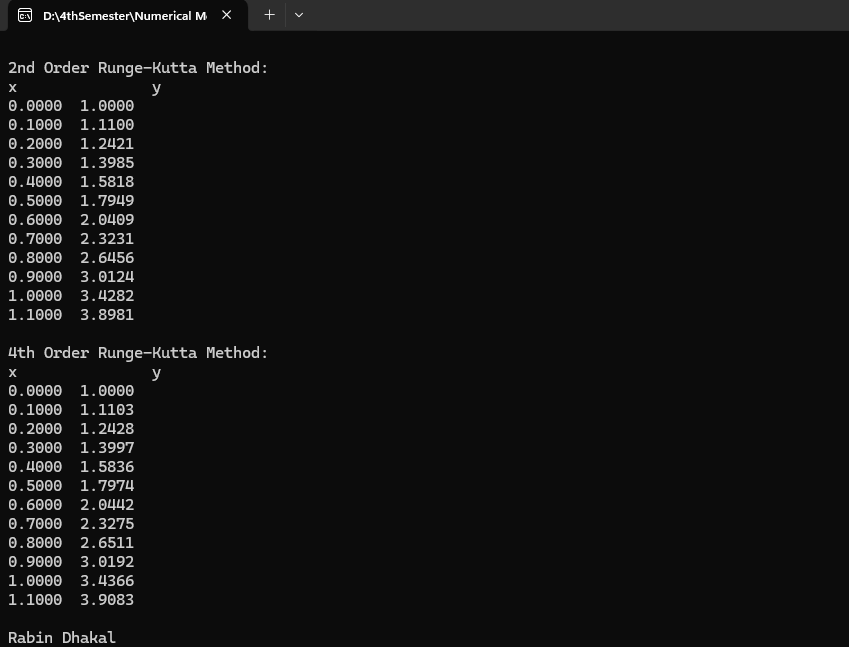
int main() {

double x0 = 0, y0 = 1; double h = 0.1; double xn = 1;

rungeKutta2ndOrder(x0, y0, h, xn); rungeKutta4thOrder(x0, y0, h, xn); return 0;

}

***Output: -***

******