MINOR: COMPUTATIONAL MATHEMATICS

Transcendental Equations

Q.1) Finding Real Root using Bisection Method in C Programming.

AIM: To write a C program to find the real root of an equation using the Bisection Method.

Formula:

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

Programming Formula:

$$xr = (a + b) / 2;$$

Program:

```c
#include <stdio.h>

#include <math.h>

#include <conio.h>

float f(float x) {

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

}

int main() {

    float a = 2, b = 3, xr, fx, fb, fxr;

    int intr = 0;

    float eps = 0.001;

    float prev1 = 0.0, prev2 = 0.0;

    int same_count = 0;

    clrscr();

    fx = f(a);

    fb = f(b);

    if (fx * fb > 0) {

        printf("Root not found in the given interval.\n");

        return 0;
```

```
        }
    printf("\nItr\t a\t  b\t  xr\t f(a)\t f(b)\t f(xr)\n");
    printf("-----------------------------------------------------\n");
    do {
        intr++;
        xr = (a + b) / 2;
        fxr = f(xr);
        fx = f(a);
        fb = f(b);
        printf("%2d\t%.4f\t%.4f\t%.4f\t%.4f\t%.4f\t%.4f\n",
            intr, a, b, xr, fx, fb, fxr);
        if (fx * fxr < 0)
            b = xr;
        else
            a = xr;
        float xr3 = floor(xr * 1000 + 0.5) / 1000;
        if (fabs(xr3 - prev1) < 0.0001 && fabs(prev1 - prev2) < 0.0001)
            same_count++;
        else
            same_count = 0;
        prev2 = prev1;
        prev1 = xr3;
        if (same_count >= 1)
            break;
    } while (fabs(fxr) > eps);
    printf("\nApproximate root (correct up to 3 decimal places) = %.3f\n", prev1);
    getch();
    return 0;
}
```

Output:

```
Itr     a       b       xr      f(a)    f(b)    f(xr)
---------------------------------------------------------------
 1    2.0000  3.0000  2.5000  -9.0000 6.0000  -3.3750
 2    2.5000  3.0000  2.7500  -3.3750 6.0000  0.7969
 3    2.5000  2.7500  2.6250  -3.3750 0.7969  -1.4121
 4    2.6250  2.7500  2.6875  -1.4121 0.7969  -0.3391
 5    2.6875  2.7500  2.7188  -0.3391 0.7969  0.2209
 6    2.6875  2.7188  2.7031  -0.3391 0.2209  -0.0611
 7    2.7031  2.7188  2.7109  -0.0611 0.2209  0.0794
 8    2.7031  2.7109  2.7070  -0.0611 0.0794  0.0090
 9    2.7031  2.7070  2.7051  -0.0611 0.0090  -0.0260
10    2.7051  2.7070  2.7061  -0.0260 0.0090  -0.0085
11    2.7061  2.7070  2.7065  -0.0085 0.0090  0.0003

Approximate root (correct up to 3 decimal places) = 2.707

_
```

Conclusion: The program was executed successfully.

Q.2) Finding Real Root using Regular Falsi Method in C Programming.

Aim: To write a C program to find the real root of an equation using the Regular Falsi Method.

Formula:

$$x_r = \frac{a \times f(b) - b \times f(a)}{f(b) - f(a)}$$

Programming Formula:

xr = (a * f(b) - b * f(a)) / (f(b) - f(a));

Program:

#include <stdio.h>

#include <conio.h>

#include <math.h>

float f(float x) {

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

}

int main() {

   float a = 2, b = 3, xr, fx, fb, fxr;

   int i = 0;

   float eps = 0.001;

   float prev1 = 0.0, prev2 = 0.0;

   int same_count = 0;

   clrscr();

   fx = f(a);

   fb = f(b);

   if (fx * fb > 0) {

      printf("Root not found in the given interval.\n");

      getch();

      return 0;

   }

   printf("\nIter\t a\t\t b\t\t xr\t\t f(a)\t\t f(b)\t\t f(xr)\n");

   printf("---\n");

```c
    do {
      i++;
      xr = (a * fb - b * fx) / (fb - fx);
      fxr = f(xr);
      fx = f(a);
      fb = f(b);
     printf("%d\t %.6f\t %.6f\t %.6f\t %.6f\t %.6f\t %.6f\n", i, a, b, xr, fx, fb, fxr);
      if (fx * fxr < 0)
        b = xr;
      else
        a = xr;
      float xr3 = floor(xr * 1000 + 0.5) / 1000;
      if (fabs(xr3 - prev1) < 0.0001 && fabs(prev1 - prev2) < 0.0001)
        same_count++;
      else
        same_count = 0;
      prev2 = prev1;
      prev1 = xr3;
      if (same_count >= 1)
        break;
    } while (fabs(fxr) > eps);
    printf("\nApproximate root (correct up to 3 decimal places) = %.3f\n", prev1);
    getch();
    return 0;
}
```

Output:

```
   Itr     a        b        xr      f(a)    f(b)     f(xr)
-------------------------------------------------------------
    1     2.0000   3.0000   2.5000  -9.0000 6.0000   -3.3750
    2     2.5000   3.0000   2.7500  -3.3750 6.0000    0.7969
    3     2.5000   2.7500   2.6250  -3.3750 0.7969   -1.4121
    4     2.6250   2.7500   2.6875  -1.4121 0.7969   -0.3391
    5     2.6875   2.7500   2.7188  -0.3391 0.7969    0.2209
    6     2.6875   2.7188   2.7031  -0.3391 0.2209   -0.0611
    7     2.7031   2.7188   2.7109  -0.0611 0.2209    0.0794
    8     2.7031   2.7109   2.7070  -0.0611 0.0794    0.0090
    9     2.7031   2.7070   2.7051  -0.0611 0.0090   -0.0260
   10     2.7051   2.7070   2.7061  -0.0260 0.0090   -0.0085
   11     2.7061   2.7070   2.7065  -0.0085 0.0090    0.0003

Approximate root (correct up to 3 decimal places) = 2.707

_
```

Conclusion: The program was executed successfully.

Q.3) Finding Real Root using Newton–Raphson Method in C Programming

Aim: To write a C program to find the real root of an equation using the Newton–Raphson Method.

Formula:

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

Programming Formula:

x1 = x0 - (f(x0) / f1(x0));

Program:

```
#include <stdio.h>

#include <conio.h>

#include <math.h>

float f(float x) {

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

}

float f1(float x) {

    return (3 * x * x - 4);

}

int main() {

    float x0 = 2, x1, f0, f1x;

    int i = 0;

    float eps = 0.001;

    float prev1 = 0.0, prev2 = 0.0;

    int same_count = 0;

    clrscr();

    printf("\nIter\t x0\t\t f(x0)\t\t f'(x0)\t\t x1\n");

    printf("-----------------------------------------------------------\n");

    do {

        i++;

        f0 = f(x0);

        f1x = f1(x0);
```

```
x1 = x0 - (f0 / f1x);

printf("%d\t %.6f\t %.6f\t %.6f\t %.6f\n", i, x0, f0, f1x, x1);

float x13 = floor(x1 * 1000 + 0.5) / 1000;

if (fabs(x13 - prev1) < 0.0001 && fabs(prev1 - prev2) < 0.0001)

    same_count++;

else

    same_count = 0;

prev2 = prev1;

prev1 = x13;

if (same_count >= 1)

    break;

x0 = x1;

} while (fabs(f0) > eps);

printf("\nApproximate root (correct up to 3 decimal places) = %.3f\n", prev1);

getch();

return 0;

}
```

Output:

```
Iter    x0              f(x0)           f'(x0)          x1
-----------------------------------------------------------------
1       2.000000        -9.000000       8.000000        3.125000
2       3.125000        9.017578        25.296875       2.768530
3       2.768530        1.145991        18.994274        2.708196
4       2.708196        0.030014        18.002983        2.706529
5       2.706529        0.000021        17.975901        2.706528

Approximate root (correct up to 3 decimal places) = 2.707
```

Conclusion: The program was executed successfully.

Interpolation

Q.1) Newton Forward Interpolation Table in C Programming

Aim: To create a Newton Forward Difference Table and interpolate a value using Newton's Forward Interpolation Formula.

Formula:

$$y(x) = y_0 + p\Delta y_0 + \frac{i(p-1)}{2!}\Delta^2 y_0 + \frac{p(p-1)(p-2)}{3!}\Delta^3 y_0 + \cdots$$

$$p = \frac{x - x_0}{h}, h = x_0 - x_1$$

Programming Formula:

y_int = y_int + (p_term * diff[0][i]) / fact(i);

p_term = p_term * (p - i);

Program:

```c
#include <stdio.h>
#include <conio.h>
float fact(int num)
{
   float f = 1;
   int i;
   for(i = 2; i <= num; i++)
     f = f * i;
   return f;
}
int main()
{
   int n = 5;
   float x[5] = {0, 1, 2, 3, 4};
   float y[5][5] = {
     {1}, {8}, {27}, {64}, {125}
```

```c
    };
    float value = 2.5, h, p, y_int, p_term;
    int i, j;
    clrscr();
    for(j = 1; j < n; j++)
        for(i = 0; i < n-j; i++)
            y[i][j] = y[i+1][j-1] - y[i][j-1];
    printf("\nNewton Forward Difference Table:\n");
    printf("x\tf(x)\ty\tDeltay\tDelta2y\tDelta3y\n");
    for(i = 0; i < n; i++)
    {
        printf("%.1f", x[i]);
        for(j = 0; j < n-i; j++)
            printf("\t%.2f", y[i][j]);
        printf("\n");
    }
    h = x[1] - x[0];
    p = (value - x[0]) / h;
    y_int = y[0][0];
    p_term = 1;
    for(i = 1; i < n; i++)
    {
        p_term = p_term * (p - i + 1);
        y_int = y_int + (p_term * y[0][i]) / fact(i);
    }
    printf("\nValue at x = %.2f is %.4f\n", value, y_int);
    getch();
    return 0;
}
```

Output:

```
Newton Forward Difference Table:
x       f(x)    y       Deltay  Delta2y Delta3y
0.0     1.00    7.00    12.00   6.00    0.00
1.0     8.00    19.00   18.00   6.00
2.0     27.00   37.00   24.00
3.0     64.00   61.00
4.0     125.00

Value at x = 2.50 is 42.8750

_
```

Conclusion: The program was executed successfully.

Q.2) Newton Backward Interpolation Table in C Programming

Aim: To write a C program to create a Newton Backward Difference Table and interpolate a value using Newton's Backward Interpolation Method.

Formula:

$$y(x) = y_n + p\Delta y_n + \frac{p(p + 1)}{2!}\Delta^2 y_n + \frac{p(p + 1)(p + 2)}{3!}\Delta^3 y_n + \cdots$$

$$p = \frac{x - x_0}{h}, h = x_0 - x_1$$

Programming Formula:

y_int = y_int + (p_term * diff[n-i-1][i]) / fact(i);

p_term = p_term * (p + i);

Program:

```c
#include <stdio.h>

#include <conio.h>

float fact(int num)

{

    float f = 1;

    int i;

    for(i = 2; i <= num; i++)

        f = f * i;

    return f;

}

int main()

{

    int n = 5;

    float x[5] = {0, 1, 2, 3, 4};

    float y[5][5] = {

        {1}, {8}, {27}, {64}, {125}

    };

    float value = 3.5, h, p, y_int, p_term;

    int i, j;
```

```c
clrscr();
for(j = 1; j < n; j++)
    for(i = n-1; i >= j; i--)
        y[i][j] = y[i][j-1] - y[i-1][j-1];
printf("\nNewton Backward Difference Table:\n");
printf("x\tf(x)\tDelta y\tDelta^2 y\tDelta^3 y\tDelta^4 y\n");
for(i = 0; i < n; i++)
{
    printf("%.1f", x[i]);
    for(j = 0; j <= i; j++)
        printf("\t%.2f", y[i][j]);
    printf("\n");
}
h = x[1] - x[0];
p = (value - x[n-1]) / h;
y_int = y[n-1][0];
p_term = 1;
for(i = 1; i < n; i++)
{
    p_term = p_term * (p + i - 1);
    y_int = y_int + (p_term * y[n-1][i]) / fact(i);
}
printf("\nValue at x = %.2f is %.4f\n", value, y_int);
getch();
return 0;
}
```

Output:

```
Newton Backward Difference Table:
x       f(x)     Deltay  Delta2y Delta3y Delta4y
0.0     1.00
1.0     8.00     7.00
2.0     27.00    19.00   12.00
3.0     64.00    37.00   18.00   6.00
4.0     125.00   61.00   24.00   6.00    0.00

Value at x = 3.50 is 91.1250
```

Conclusion: The program was executed successfully.

Numerical Integration

Q.1) To find the approximate value of a definite integral using the Trapezoidal Rule.

Aim: To find the approximate value of a definite integral using the Trapezoidal Rule.

Formula:

$$\int_{a}^{b} f(x)\, dx = \frac{h}{2}[y_0 + 2(y_1 + y_2 + y_3 + \cdots + y_{n-1}) + y_n]$$

Programming Formula :

```
sum = y0 + yn;
for(i = 1; i < n; i++)
sum += 2 * yi;
area = (h / 2) * sum;
```

Program:

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
float f(float x) {
   return x*x;
}
int main() {
   int n = 4;
   float a = 0.0, b = 2.0;
   float h, area, sum = 0.0;
   int i;
clrscr();
   h = (b - a) / n;
   sum = f(a) + f(b);
   for(i = 1; i < n; i++) {
      sum += 2 * f(a + i*h);
   }
```

area = (h / 2) * sum;

    printf("Approximate value of the integral: %.4f\n", area);

getch()

    return 0;

}

Output:

Conclusion: The program was executed successfully.

Q.2):To find the approximate value of a definite integral using Simpson's 1/3 Rule.

Aim: To calculate the area under the curve using Simpson's 1/3 Rule.

Formula:

$$\int_a^b f(x)dx \approx \frac{h}{3}[y_0 + 4(y_1 + y_3 + \cdots + y_{n-1}) + 2(y_2 + y_4 + \cdots + y_{n-2}) + y_n]$$

Programming Formula:

```
sum = y0 + yn;
for(i = 1; i < n; i++) {
if(i % 2 == 0)
sum += 2 * yi;
    else
sum += 4 * yi;
}
integral = (h / 3) * sum;
```

Program:

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
float f(float x) {
    return x*x;
}
int main() {
    int n = 4;
    float a = 0.0, b = 2.0;
    float h, area, sum = 0.0;
    int i;
clrscr();
    h = (b - a) / n;
    sum = f(a) + f(b);
    for(i = 1; i < n; i++) {
```

```
    if(i % 2 == 0)

        sum += 2 * f(a + i*h);

    else

        sum += 4 * f(a + i*h);

}

area = (h / 3) * sum;

printf("Approximate value of the integral (Simpson 1/3): %.4f\n", area);

getch();

return 0;

}
```

Output:

```
    Approximate value of the integral (Simpson 1/3): 2.6667
    _
```

Conclusion: The program was executed successfully.

Q.3) To find the approximate value of a definite integral using Simpson's 3/8 Rule.

Aim: To calculate the area under the curve using Simpson's 3/8 Rule.

Formula:

$$\int_a^b f(x)dx \approx \frac{3h}{8}[y_0 + 3(y_1 + y_2 + y_4 + y_5 + \cdots) + 2(y_3 + y_6 + \cdots) + y_n]$$

where

- n is a **multiple of 3**
- $h = \frac{b-a}{n}$

Programming Formula:

```
sum = y0 + yn;
for(i = 1; i < n; i++) {
    if(i % 3 == 0)
        sum += 2 * yi;
    else
        sum += 3 * yi;
}
integral = (3*h/8) * sum;
```

Program:

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
float f(float x) {
    return x*x;
}
int main() {
    int n = 6;
    float a = 0.0, b = 2.0;
    float h, area, sum = 0.0;
    int i;
    clrscr();
```

```
    h = (b - a) / n;

    sum = f(a) + f(b);

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

        if(i % 3 == 0)

            sum += 2 * f(a + i*h);

        else

            sum += 3 * f(a + i*h);

    }

    area = (3*h / 8) * sum;

    printf("Approximate value of the integral (Simpson 3/8): %.4f\n", area);

getch();

    return 0;

}
```

Output:

```
    Approximate value of the integral (Simpson 3/8): 2.6667
    _
```

Conclusion: The program was executed successfully.

Curve fitting

Q.1) To find the best-fitting curve for a set of data points using the method of least squares.

Aim: To determine the quadratic curve $y = a + bx + cx^2$ that best fits the given data points.

Formulas (Least Squares)

$$\sum y_i = na + b\sum x_i + c\sum x_i^2$$
$$\{\sum x_i y_i = a\sum x_i + b\sum x_i^2 + c\sum x_i^3$$
$$\sum x_i^2 y_i = a\sum x_i^2 + b\sum x_i^3 + c\sum x_i^4$$

Programming Formula :

sum_x = sum_x2 = sum_x3 = sum_x4 = 0;

sum_y = sum_xy = sum_x2y = 0;

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

   sum_x += x[i];

   sum_x2 += x[i]*x[i];

   sum_x3 += x[i]*x[i]*x[i];

   sum_x4 += x[i]*x[i]*x[i]*x[i];

   sum_y += y[i];

   sum_xy += x[i]*y[i];

   sum_x2y += x[i]*x[i]*y[i];

}

a = Da / D;

b = Db / D;

c = Dc / D;

Program:

#include <stdio.h>

#include <conio.h>

int main() {

   clrscr();

   int n = 5;

```
float x[] = {1, 2, 3, 4, 5};

float y[] = {2.2, 2.8, 3.6, 4.5, 5.1};

float sum_x=0, sum_x2=0, sum_x3=0, sum_x4=0;

float sum_y=0, sum_xy=0, sum_x2y=0;

int i;

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

   sum_x += x[i];

   sum_x2 += x[i]*x[i];

   sum_x3 += x[i]*x[i]*x[i];

   sum_x4 += x[i]*x[i]*x[i]*x[i];

   sum_y += y[i];

   sum_xy += x[i]*y[i];

   sum_x2y += x[i]*x[i]*y[i];

}

float D = n*(sum_x2*sum_x4 - sum_x3*sum_x3) - sum_x*(sum_x*sum_x4 -
sum_x2*sum_x3) + sum_x2*(sum_x*sum_x3 - sum_x2*sum_x2);

float Da = sum_y*(sum_x2*sum_x4 - sum_x3*sum_x3) - sum_x*(sum_xy*sum_x4 -
sum_x2y*sum_x3) + sum_x2*(sum_xy*sum_x3 - sum_x2y*sum_x2);

float Db = n*(sum_xy*sum_x4 - sum_x2y*sum_x3) - sum_y*(sum_x*sum_x4 -
sum_x2*sum_x3) + sum_x2*(sum_x*sum_x2y - sum_y*sum_x2);

float Dc = n*(sum_x2*sum_x2y - sum_x3*sum_xy) - sum_x*(sum_x*sum_x2y -
sum_x3*sum_y) + sum_y*(sum_x*sum_x3 - sum_x2*sum_x2);

float a = Da / D;

float b = Db / D;

float c = Dc / D;

printf("Quadratic curve fitting equation: y = %.4f + %.4fx + %.4fx^2\n", a, b, c);

getch();

return 0;

}
```

Output:

```
Quadratic curve fitting equation: y = 1.4400 + 190.4178x + 14.5679x^2
```

Conclusion: The program was executed successfully.

Q.1) Runge–Kutta 4th Order Method in C Programming.

Aim: To solve the differential equation $y' = x + y$ using the Fourth Order Runge–Kutta Method and compute the values of y for the given step size.

Formula:

$$k_1 = h f(x_0, y_0)$$
$$k_2 = h f(x_0 + \frac{h}{2}, y_0 + \frac{k_1}{2})$$
$$k_3 = h f(x_0 + \frac{h}{2}, y_0 + \frac{k_2}{2})$$
$$k_4 = h f(x_0 + h, y_0 + k_3)$$

Weighted average slope:

$$k = \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}$$

Final update formula:

$$y_1 = y_0 + k$$
$$x_1 = x_0 + h$$

Programming Formula:

k1 = H * f(x, y);

k2 = H * f(x + H/2, y + k1/2);

k3 = H * f(x + H/2, y + k2/2);

k4 = H * f(x + H, y + k3);

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

y = y + kavg;

x = x + H;

Program:

#include <stdio.h>

#include <math.h>

#include <conio.h>

double f(double x, double y) {

```c
    return x + y;
}
void runge_kutta_4th_order_solve() {
    const double X0 = 0.0;
    const double Y0 = 1.0;
    const double H = 0.1;
    const int N = 5;
    double x = X0, y = Y0;
    double k1, k2, k3, k4;
    int i;
    clrscr();
    printf("---- RK4 Method ----\n");
    printf("y' = x + y , h=%.1f\n", H);
    printf("Step |   x   |   y   | k1 |  k2  |  k3  |  k4  | kavg\n");
    printf("------------------------------------------------------------------------\n");
    printf("0    | %7.4f | %7.4f |\n", x, y);
    for (i = 1; i <= N; i++) {
        k1 = H * f(x, y);
        k2 = H * f(x + H/2, y + k1/2);
        k3 = H * f(x + H/2, y + k2/2);
        k4 = H * f(x + H, y + k3);
        double kavg = (k1 + 2*k2 + 2*k3 + k4) / 6.0;
        y = y + kavg;
        x = x + H;
        printf("%-4d | %7.4f | %7.4f | %7.4f | %7.4f | %7.4f | %7.4f | %7.4f\n",
            i, x, y, k1, k2, k3, k4, kavg);
    }

    printf("------------------------------------------------------------------------\n");
    printf("Final y(%.1f) = %.6f", x, y);
```

```
}
int main() {

    runge_kutta_4th_order_solve();

    getch();

    return 0;

}
```

Code:

```
---- Runge-Kutta Method ----
y' = x + y , h=0.1
Step |    x    |    y    |   k1   |   k2   |   k3   |   k4   |  kavg
----------------------------------------------------------------------
0    |  0.0000 |  1.0000 |
1    |  0.1000 |  1.1103 |  0.1000 |  0.1100 |  0.1105 |  0.1211 |  0.1103
2    |  0.2000 |  1.2428 |  0.1210 |  0.1321 |  0.1326 |  0.1443 |  0.1325
3    |  0.3000 |  1.3997 |  0.1443 |  0.1565 |  0.1571 |  0.1700 |  0.1569
4    |  0.4000 |  1.5836 |  0.1700 |  0.1835 |  0.1841 |  0.1984 |  0.1839
5    |  0.5000 |  1.7974 |  0.1984 |  0.2133 |  0.2140 |  0.2298 |  0.2138
----------------------------------------------------------------------
Final y(0.5) = 1.797441
```

Conclusion: The program was executed successfully.

Solution Of Simultaneous Algebraic Equations

Q.1) Gauss Elimination Method.

Aim: To write a C program to find the solution of **simultaneous algebraic equations** using the **Gauss Elimination Method**.

Formula:

For a system of 3 equations:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\
a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\
a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3
\end{aligned}$$

Row transformation used:

$$R_j \;\rightarrow\; R_j - (\frac{a_{ji}}{a_{ii}})R_i$$

Programming formula:

ratio = a[j][i] / a[i][i];

a[j][k] = a[j][k] - ratio * a[i][k];

Program:

```c
#include <stdio.h>
#include <conio.h>
#include <math.h>
int main()
{
   float a[10][11] = {
      {  5, -3,  2,  12 },
      { -4,  6,  1,  -7 },
      {  3,  2, -5,  10 }
   };
   float x[10], ratio, temp;
   int n = 3;
```

```c
int i, j, k, max_row;
clrscr();
printf("\nSolving the augmented matrix:\n");
for(i = 0; i < n; i++)
{
    for(j = 0; j <= n; j++)
    {
        printf("%0.0f ", a[i][j]);
    }
    printf("\n");
}
for(i = 0; i < n; i++)
{
    max_row = i;
    for(k = i + 1; k < n; k++)
    {
        if(fabs(a[k][i]) > fabs(a[max_row][i]))
            max_row = k;
    }
    for(k = 0; k <= n; k++)
    {
        temp = a[i][k];
        a[i][k] = a[max_row][k];
        a[max_row][k] = temp;
    }
    if(a[i][i] == 0.0)
    {
        printf("\nMathematical Error: No unique solution\n");
        getch();
        return 0;
```

```
    }
    for(j = 0; j < n; j++)
    {
        if(i != j)
        {
            ratio = a[j][i] / a[i][i];
            for(k = 0; k <= n; k++)
            {
                a[j][k] = a[j][k] - ratio * a[i][k];
            }
        }
    }
}
for(i = 0; i < n; i++)
{
    x[i] = a[i][n] / a[i][i];
}
printf("\nThe solution is:\n");
for(i = 0; i < n; i++)
{
    printf("x%d = %0.3f\n", i+1, x[i]);
}

getch();
return 0;
}
```

Output:

```
Solving the augmented matrix:
5 -3 2 12
-4 6 1 -7
3 2 -5 10

The solution is:
x1 = 2.839
x2 = 0.727
x3 = -0.006
_
```

Conclusion: The program was executed successfully.

Numerical solutions of first and second order differential equation

Q.1) Euler's Method in C Programming.

Aim: To solve the differential equation $y' = x + y$ using Standard Euler's Method and compute the approximate values of y for a given step size.

Formula:

$$y_{n+1} = y_n + h \cdot f(x_n, y_n)$$

Programming Formula:

slope = f(x, y);

y = y + STEP_SIZE * slope;

x = x + STEP_SIZE;

Program:

```
#include <stdio.h>

#include <math.h>

#include <conio.h>

double f(double x, double y) {

    return x + y;

}

void eulers_method_solve() {

    const double X_INITIAL = 0.0;

    const double Y_INITIAL = 1.0;

    const double STEP_SIZE = 0.1;

    const int N_STEPS = 5;

    double x = X_INITIAL;

    double y = Y_INITIAL;

    double slope;

    int i;

clrscr();

    printf("--- Standard Euler's Method (y' = x + y) ---\n");

    printf("Initial Condition: y(%.1f) = %.1f, Step Size h = %.1f\n", X_INITIAL, Y_INITIAL, STEP_SIZE);
```

```
    printf("Step |    x    |    y    |   f(x, y)\n");

    printf("---------------------------------------------------\n");

    printf(" 0   |%10.6f| %10.6f |\n", x, y);

    for (i = 1; i <= N_STEPS; i++) {

        slope = f(x, y);

        y = y + STEP_SIZE * slope;

        x = x + STEP_SIZE;

        printf(" %d   |%10.6f| %10.6f| %10.6f\n", i, x, y, slope);

    }

    printf("---------------------------------------------------\n");

    printf("Final Approximation at x=%.1f: y = %.6f\n", x, y);

}

int main() {

    eulers_method_solve();

getch();

    return 0;

}
```

Output:

```
--- Standard Euler's Method (y' = x + y) ---
Initial Condition: y(0.0) = 1.0, Step Size h = 0.1
Step |     x      |      y      |    f(x, y)
------------------------------------------------------
  0  |   0.000000 |   1.000000 |
  1  |   0.100000 |   1.100000 |   1.000000
  2  |   0.200000 |   1.220000 |   1.200000
  3  |   0.300000 |   1.362000 |   1.420000
  4  |   0.400000 |   1.528200 |   1.662000
  5  |   0.500000 |   1.721020 |   1.928200
------------------------------------------------------
Final Approximation at x=0.5: y = 1.721020

 _
```

Conclusion: The program was executed successfully.

Q.2) Modified Euler's Method in C Programming

Aim: To solve the differential equation $y' = x + y$ using the Modified Euler's Method and compute the approximate values of y for each step.

Formula:

$$y_{n+1} = y_n + \frac{h}{2}[f(x_n, y_n) + f(x_{n+1}, y_p)]$$

$$x_{n+1} = x_n + h$$

Programming Formula:

k1 = f(x, y);

y_predictor = y + H * k1;

k2 = f(x_next, y_predictor);

y = y + (H / 2.0) * (k1 + k2);

x = x_next;

Program:

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
double f(double x, double y) {
    return x + y;
}
void modified_eulers_method_solve() {
    const double X_INITIAL = 0.0;
    const double Y_INITIAL = 1.0;
    const double H = 0.1;
    const int N_STEPS = 5;
    double x = X_INITIAL;
    double y = Y_INITIAL;
```

```c
    double x_next;

    double y_predictor;

    double k1, k2;

    int i;

clrscr();

    printf("--- Modified Euler's Method (y' = x + y) ---\n");

    printf("Initial Condition: y(%.1f) = %.1f, Step Size h = %.1f\n", X_INITIAL, Y_INITIAL, H);

    printf("Step |    x    |    y    | Predictor y | Avg. Slope\n");

    printf("-------------------------------------------------------------\n");

    printf(" 0   | %10.6f | %10.6f |\n", x, y);

    for (i = 1; i <= N_STEPS; i++) {

        k1 = f(x, y);

        x_next = x + H;

        y_predictor = y + H * k1;

        k2 = f(x_next, y_predictor);

        y = y + (H / 2.0) * (k1 + k2);

        x = x_next;

        printf(" %d   | %10.6f | %10.6f | %10.6f | %10.6f\n",
            i, x, y, y_predictor, (k1 + k2) / 2.0);

    }

    printf("-------------------------------------------------------------\n");

    printf("Final Approximation at x=%.1f: y = %.6f\n", x, y);

}

int main() {

    modified_eulers_method_solve();

getch();

    return 0;

}
```

Output:

```
--- Modified Euler's Method (y' = x + y) ---
Initial Condition: y(0.0) = 1.0, Step Size h = 0.1
Step |      x      |      y      | Predictor y | Avg. Slope
-----------------------------------------------------------------
 0   |  0.000000 |  1.000000 |
 1   |  0.100000 |  1.110000 |  1.100000 |  1.100000
 2   |  0.200000 |  1.242050 |  1.231000 |  1.320500
 3   |  0.300000 |  1.398465 |  1.386255 |  1.564153
 4   |  0.400000 |  1.581804 |  1.568312 |  1.833389
 5   |  0.500000 |  1.794894 |  1.779985 |  2.130894
-----------------------------------------------------------------

Final Approximation at x=0.5: y = 1.794894

_
```

Conclusion: The program was executed successfully.

Q. 3) Taylor Series Method

Aim: To solve the differential equation $y' = x + y$ using the Taylor Series Method with given initial conditions.

Formula:

$$y_{i+1} = y_i + h\,y_i' + \frac{h^2}{2}\,y_i''$$
$$y_i' = f(x_i, y_i) = x_i + y_i$$
$$y_i'' = f'(x_i, y_i) = 1 + x_i + y_i$$

Programming Formula:

y_prime = f(x, y);

y_double_prime = f_prime(x, y);

y = y + h * y_prime + (h * h / 2) * y_double_prime;

Program:

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
double f(double x, double y) {
    return x + y;
}
double f_prime(double x, double y) {
    return 1.0 + x + y;
}
void taylor_series_solve() {
    const double X_INITIAL = 0.0;
    const double Y_INITIAL = 1.0;
    const double H = 0.1;
    const int N_STEPS = 5;
    double x = X_INITIAL;
    double y = Y_INITIAL;
    double y_prime;
```

```c
    double y_double_prime;

    int i;
clrscr();
    printf("--- Taylor Series Method (y' = x + y) ---\n");
    printf("Initial Condition: y(%.1f) = %.1f, Step Size h = %.1f\n", X_INITIAL, Y_INITIAL,
H);
    printf("Step |    x    |    y    |   y' (f)  |   y" (f) \n");
    printf("----------------------------------------------------------------\n");
    printf(" 0   | %10.6f | %10.6f | \n", x, y);
    for (i = 1; i <= N_STEPS; i++) {
        y_prime = f(x, y);
        y_double_prime = f_prime(x, y);
        y = y + H * y_prime + (H * H / 2.0) * y_double_prime;
        x = x + H;
        printf(" %d   | %10.6f | %10.6f | %10.6f | %10.6f\n", i, x, y, y_prime, y_double_prime);
    }
    printf("----------------------------------------------------------------\n");
    printf("Final Approximation at x=%.1f: y = %.6f\n", x, y);
}
int main() {
    taylor_series_solve();
    getch();
    return 0;
}
```

Output:

```
--- Taylor Series Method (y' = x + y) ---
Initial Condition: y(0.0) = 1.0, Step Size h = 0.1
Step |      x      |      y      |    y' (f)   |    y'' (f')
----------------------------------------------------------------
  0  |  0.000000  |  1.000000  |
  1  |  0.100000  |  1.110000  |  1.000000  |  2.000000
  2  |  0.200000  |  1.242050  |  1.210000  |  2.210000
  3  |  0.300000  |  1.398465  |  1.442050  |  2.442050
  4  |  0.400000  |  1.581804  |  1.698465  |  2.698465
  5  |  0.500000  |  1.794894  |  1.981804  |  2.981804
----------------------------------------------------------------

Final Approximation at x=0.5: y = 1.794894
```

Conclusion: The program was executed successfully.

Transportation problem

Q.1) Transportation Problem using the **Least Cost Method**.

Aim:

To write a C program to find the **initial feasible solution** of a **Transportation Problem** using the **Least Cost Method**, and to compute the **minimum transportation cost**.

Formula:

1. Least Cost Selection Rule

Choose the cell with:

$$\text{Minimum Cost} = \min(c_{ij})$$

Allocation Rule

$$x_{ij} = \min(\text{Supply}_i, \text{Demand}_j)$$

Update Supply & Demand

$$\text{Supply}_i = \text{Supply}_i - x_{ij}$$
$$\text{Demand}_j = \text{Demand}_j - x_{ij}$$

Total Transportation Cost

$$\text{Total Cost} = \sum(x_{ij} \times c_{ij})$$

Programming Formula:

min_cost = temp_cost[i][j];

alloc = (supply_left[min_i] < demand_left[min_j]) ? supply_left[min_i] : demand_left[min_j];

supply_left[min_i] -= alloc;

demand_left[min_j] -= alloc;

total_cost += alloc * cost[min_i][min_j];

Program:

```
#include <stdio.h>

#include <conio.h>

int main() {

    int cost[3][4] = {
```

```c
    { 4, 7, 6, 3 },

    { 8, 5, 9, 2 },

    { 1, 4, 3, 8 }

};

int supply[3] = { 40, 35, 45 };

int demand[4] = { 30, 25, 50, 15 };

int allocation[3][4];

int i, j;

int min_cost, min_i, min_j, alloc;

long total_cost = 0;

int items_to_alloc = 0;

int supply_left[3];

int demand_left[4];

int temp_cost[3][4];

clrscr();

printf("======TRANSPORTATION PROBLEM======\n\n");

printf("Cost Matrix:\n");

for (i = 0; i < 3; i++) {

    for (j = 0; j < 4; j++) {

        printf("%4d", cost[i][j]);

    }

    printf("   | Supply: %d\n", supply[i]);

}

printf("\nDemand:");

for (j = 0; j < 4; j++) {

    printf("%4d", demand[j]);

}

printf("\n\n");

for (i = 0; i < 3; i++) {

    supply_left[i] = supply[i];
```

```
      items_to_alloc += supply[i];

      for (j = 0; j < 4; j++) {

         allocation[i][j] = 0;

         temp_cost[i][j] = cost[i][j];

      }

   }

   for (j = 0; j < 4; j++) {

      demand_left[j] = demand[j];

   }

   while (items_to_alloc > 0) {

      min_cost = 9999;

      min_i = -1;

      min_j = -1;

      for (i = 0; i < 3; i++) {

         for (j = 0; j < 4; j++) {

            if (temp_cost[i][j] < min_cost) {

               min_cost = temp_cost[i][j];

               min_i = i;

               min_j = j;

            }

         }

      }

      if (supply_left[min_i] < demand_left[min_j]) {

         alloc = supply_left[min_i];

      } else {

         alloc = demand_left[min_j];

      }

      allocation[min_i][min_j] = alloc;

      total_cost += (long)alloc * cost[min_i][min_j];

      items_to_alloc -= alloc;
```

```c
        supply_left[min_i] -= alloc;

        demand_left[min_j] -= alloc;

        if (supply_left[min_i] == 0) {

            for (j = 0; j < 4; j++) {

                temp_cost[min_i][j] = 9999;

            }

        }

        if (demand_left[min_j] == 0) {

            for (i = 0; i < 3; i++) {

                temp_cost[i][min_j] = 9999;

            }

        }

    }

    printf("=======LEAST COST METHOD SOLUTION=======\n\n");

    printf("Allocation Matrix:\n");

    for (i = 0; i < 3; i++) {

        for (j = 0; j < 4; j++) {

            if (allocation[i][j] > 0) {

                printf("%4d", allocation[i][j]);

            } else {

                printf("%4s", "-");

            }

        }

        printf("\n");

    }

    printf("\nTotal Transportation Cost: %ld", total_cost);

    getch();

return 0;

}
```

Output:

```
=======TRANSPORTATION PROBLEM=======

Cost Matrix:
    4   7   6   3   ¦ Supply: 40
    8   5   9   2   ¦ Supply: 35
    1   4   3   8   ¦ Supply: 45

Demand:  30  25  50  15

=======LEAST COST METHOD SOLUTION=======

Allocation Matrix:
    -   5  35   -
    -  20   -  15
   30   -  15   -

Total Transportation Cost: 450
```

Conclusion: The program was executed successfully.