Question 1: */\**

*Write a program to implement Jacobi iteration method for system of Linear equations.*

*Exam Roll No. --> 20220PHY014*

*Given Equations:  20x + y - 2z = 17*

*3x + 20y - z = -18*

*2x - 3y + 20z = 25*

*So, x = (17 - y + 2z) / 20*

*y = (-18 - 3x + z) / 20*

*z = (25 - 2x + 3y) / 20*

*\*/*

#include <stdio.h>

#include <math.h>

#define f1(*x*, *y*, *z*) (17 - y + 2 \* z) / 20

#define f2(*x*, *y*, *z*) (-18 - 3 \* x + z) / 20

#define f3(*x*, *y*, *z*) (25 - 2 \* x + 3 \* y) / 20

#define ToleranceError 0.0001

int main()

{

   float x0 = 0, y0 = 0, z0 = 0, x1, y1, z1, error\_in\_x, error\_in\_y, error\_in\_z; *//notes: 0, 0, 0 is the initial guess*

   int iteration = 1;

   do

   {

      x1 = f1(x0, y0, z0);

      y1 = f2(x0, y0, z0);

      z1 = f3(x0, y0, z0);

      printf("Iteration *%d*:\nx = *%0.4f*\ny = *%0.4f*\nz = *%0.4f*\n\n", iteration, x1, y1, z1);

      error\_in\_x = fabs(x0 - x1);

      error\_in\_y = fabs(y0 - y1);

      error\_in\_z = fabs(z0 - z1);

      iteration++;

*//\* Assigning values for next iteration*

      x0 = x1;

      y0 = y1;

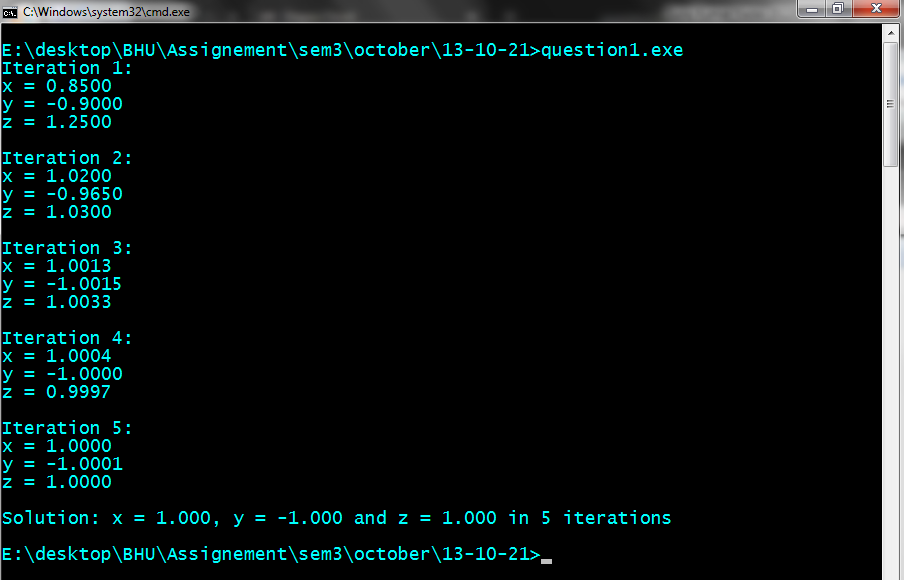
      z0 = z1;

   } while (error\_in\_x > ToleranceError && error\_in\_y > ToleranceError && error\_in\_z > ToleranceError);

   printf("Solution: x = *%0.3f*, y = *%0.3f* and z = *%0.3f* in *%d* iterations\n", x1, y1, z1, iteration - 1);

   return 0;

}

Output: 

Question 2: */\**

*Write a program to implement Gauss Seidel method for system of Linear equations.*

*Exam Roll No. --> 20220PHY014*

*Given Equations:  20x + y - 2z = 17*

*3x + 20y - z = -18*

*2x - 3y + 20z = 25*

*So, x = (17 - y + 2z) / 20*

*y = (-18 - 3x + z) / 20*

*z = (25 - 2x + 3y) / 20*

*\*/*

#include <stdio.h>

#include <math.h>

#define f1(*x*, *y*, *z*) (17 - y + 2 \* z) / 20

#define f2(*x*, *y*, *z*) (-18 - 3 \* x + z) / 20

#define f3(*x*, *y*, *z*) (25 - 2 \* x + 3 \* y) / 20

#define ToleranceError 0.0001

int main()

{

   float x0 = 0, y0 = 0, z0 = 0, x1, y1, z1, error\_in\_x, error\_in\_y, error\_in\_z; *//notes: 0, 0, 0 is the initial guess*

   int iteration = 1;

   do

   {

      x1 = f1(x0, y0, z0);

      y1 = f2(x1, y0, z0);

      z1 = f3(x1, y1, z0);

      printf("Iteration *%d*:\nx = *%0.4f*\ny = *%0.4f*\nz = *%0.4f*\n\n", iteration, x1, y1, z1);

      error\_in\_x = fabs(x0 - x1);

      error\_in\_y = fabs(y0 - y1);

      error\_in\_z = fabs(z0 - z1);

      iteration++;

*//\* Assigning values for next iteration*

      x0 = x1;

      y0 = y1;

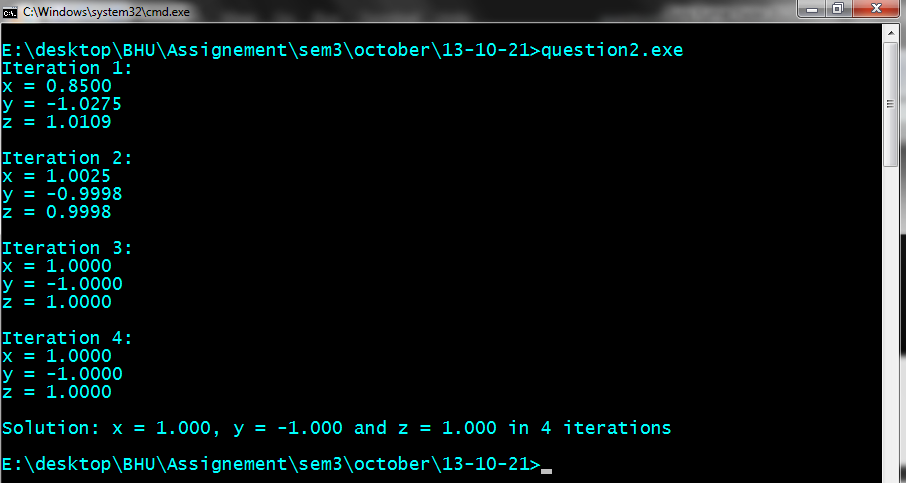
      z0 = z1;

   } while (error\_in\_x > ToleranceError && error\_in\_y > ToleranceError && error\_in\_z > ToleranceError);

   printf("Solution: x = *%0.3f*, y = *%0.3f* and z = *%0.3f* in *%d* iterations\n", x1, y1, z1, iteration - 1);

   return 0;

}

Output: 

Question 3: *// Write a program to implement Power method to find the largest Eigen values and vectors.*

*// Exam Roll No. --> 20220PHY014*

#include <stdio.h>

#include <math.h>

#define SIZE 10

#define ToleranceError 0.0001

int main()

{

    float x\_new[SIZE];

    float temp, lambda\_new, lambda\_old;

    int i, j, matrix\_order = 3, iteration = 1;

    float matrix[SIZE][SIZE] = {{1, 2, 0},

                                {2, 1, 0},

                                {0, 0, -1}};

    float x[3] = {0, 1, 0}; *//notes: initial guess is 0, 1, 0*

    lambda\_old = 1;

run\_again:

*//\* finding x\_new*

    for (int i = 0; i < matrix\_order; i++)

    {

        temp = 0.0;

        for (int j = 0; j < matrix\_order; j++)

            temp += matrix[i][j] \* x[j];

        x\_new[i] = temp;

    }

*//\* Assigning x\_new to x*

    for (int i = 0; i < matrix\_order; i++)

        x[i] = x\_new[i];

*//\* Finding Largest element in x*

    lambda\_new = fabs(x[0]);

    for (int i = 1; i < matrix\_order; i++)

    {

        if (fabs(x[i]) > lambda\_new)

            lambda\_new = fabs(x[i]);

    }

*//\* Eigen matrix after dividing by largest element*

    for (int i = 0; i < matrix\_order; i++)

        x[i] /= lambda\_new;

    printf("\n\nIteration *%d*:\n", iteration);

    printf("Eigen Value = *%f*\n", lambda\_new);

    printf("Eigen Vector = ");

    for (int i = 0; i < matrix\_order; i++)

    {

        if (i == 0)

            printf("[*%f*\t", x[i]);

        else if (i == 2)

            printf("*%f*]", x[i]);

        else

            printf("*%f*\t", x[i]);

    }

*//\* repeat this process till the required condition is meet*

    if (fabs(lambda\_new - lambda\_old) > ToleranceError)

    {

        lambda\_old = lambda\_new;

        iteration++;

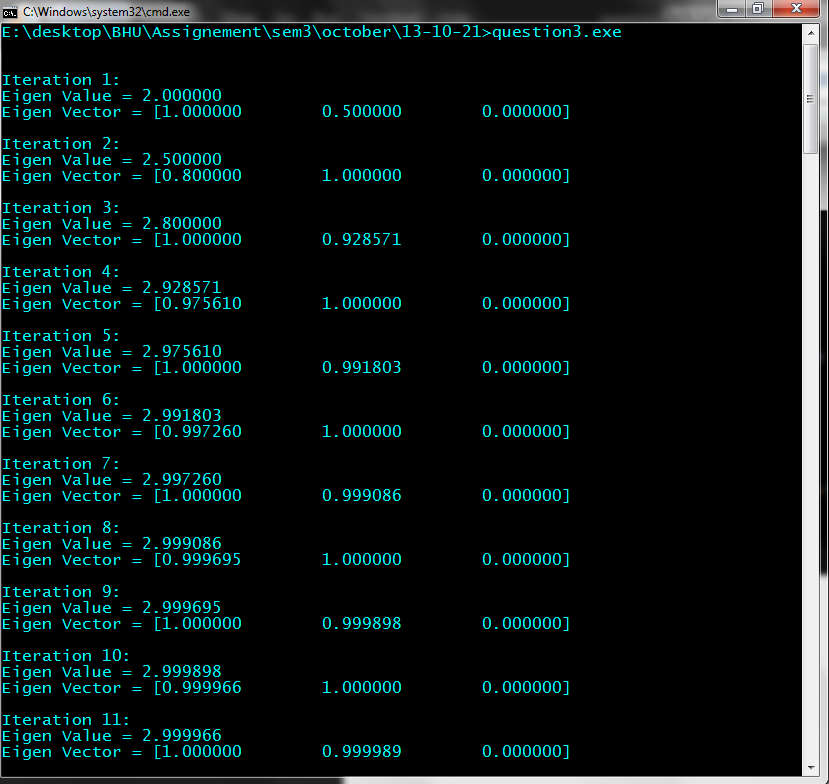
        goto run\_again;

    }

    printf("\n\nSo, finally after *%d* iterations\nEigen Value = *%f*\nAnd\nEigen Vector = [*%f*\t*%f*\t*%f*]\n", iteration, lambda\_new, x[0], x[1], x[2]);

    return (0);

}

Output: 

Question 4: *// Write a program to implement Jacobi method for diagonalization of real symmetric matrix.*

*// Exam Roll No. --> 20220PHY014*

#include <stdio.h>

#include <math.h>

#define N 2               *//notes: N is number of loops*

#define Tolerance 0.00001 *//notes: Tolerance for the accuracy of the result*

float x, angle, large, sum = 0; *//notes: large for finding largest element in matrix and sum used in product of matrix*

int iNew = 0, jNew = 0;         *//notes: corresponding to largest element in matrix*

void findingResult(float *a\_mat*[3][3], float *b\_mat*[3][3], float *b\_inv\_mat*[3][3], float *b\_inv\_a*[3][3], float *d\_b\_inv\_a\_b*[3][3]);

void product\_matrix(float *mat1*[3][3], float *mat2*[3][3], float *product*[3][3]);

void display\_matrix(float *mat*[3][3]);

int main()

{

    float b\_matrix[3][3], bInverse[3][3], b\_inverse\_a[3][3], final[3][3];

    float a\_matrix[3][3] = {{1, 1.41421, 2},

                            {1.41421, 3, 1.41421},

                            {2, 1.41421, 1}};

    for (int i = 0; i < N; i++)

    {

        printf("Iteration *%d*:\n", i + 1);

*//\* finding largest element*

        large = 0;

        for (int i = 0; i < 3; i++)

        {

            for (int j = 0; j < 3; j++)

            {

                if ((i != j) && (a\_matrix[i][j] >= large))

                {

                    large = a\_matrix[i][j];

                    iNew = i, jNew = j;

                }

            }

        }

*//\* finding angle for orthogonal matrix*

        if (a\_matrix[iNew][iNew] == a\_matrix[jNew][jNew])

            angle = M\_PI\_4;

        else

        {

            x = (2 \* a\_matrix[iNew][jNew]) / (a\_matrix[iNew][iNew] - a\_matrix[jNew][jNew]);

            angle = atan(x) / 2;

        }

*//\* assigning orthogonal matrices --> if the matrix is symmetric then there are only 3(each has one symmetric) possible position for largest off-diagonal element*

        if (iNew == 1 && jNew == 0)

        {

            float b\_matrix[3][3] = {{cos(angle), -sin(angle), 0},

                                    {sin(angle), cos(angle), 0},

                                    {0, 0, 1}};

            float bInverse[3][3] = {{cos(angle), sin(angle), 0},

                                    {-sin(angle), cos(angle), 0},

                                    {0, 0, 1}};

            findingResult(a\_matrix, b\_matrix, bInverse, b\_inverse\_a, final);

        }

        else if (iNew == 2 && jNew == 0)

        {

            float b\_matrix[3][3] = {{cos(angle), 0, -sin(angle)},

                                    {0, 1, 0},

                                    {sin(angle), 0, cos(angle)}};

            float bInverse[3][3] = {{cos(angle), 0, sin(angle)},

                                    {0, 1, 0},

                                    {-sin(angle), 0, cos(angle)}};

            findingResult(a\_matrix, b\_matrix, bInverse, b\_inverse\_a, final);

        }

        else if (iNew == 2 && jNew == 1)

        {

            float b\_matrix[3][3] = {{1, 0, 0},

                                    {0, cos(angle), -sin(angle)},

                                    {0, sin(angle), cos(angle)}};

            float bInverse[3][3] = {{1, 0, 0},

                                    {0, cos(angle), sin(angle)},

                                    {0, -sin(angle), cos(angle)}};

            findingResult(a\_matrix, b\_matrix, bInverse, b\_inverse\_a, final);

        }

*//\* Assigning new matrix A equals to the final previous matrix*

        for (int i = 0; i < 3; i++)

        {

            for (int j = 0; j < 3; j++)

                a\_matrix[i][j] = final[i][j];

        }

        if (i == (N - 1))

        {

            int k = 0;

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

            {

                for (int j = 0; j < 3; j++)

                {

                    if ((i != j) && (a\_matrix[i][j] >= Tolerance))

                        k++;

                }

            }

            if (k != 0) *// if the resultant matrix not match the required Tolerance*

                printf("Try with more number of loops to get more accurate result\n");

        }

    }

    return 0;

}

void findingResult(float *a\_mat*[3][3], float *b\_mat*[3][3], float *b\_inv\_mat*[3][3], float *b\_inv\_a*[3][3], float *d\_b\_inv\_a\_b*[3][3])

{

    sum = 0;

    product\_matrix(*b\_inv\_mat*, *a\_mat*, *b\_inv\_a*);

    sum = 0;

    product\_matrix(*b\_inv\_a*, *b\_mat*, *d\_b\_inv\_a\_b*);

    printf("Matrix A is:\n");

    display\_matrix(*a\_mat*);

    printf("Matrix D is:\n");

    display\_matrix(*d\_b\_inv\_a\_b*);

}

void product\_matrix(float *mat1*[3][3], float *mat2*[3][3], float *product*[3][3])

{

    for (int i = 0; i < 3; i++)

    {

        for (int j = 0; j < 3; j++)

        {

            for (int k = 0; k < 3; k++)

                sum += *mat1*[i][k] \* *mat2*[k][j];

*product*[i][j] = sum;

            sum = 0;

        }

    }

}

void display\_matrix(float *mat*[3][3])

{

    for (int i = 0; i < 3; i++)

    {

        for (int j = 0; j < 3; j++)

            printf("*%-6f*\t", *mat*[i][j]);

        printf("\n");

    }

    printf("\n");

}

Output: 