**Họ và tên: Nguyễn Hoàng Tuấn Anh**

**MSSV: 102220219**

**Đề: 1**

Bài 1: Source code:

#include <bits/stdc++.h>

using namespace std;

#define ll long long

ll **Mcalc**(ll *m*[], int *size*)

{

    ll M = 1;

    for (int i = 0; i<*size*; i++)

    {

        M\*=*m*[i];

    }

    return M;

}

ll **X\_of**(ll *m*[], int *index*, int *size*)

{

    ll X = 1;

    int i = 0;

    while (i<*size*)

    {

        if (i == *index*)

        {

            i++;

        }

        else{

            X \*= *m*[i];

            i++;

        }

    }

    return X;

}

ll **index\_invertOf**(ll *m*[], int *index*, int *size*)

{

    ll temp = **X\_of**(*m*,*index*,*size*);

    int k = 0;

    while ((k\**m*[*index*] + 1) % temp != 0)

    {

        k++;

    }

    ll temp\_equal = (k\**m*[*index*] + 1) / temp;

    return temp\_equal;

}

void **ChineseTheorem**(int *size*, ll *a*[], ll *m*[])

{

    ll M\_temp = **Mcalc**(*m*,*size*);

    ll S = 0;

    for (int i = 0; i<*size*; i++)

    {

        ll temp\_X = **X\_of**(*m*, i, *size*);

        ll temp\_Y = **index\_invertOf**(*m*, i, *size*);

        S += *a*[i]\*temp\_X\*temp\_Y;

    }

    S = S % M\_temp;

    while(S % M\_temp > M\_temp)

    {

        S = S % M\_temp;

    }

    cout << "Chinese theorem: X = " << S << " + " << "k\*" << M\_temp << **endl**;

}

int **main**()

{

    int size = 5;

    ll a[size] = {2,3,4,5,6};

    ll m[size] = {5,7,9,11,13};

    cout << "All equations: \n";

    for (int i = 0; i< size; i++)

    {

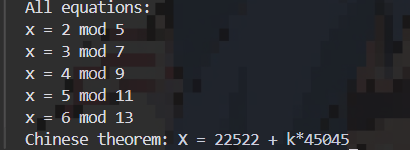
        cout << "x = " << a[i] << " mod " << m[i] << "\n";

    }

**ChineseTheorem**(size,a,m);

}

Kết quả:



Bài 2: Source code:

2.a:

#include <bits/stdc++.h>

using namespace std;

#define MAX 100

void **Identify**(float *a*[][MAX], int *n*)

{

    for (int i = 0; i < *n*; i++)

    {

        for (int j = 0; j < *n*; j++)

        {

            if (i == j)

*a*[i][j] = 1;

            else

*a*[i][j] = 0;

        }

    }

}

void **transpose**(float *a*[][MAX], int *row*, int *column*, float *b*[][MAX])

{

    for (int i = 0; i < *row*; i++)

        for (int j = 0; j < *column*; j++)

        {

*b*[i][j] = *a*[j][i];

        }

}

void **multiple**(float *a*[][MAX], float *b*[][MAX], int *row\_1*, int *column\_1*, int *row\_2*, int *column\_2*, float *result*[][MAX])

{

    if (*column\_1* != *row\_2*)

        return;

    for (int i = 0; i < *row\_1*; i++)

    {

        for (int j = 0; j < *column\_2*; j++)

        {

*result*[i][j] = 0;

            for (int k = 0; k < *row\_2*; k++)

            {

*result*[i][j] += *a*[i][k] \* *b*[k][j];

            }

        }

    }

}

void **printMatrix**(float *matrix*[][MAX], int *n*, const string &*name*)

{

    cout << "Matrix " << *name* << ":\n";

    for (int i = 0; i < *n*; i++)

    {

        for (int j = 0; j < *n*; j++)

        {

            cout << **left** << **setw**(10) << *matrix*[i][j];

        }

        cout << **endl**;

    }

    cout << **endl**;

}

void **LDLT\_Decomposition**(float *matrix*[][MAX], float *output1*[][MAX], float *output2*[][MAX], float *output3*[][MAX], int *n*)

{

**Identify**(*output1*, *n*);

**Identify**(*output3*, *n*);

    for (int i = 0; i < *n*; i++)

    {

        for (int j = 0; j <= i; j++)

        {

            float sum = 0;

            if (j == i)

            {

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

                    sum += **pow**(*output1*[j][k], 2) \* *output2*[k][k];

*output2*[j][j] = *matrix*[j][j] - sum;

            }

            else

            {

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

                    sum += (*output1*[i][k] \* *output1*[j][k] \* *output2*[k][k]);

*output1*[i][j] = (*matrix*[i][j] - sum) / *output2*[j][j];

            }

        }

    }

**transpose**(*output1*, *n*, *n*, *output3*);

**printMatrix**(*output1*, *n*, "L");

**printMatrix**(*output2*, *n*, "D");

**printMatrix**(*output3*, *n*, "LT");

}

int **main**()

{

    int n = 3;

    float matrix[MAX][MAX] = {{-10, 4, 7}, {4, -5, 8}, {7, 8, -9}};

    float L1[MAX][MAX], L2[MAX][MAX], L3[MAX][MAX];

    cout << "LDLT Decomposition: \n";

**LDLT\_Decomposition**(matrix, L1, L2, L3, n);

    return 0;

}

Kết quả:

A screenshot of a computer

Description automatically generated

2.b:

#include <iostream>

#include <string>

#include <cmath>

#include <iomanip>

#include <fstream>

using namespace std;

#define MAX 100

void **Solve**(double *a*, double *b*, double *c*, double *d*, double *output*[]) {

*b* /= *a*;

*c* /= *a*;

*d* /= *a*;

    double disc, q, r, dum1, s, t, term1, r13;

    q = (3.0 \* *c* - (*b* \* *b*)) / 9.0;

    r = -(27.0 \* *d*) + *b* \* (9.0 \* *c* - 2.0 \* (*b* \* *b*));

    r /= 54.0;

    disc = q \* q \* q + r \* r;

    term1 = (*b* / 3.0);

    double x1\_real, x2\_real, x3\_real;

    double x2\_imag, x3\_imag;

    string x2\_imag\_s, x3\_imag\_s;

    if (disc > 0) { *// Một nghiệm thực, hai nghiệm phức*

        s = r + **sqrt**(disc);

        s = s < 0 ? -**cbrt**(-s) : **cbrt**(s);

        t = r - **sqrt**(disc);

        t = t < 0 ? -**cbrt**(-t) : **cbrt**(t);

        x1\_real = -term1 + s + t;

        term1 += (s + t) / 2.0;

        x3\_real = x2\_real = -term1;

        term1 = **sqrt**(3.0) \* (-t + s) / 2;

        x2\_imag = term1;

        x3\_imag = -term1;

        x2\_imag\_s = " + " + **to\_string**(x2\_imag) + "i";

        x3\_imag\_s = " - " + **to\_string**(x2\_imag) + "i";

    } else if (disc == 0) { *// Các nghiệm thực, ít nhất hai nghiệm bằng nhau*

        r13 = r < 0 ? -**cbrt**(-r) : **cbrt**(r);

        x1\_real = -term1 + 2.0 \* r13;

        x3\_real = x2\_real = -(r13 + term1);

    } else { *// Ba nghiệm thực không bằng nhau*

        q = -q;

        dum1 = q \* q \* q;

        dum1 = **acos**(r / **sqrt**(dum1));

        r13 = 2.0 \* **sqrt**(q);

        x1\_real = -term1 + r13 \* **cos**(dum1 / 3.0);

        x2\_real = -term1 + r13 \* **cos**((dum1 + 2.0 \* M\_PI) / 3.0);

        x3\_real = -term1 + r13 \* **cos**((dum1 + 4.0 \* M\_PI) / 3.0);

    }

*output*[0] = x1\_real;

*output*[1] = x2\_real;

*output*[2] = x3\_real;

}

void **sort**(double *arr*[], int *n*) {

    for (int i = 0; i < *n* - 1; i++) {

        for (int j = 0; j < *n* - i - 1; j++) {

            if (*arr*[j] < *arr*[j + 1]) {

**swap**(*arr*[j], *arr*[j + 1]);

            }

        }

    }

}

void **Identify**(double *a*[][MAX], int *n*) {

    for (int i = 0; i < *n*; i++) {

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

*a*[i][j] = (i == j) ? 1.0 : 0.0;

        }

    }

}

void **Zerotify**(double *a*[][MAX], int *n*) {

    for (int i = 0; i < *n*; i++) {

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

*a*[i][j] = 0.0;

        }

    }

}

void **Assign**(double *a*[][MAX], double *b*[][MAX], int *n*) {

    for (int i = 0; i < *n*; i++) {

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

*a*[i][j] = *b*[i][j];

        }

    }

}

void **print**(double *a*[][MAX], int *n*) {

    for (int i = 0; i < *n*; i++) {

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

            cout << **setprecision**(5) << *a*[i][j] << "\t";

        }

        cout << "\n";

    }

}

void **multiple**(double *a*[][MAX], double *b*[][MAX], int *n*, double *result*[][MAX]) {

    for (int i = 0; i < *n*; i++) {

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

*result*[i][j] = 0;

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

*result*[i][j] += *a*[i][k] \* *b*[k][j];

            }

        }

    }

}

void **FormMatrix**(double *numb*, double *M*[][MAX], double *output*[][MAX], int *row*, int *column*) {

    double temp[MAX][MAX], temp1[MAX][MAX];

**Zerotify**(temp, *row*);

**Zerotify**(temp1, *row*);

    for (int i = 0; i < *row*; i++) {

        temp[i][0] = **pow**(*numb*, (double)((*row* - 1) - i));

    }

**multiple**(*M*, temp, *row*, temp1);

    for (int i = 0; i < *row*; i++) {

*output*[i][*column*] = temp1[i][0];

    }

}

void **Eigendecomposite**(double *a*[][MAX], int *n*) {

    double M[MAX][MAX], M1[MAX][MAX], temp[MAX][MAX], C[MAX][MAX], P[MAX][MAX];

**Identify**(C, *n*);

    for (int k = *n* - 2; k >= 0; k--) {

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

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

                if (i != k) {

                    M[i][j] = (i == j) ? 1 : 0;

                    M1[i][j] = (i == j) ? 1 : 0;

                } else {

                    M1[i][j] = *a*[k + 1][j];

                    M[i][j] = (j == k) ? 1 / *a*[k + 1][k] : -*a*[k + 1][j] / *a*[k + 1][k];

                }

            }

        }

**multiple**(*a*, M, *n*, temp);

**multiple**(M1, temp, *n*, *a*);

**multiple**(C, M, *n*, temp);

**Assign**(C, temp, *n*);

    }

    double output[MAX];

**Solve**(1, -*a*[0][0], -*a*[0][1], -*a*[0][2], output);

**sort**(output, *n*);

    cout << "\nEigenvalues (Giá trị riêng):\n";

    for (int i = 0; i < *n*; i++) {

        cout << "λ" << i + 1 << " = " << **setprecision**(5) << output[i] << **endl**;

    }

    for (int i = 0; i < *n*; i++) {

**FormMatrix**(output[i], C, P, *n*, i);

    }

    cout << "\nEigenvectors (Vector riêng):\n";

**print**(P, *n*);

}

int **main**() {

    double matrix[MAX][MAX] = {{-10, 4, 7}, {4, -5, 8}, {7, 8, -9}};

    int n = 3;

**Eigendecomposite**(matrix, n);

    return 0;

}

Kết quả:

A screenshot of a computer

Description automatically generated

Bài 3:

#include <bits/stdc++.h>

#include <Eigen/Dense>

using namespace std;

using namespace Eigen;

*//g++ -I D:/BACH\_KHOA\_LAME/nam3/ToanCNTT/eigen-3.4.0/eigen-3.4.0 SVD.cpp -o SVD*

MatrixXf **normalize**(const MatrixXf& *v*) {

    float norm = 0.0;

    for (int i = 0; i < *v*.rows(); ++i) {

        for (int j = 0; j < *v*.cols(); ++j) {

            norm += *v*(i,j) \* *v*(i,j);

        }

    }

    norm = **sqrt**(norm);

    MatrixXf result = *v*;

    if (norm > 1e-10) {

        for (int i = 0; i < *v*.rows(); ++i) {

            for (int j = 0; j < *v*.cols(); ++j) {

                result(i,j) /= norm;

            }

        }

    }

    return result;

}

MatrixXf **transpose**(const MatrixXf& *matrix*) {

    MatrixXf result(*matrix*.cols(), *matrix*.rows());

    for (int i = 0; i < *matrix*.rows(); ++i) {

        for (int j = 0; j < *matrix*.cols(); ++j) {

            result(j,i) = *matrix*(i,j);

        }

    }

    return result;

}

MatrixXf **Multiple**(MatrixXf *A*, int *A\_row*, int *A\_col*, MatrixXf *B*,int *B\_row*, int *B\_col*)

{

     MatrixXf result(*A\_row*, *B\_col*);

     result.setZero();

     for (int i =0 ; i< *A\_row*; i++)

     {

        for (int j = 0; j < *B\_col*; j++)

        {

            for (int k = 0; k < *A\_col*; k++)

            {

                result(i,j) += *A*(i,k)\* *B*(k,j);

            }

        }

     }

     return result;

}

MatrixXf **Multiple\_SingleNumber**(MatrixXf *A*, int *A\_row*, int *A\_col*, double *a*)

{

     MatrixXf result(*A\_row*, *A\_col*);

     for (int i =0 ; i< *A\_row*; i++)

     {

        for (int j = 0; j < *A\_col*; j++)

        {

            result(i, j) = *A*(i, j) \* *a*;

        }

     }

     return result;

}

MatrixXf **MatrixDacTrung**(MatrixXf *A*, int *row*, int *column*)

{

     MatrixXf At = **transpose**(*A*);

     MatrixXf mtdt = **Multiple**(At, *column*, *row*, *A*, *row*, *column*);

     return mtdt;

}

MatrixXf **vtCal**(MatrixXf *A*, int *row*, int *column*)

{

    MatrixXf mtdt = **MatrixDacTrung**(*A*, *row*, *column*);

    SelfAdjointEigenSolver<MatrixXf> eigensolver(mtdt);

    if (eigensolver.info() != Success) **abort**();

    MatrixXf eigenvectors = eigensolver.eigenvectors().rowwise().reverse();

    for (int i = 0; i < *column*; ++i) {

        eigenvectors.col(i) = **normalize**(eigenvectors.col(i));

        if (eigenvectors.col(i)(0) < 0) {

            eigenvectors.col(i) \*= -1;

        }

    }

    return eigenvectors;

}

MatrixXf **sigmaCal**(MatrixXf *A*, int *row*, int *column*)

{

    MatrixXf mtdt = **MatrixDacTrung**(*A*, *row*, *column*);

    SelfAdjointEigenSolver<MatrixXf> eigensolver(mtdt);

    if (eigensolver.info() != Success) **abort**();

    VectorXf eigenvalues = eigensolver.eigenvalues().reverse();

    MatrixXf sigma = MatrixXf::Zero(*row*, *column*);

    for (int i = 0; i < std::**min**(*row*, *column*); i++)

    {

        sigma(i,i) = **sqrt**(**fabs**(eigenvalues(i)));

    }

    return sigma;

}

MatrixXf **GetCol**(MatrixXf *A*, int *row*, int *column*, int *selected\_column*)

{

     MatrixXf Result(*row*, 1);

     for (int i = 0; i < *row*; i++)

     {

          Result(i,0) = *A*(i, *selected\_column*);

     }

     return Result;

}

MatrixXf **uCal**(MatrixXf *A*, int *row*, int *column*) {

    MatrixXf U(*row*, *row*);

    MatrixXf Sigma = **sigmaCal**(*A*, *row*, *column*);

    MatrixXf V = **vtCal**(*A*, *row*, *column*);

    U.setZero();

    for (int i = 0; i < std::**min**(*row*, *column*); i++) {

        if (Sigma(i, i) > 1e-6)

        {

            MatrixXf temp\_V = V.col(i);

            MatrixXf temp = *A* \* temp\_V / Sigma(i, i);

            U.col(i) = temp;

        }

    }

    for (int i = 0; i < *row*; i++) {

        U.col(i) = **normalize**(U.col(i));

        for (int j = 0; j < i; j++) {

            float proj = U.col(j).dot(U.col(i));

            U.col(i) -= proj \* U.col(j);

        }

        U.col(i) = **normalize**(U.col(i));

    }

    return U;

}

struct SVDResult {

    MatrixXf U;

    MatrixXf Sigma;

    MatrixXf Vt;

};

SVDResult **computeSVD**(MatrixXf *A*, int *row*, int *column*)

{

    SVDResult result;

    result.U = **uCal**(*A*, *row*, *column*);

    result.Sigma = **sigmaCal**(*A*, *row*, *column*);

    result.Vt = **transpose**(**vtCal**(*A*, *row*, *column*));

    return result;

}

MatrixXf **roundMatrix**(const MatrixXf& *matrix*) {

    return (*matrix* \* 1000.0).unaryExpr([](float *x*) { return std::**round**(*x*); }) / 1000.0;

}

int **main**()

{

    MatrixXf A(3,3);

    A << 0,5,10,

         5,5,5,

         10,10,10;

*// SVD Decomposition*

    SVDResult svd = **computeSVD**(A, 3, 3);

    cout << "\nMatrix A:\n" << **roundMatrix**(A) << "\n\n";

    cout << "U:\n" << **roundMatrix**(svd.U) << "\n\n";

    cout << "Sigma:\n" << **roundMatrix**(svd.Sigma) << "\n\n";

    cout << "Vt:\n" << **roundMatrix**(svd.Vt) << "\n\n";

*// Verify the decomposition*

    MatrixXf reconstructed = svd.U \* svd.Sigma \* svd.Vt;

    cout << "Reconstructed A:\n" << **roundMatrix**(reconstructed) << "\n\n";

    cout << "Reconstruction error: " << **round**((A - reconstructed).norm() \* 1000.0) / 1000.0 << "\n";

    return 0;

}

Kết quả:

A screenshot of a computer

Description automatically generated