**Họ và tên: Nguyễn Hoàng Tuấn Anh**

**MSSV: 102220219**

**Đề: 1**

Bài 1: Source code:

#include <iostream>

#include <cmath>

using namespace std;

const int MAX\_N = 500;

void **sieveOfEratosthenes**(int *n*, bool *isPrime*[]) {

    for (int i = 0; i <= *n*; i++) {

*isPrime*[i] = true;

    }

*isPrime*[0] = *isPrime*[1] = false;

    for (int i = 2; i \* i <= *n*; i++) {

        if (*isPrime*[i]) {

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

*isPrime*[j] = false;

            }

        }

    }

}

int **countPrimesInRange**(int *m*, int *n*, const bool *isPrime*[]) {

    int count = 0;

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

        if (*isPrime*[i]) count++;

    }

    return count;

}

int **findClosestPrime**(int *k*, int *m*, int *n*, const bool *isPrime*[]) {

    int closest = *m*;

    int minDiff = **abs**(*k* - *m*);

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

        if (*isPrime*[i]) {

            int diff = **abs**(*k* - i);

            if (diff < minDiff) {

                minDiff = diff;

                closest = i;

            }

        }

    }

    return closest;

}

int **main**() {

    int m = 300, n = 500, k = 436;

    bool isPrime[MAX\_N + 1];

**sieveOfEratosthenes**(n, isPrime);

    int primeCount = **countPrimesInRange**(m, n, isPrime);

    cout << "So luong so nguyen to trong khoang " << m << "-" << n << ": " << primeCount << **endl**;

    cout << "Cac so nguyen to trong khoang " << m << "-" << n << ": ";

    for (int i = m; i <= n; i++) {

        if (isPrime[i]) cout << i << " ";

    }

    cout << **endl**;

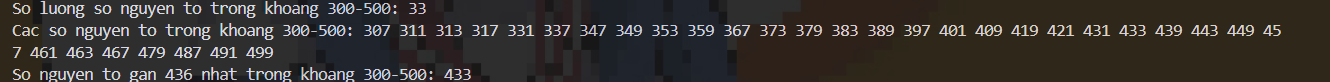
    int closestPrime = **findClosestPrime**(k, m, n, isPrime);

    cout << "So nguyen to gan " << k << " nhat trong khoang " << m << "-" << n << ": " << closestPrime << **endl**;

    return 0;

}

Kết quả:



Bài 2: Source code:

#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; *// Remove the transpose here*

}

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) *// Avoid division by zero*

        {

            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;

}

void **Cholesky\_Decomposition**(const MatrixXf& *matrix*, MatrixXf& *L*, MatrixXf& *Lt*, int *n*) {

*L* = MatrixXf::Zero(*n*, *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**(*L*(j, k), 2);

                }

*L*(j, j) = **sqrt**(*matrix*(j, j) - sum);

            } else {

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

                    sum += *L*(i, k) \* *L*(j, k);

                }

*L*(i, j) = (*matrix*(i, j) - sum) / *L*(j, j);

            }

        }

    }

*Lt* = **transpose**(*L*);

}

void **LDLT\_Decomposition**(const MatrixXf& *matrix*, MatrixXf& *L*, MatrixXf& *D*, MatrixXf& *Lt*, int *n*) {

*L* = MatrixXf::Identity(*n*, *n*);

*D* = MatrixXf::Zero(*n*, *n*);

    for (int i = 0; i < *n*; i++) {

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

            float sum = 0;

            if (i == j) {

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

                    sum += *L*(j, k) \* *L*(j, k) \* *D*(k, k);

                }

*D*(j, j) = *matrix*(j, j) - sum;

            } else {

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

                    sum += *L*(i, k) \* *L*(j, k) \* *D*(k, k);

                }

*L*(i, j) = (*matrix*(i, j) - sum) / *D*(j, j);

            }

        }

    }

*Lt* = **transpose**(*L*);

}

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 << 1, 2, 3,

         1, 2, 4,

         1, 2, 5;

*// Cholesky Decomposition*

    MatrixXf L\_cholesky(3, 3), Lt\_cholesky(3, 3);

**Cholesky\_Decomposition**(A, L\_cholesky, Lt\_cholesky, 3);

    cout << "\nCholesky Decomposition - L:\n" << **roundMatrix**(L\_cholesky) << "\n";

    cout << "Cholesky Decomposition - L^T:\n" << **roundMatrix**(Lt\_cholesky) << "\n";

*// LDLT Decomposition*

    MatrixXf L\_ldlt(3, 3), D\_ldlt(3, 3), Lt\_ldlt(3, 3);

**LDLT\_Decomposition**(A, L\_ldlt, D\_ldlt, Lt\_ldlt, 3);

    cout << "\nLDLT Decomposition - L:\n" << **roundMatrix**(L\_ldlt) << "\n";

    cout << "LDLT Decomposition - D:\n" << **roundMatrix**(D\_ldlt) << "\n";

    cout << "LDLT Decomposition - L^T:\n" << **roundMatrix**(Lt\_ldlt) << "\n";

*// 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 screen

Description automatically generated

Bài 3: Source code:

#include <iostream>

#include <cmath>

using namespace std;

const int MAX\_N = 2000;

void **sieveOfEratosthenes**(bool *isPrime*[]) {

    for (int i = 0; i <= MAX\_N; i++) {

*isPrime*[i] = true;

    }

*isPrime*[0] = *isPrime*[1] = false;

    for (int i = 2; i \* i <= MAX\_N; i++) {

        if (*isPrime*[i]) {

            for (int j = i \* i; j <= MAX\_N; j += i) {

*isPrime*[j] = false;

            }

        }

    }

}

int **findNearestPrime**(int *n*, const bool *isPrime*[]) {

    if (*isPrime*[*n*]) return *n*;

    int lower = *n* - 1, upper = *n* + 1;

    while (true) {

        if (lower >= 2 && *isPrime*[lower]) return lower;

        if (upper <= MAX\_N && *isPrime*[upper]) return upper;

        lower--;

        upper++;

    }

}

bool **isPerfectNumber**(int *num*) {

    if (*num* <= 1) return false;

    int sum = 1;

    for (int i = 2; i <= **sqrt**(*num*); i++) {

        if (*num* % i == 0) {

            sum += i;

            if (i != *num* / i) {

                sum += *num* / i;

            }

        }

    }

    return sum == *num*;

}

int **main**() {

    int n = 301;

    bool isPrime[MAX\_N + 1];

**sieveOfEratosthenes**(isPrime);

    int nearestPrime = **findNearestPrime**(n, isPrime);

    cout << "a.So nguyen to gan " << n << " nhat la: " << nearestPrime << **endl**;

    int M\_perfectCount = 0;

    int sum = 0;

    cout << "b. Cac so hoan hao trong M: ";

    for (int i = 2; i <= 2000; i++) {

        if (**isPerfectNumber**(i)) {

            cout << i << " ";

            M\_perfectCount++;

            sum += i;

        }

    }

    cout << **endl**;

    cout << "So phan tu hoan hao trong M: " << M\_perfectCount << **endl**;

    cout << "Tong cac so hoan hao: " << sum << **endl**;

    return 0;

}

Kết quả:

A screenshot of a computer screen

Description automatically generated