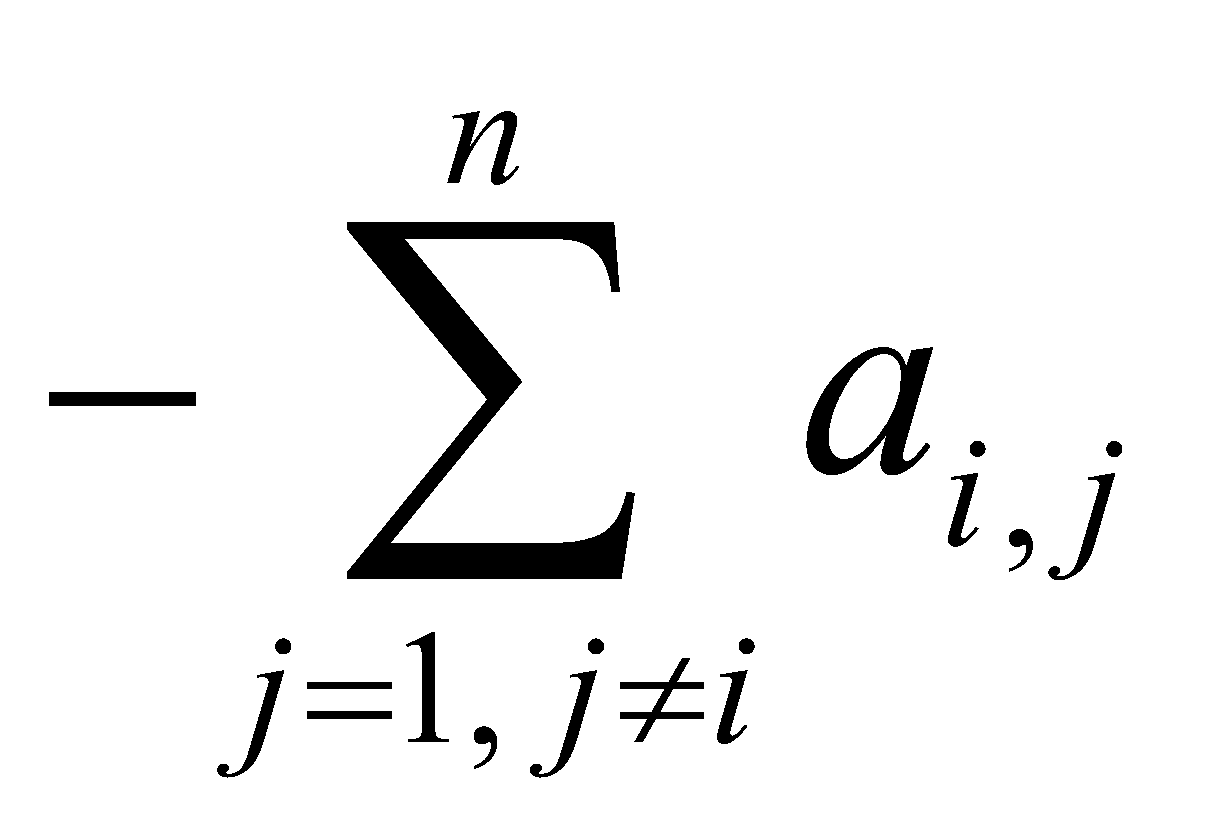
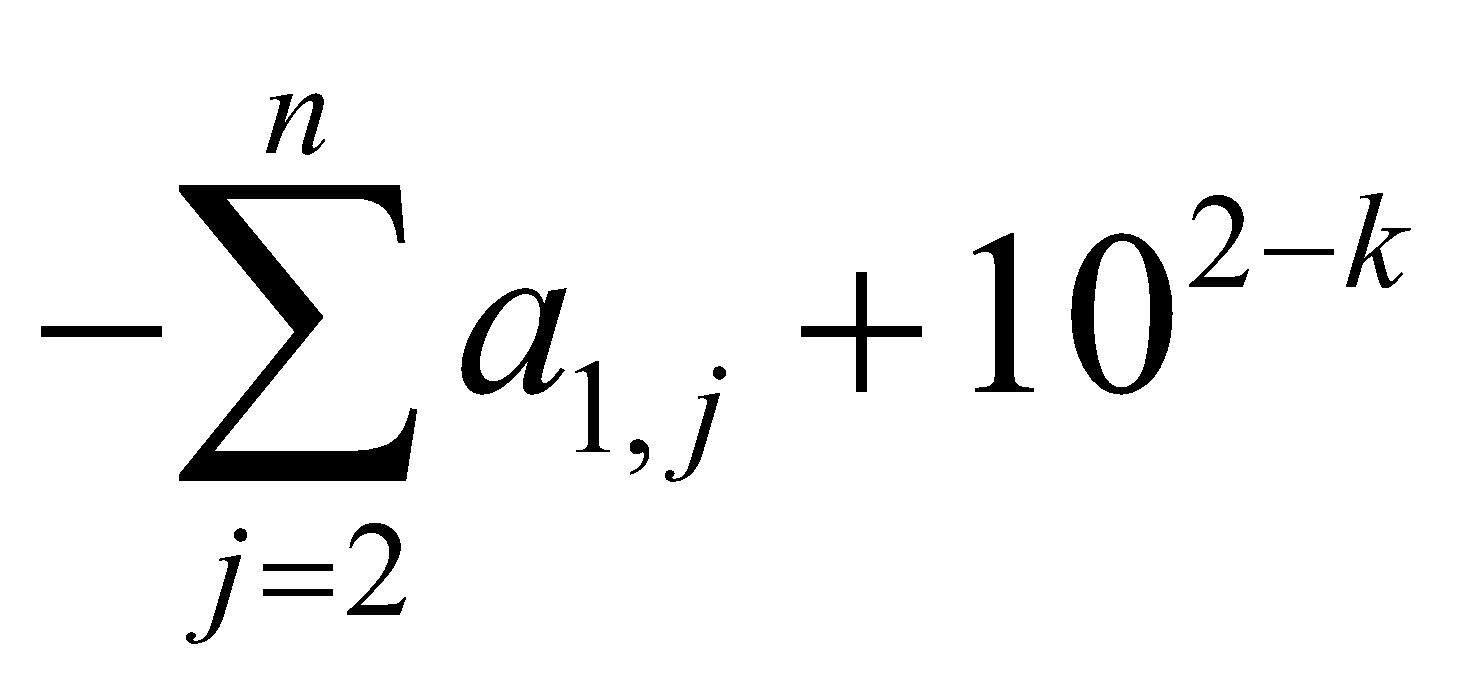
**Лабораторная работа №2 «Решение систем на основе разложения симметричных матриц»**Выполнил студент 3 курса 4 группы ФПМИ БГУ Видевич Александр

**Задача.**

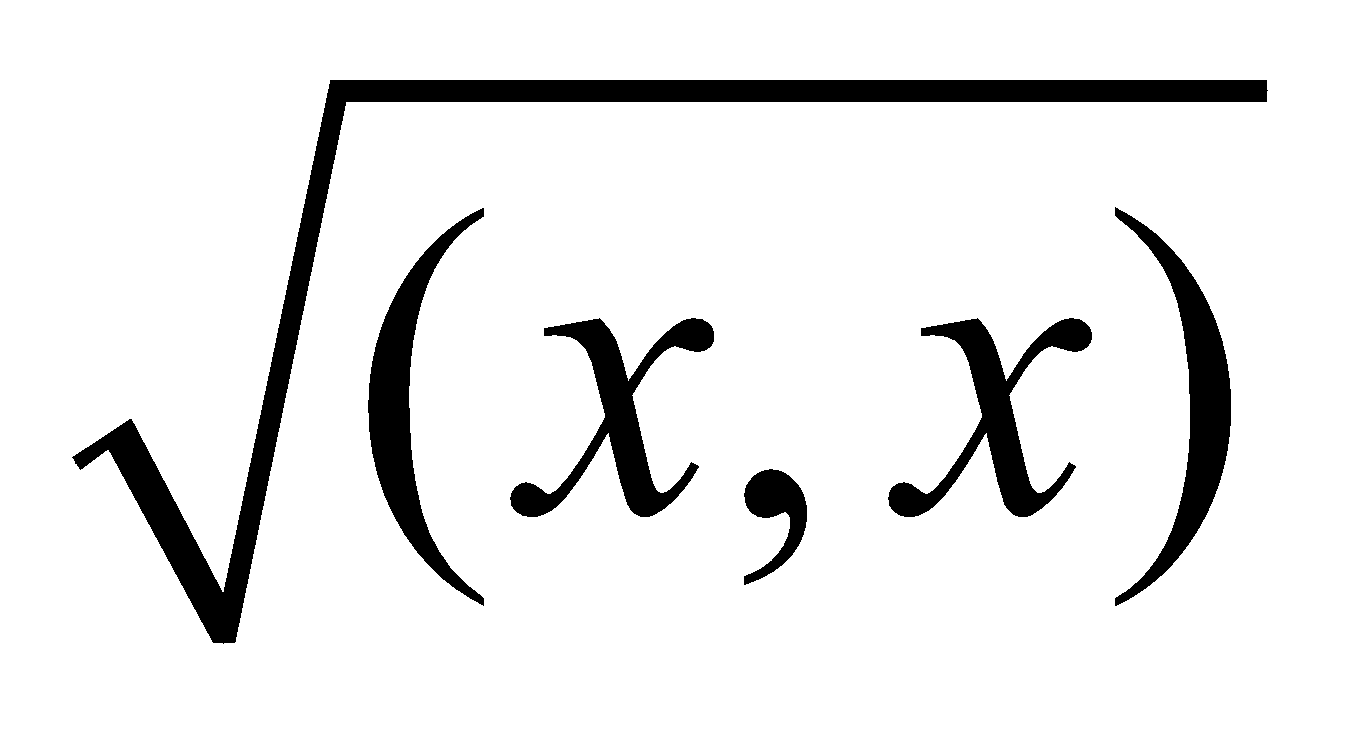
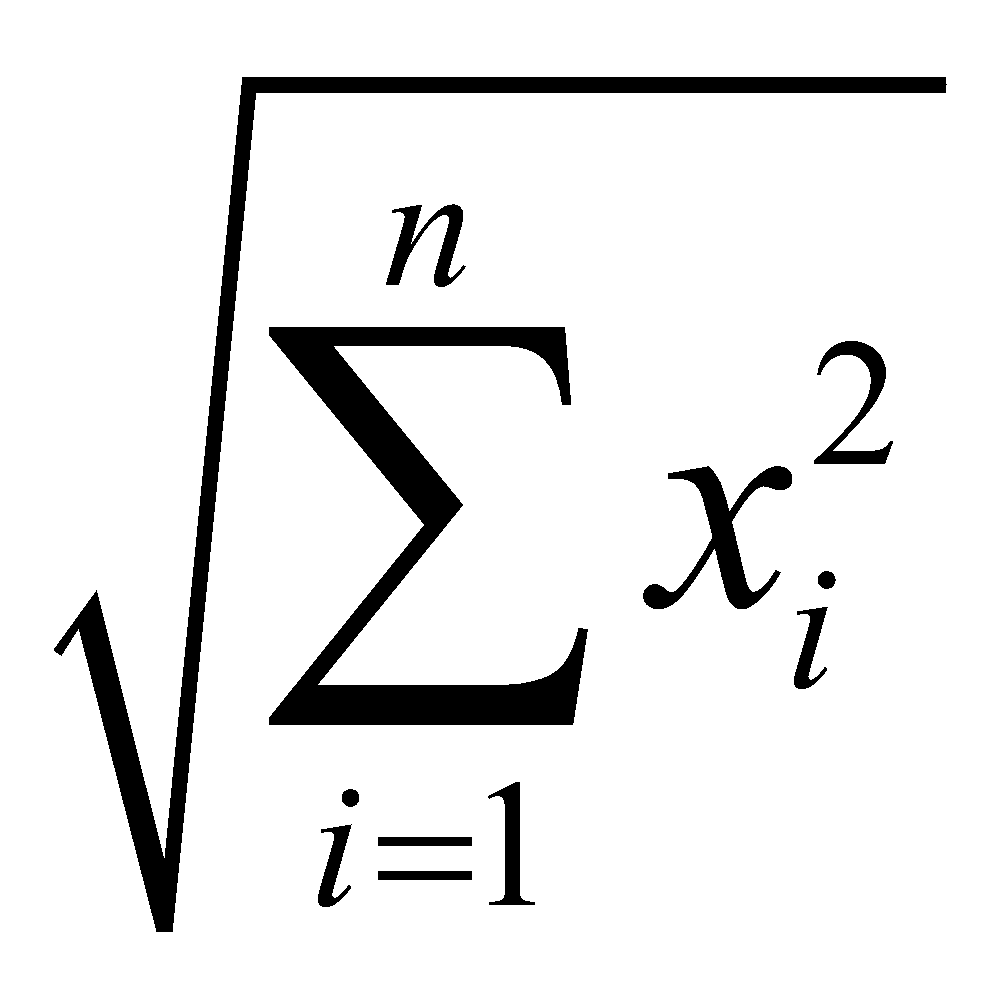
Разработать программу численного решения СЛАУ на основе LDLT-разложения. Сравнить время реализации рассматриваемого примера с временем реализации аналогичного примера в работе 1 «Метод Гаусса». Матрицу (порядка *n*) системы сформировать следующим образом:

* недиагональные элементы *ai,j*, *i<j*, выбираются случайным образом из диапазона от 0 до *–*1000; если *i>j*, то полагается *ai,j*=*aj,i*.
* *ai,i=**, 2≤i*≤*n*; *a*11*=**.*

Правую часть *b* задать умножением матрицы *A* на вектор *x=*(*m*, *m*+1, ... , *n*+*m*–1): *b=Ax*. Для вычислений выбрать параметры:

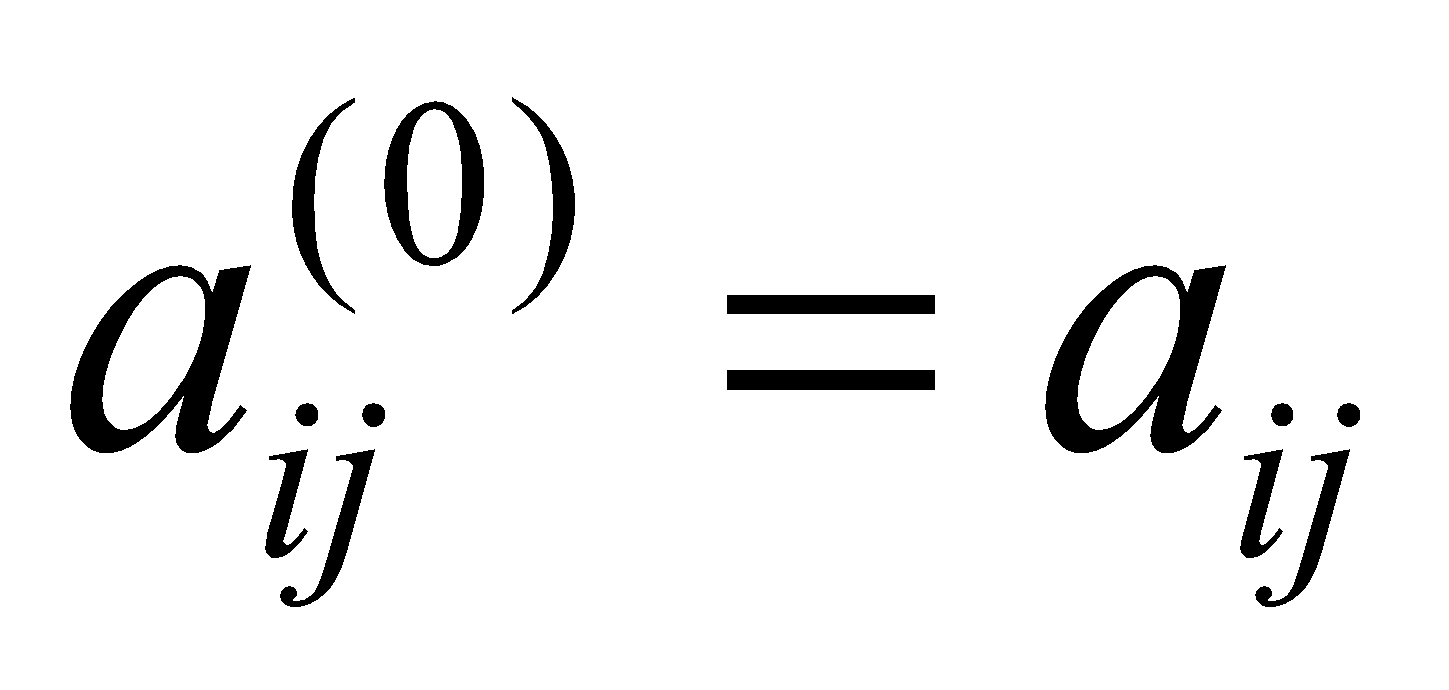
* *n* – одно из чисел в пределах от 1500 до 2000;
* *m* – номер в списке студенческой группы;

*k* – номер студенческой группы.

Для оценки погрешности вычислений использовалась евклидова (сферическая норма): ||*x*||2==.

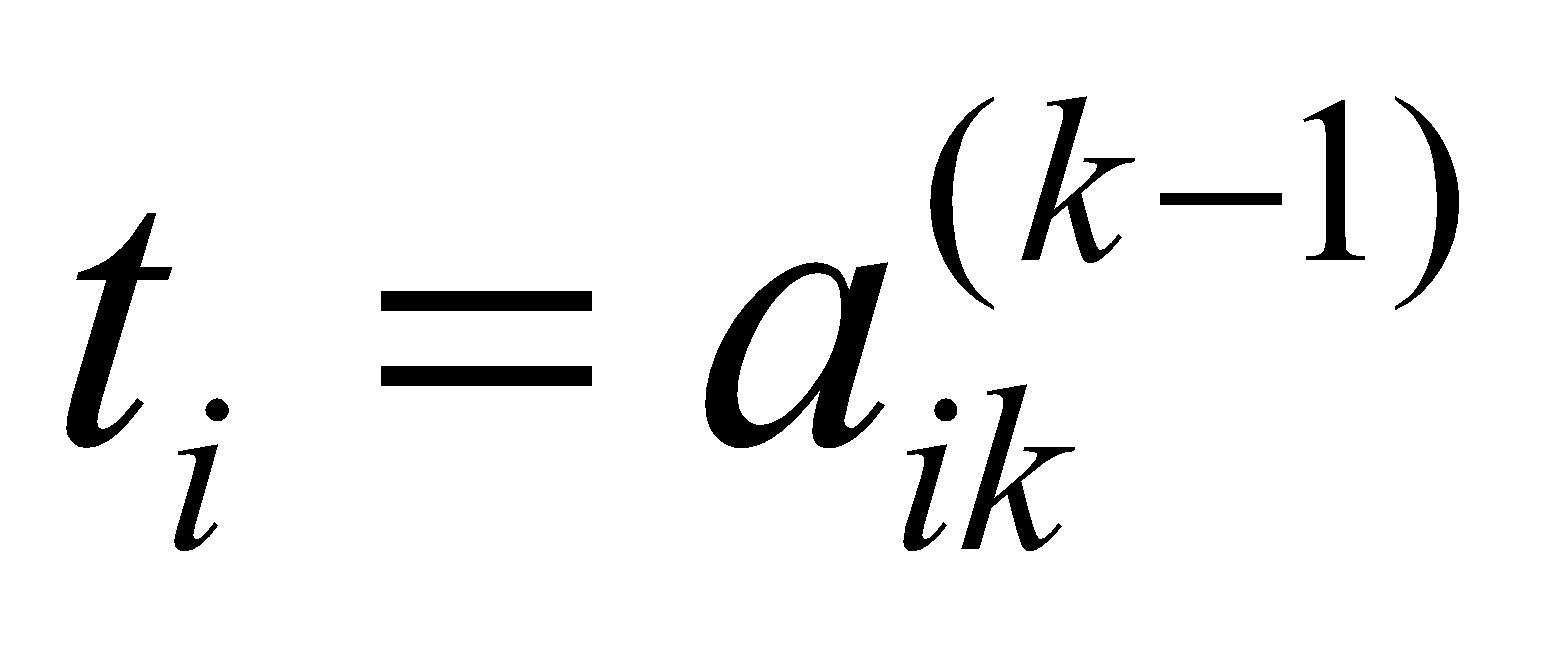
Относительная погрешность , где – точное решение (*m*, *m*+1, ..., *n*+*m*–1).

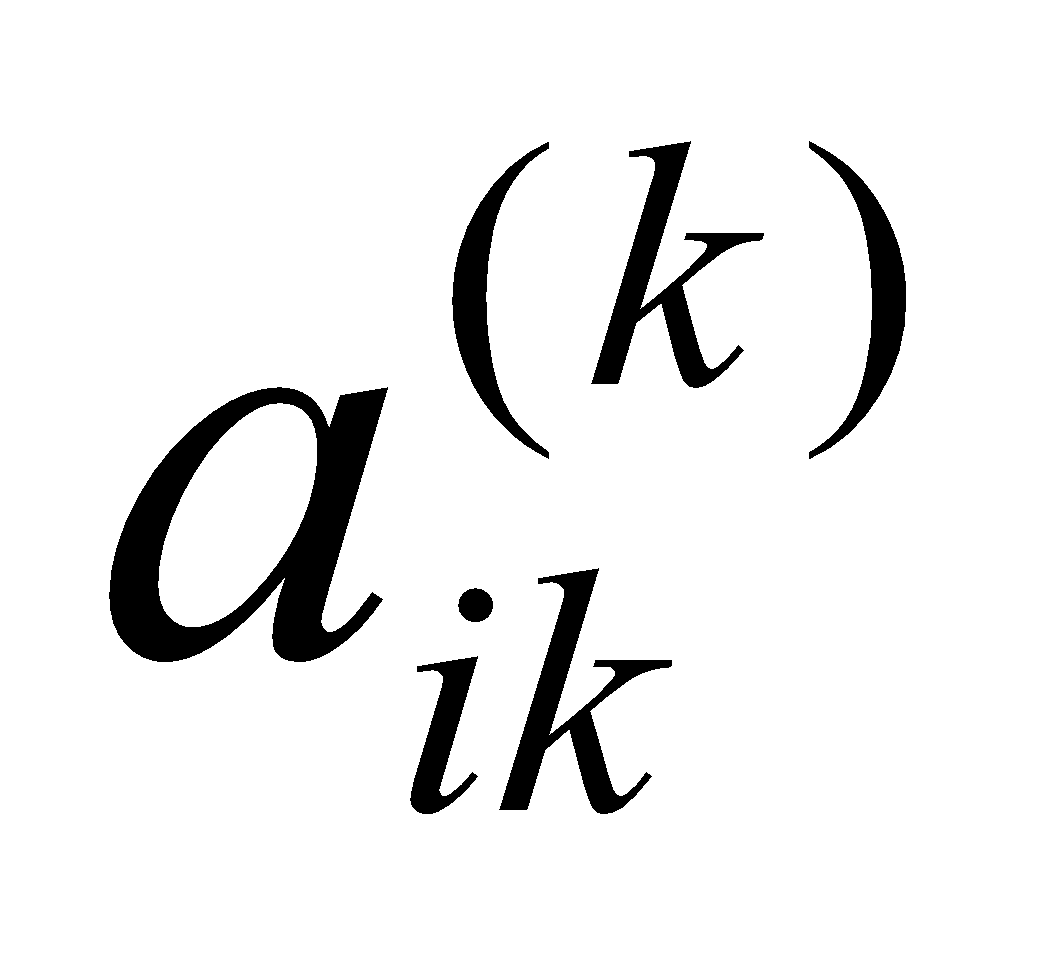
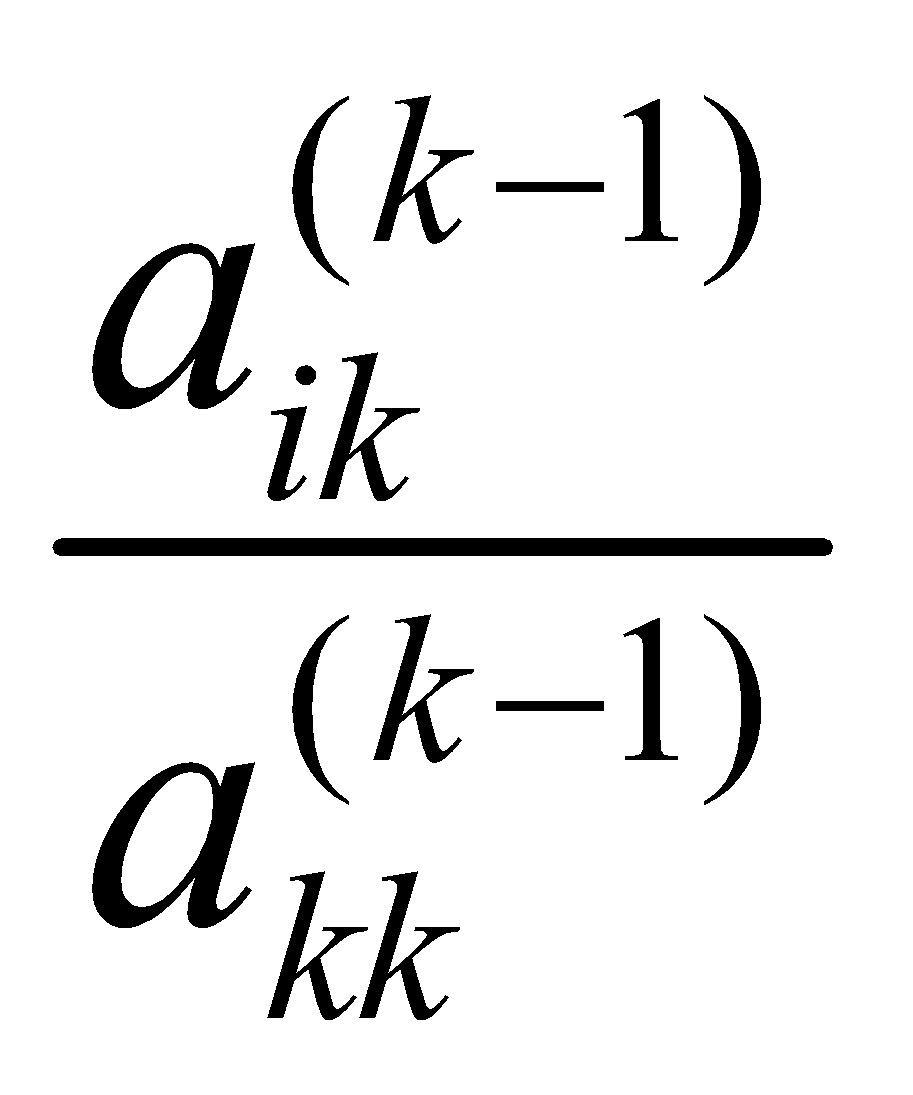
Разложение происходит по алгоритму:

*, i =*1*,*2*,… ,n, j =*1*,*2*,…,i*;

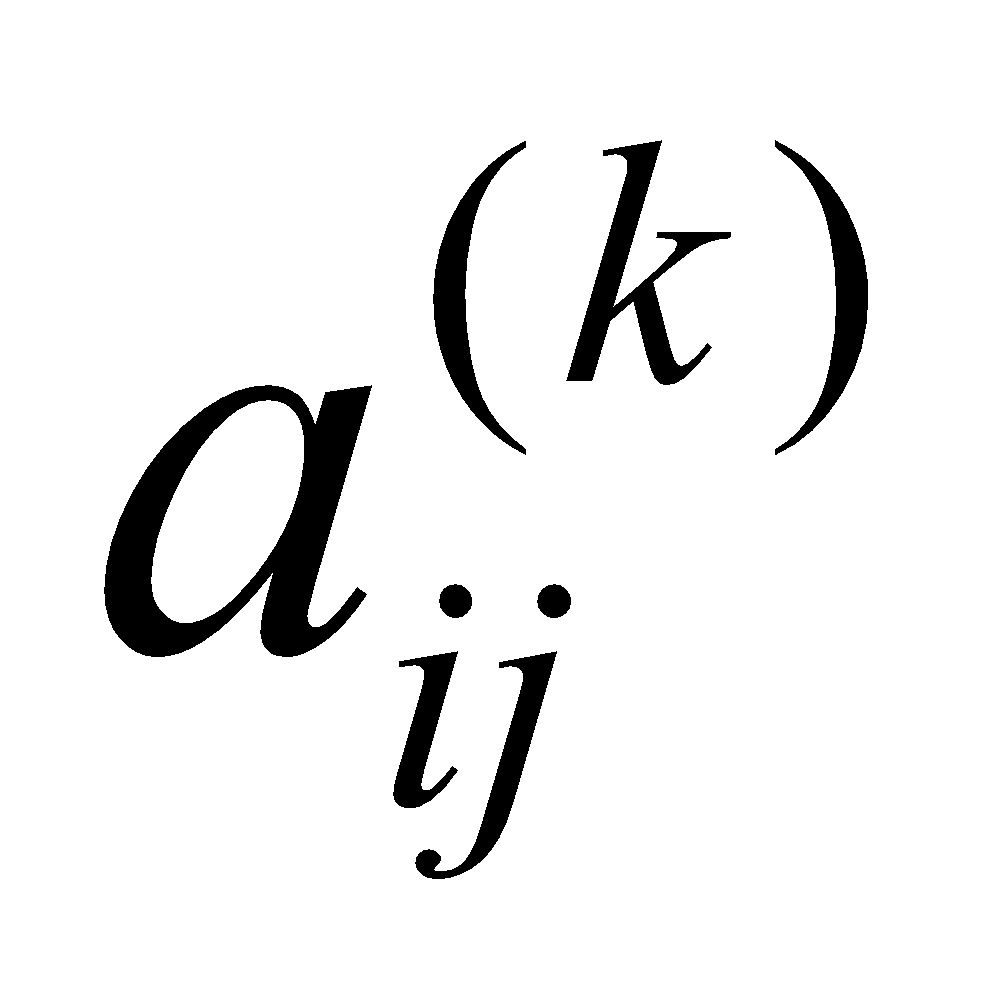
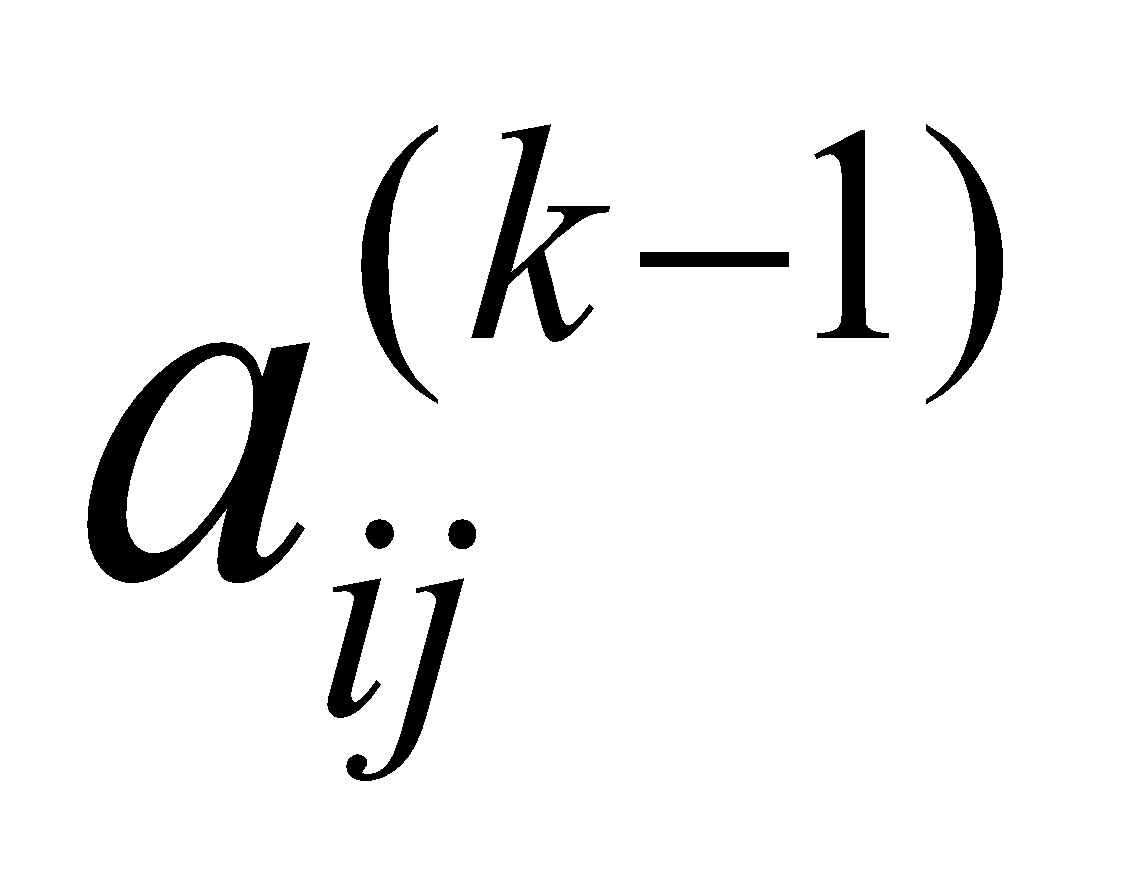
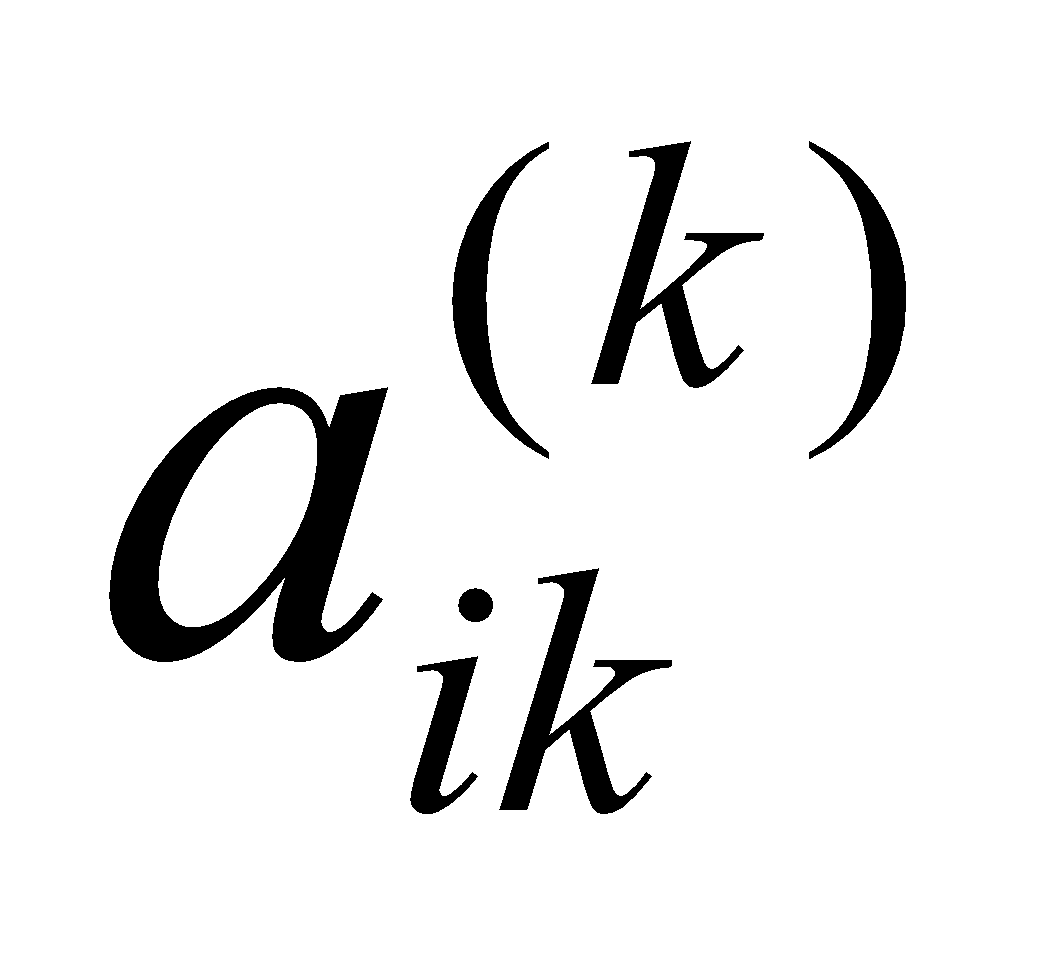
*k =* 1*,* 2*, … , n–*1:

*i = k+*1*, k+*2*, … , n*:

,

=,

*j = k+*1*, … , i*:

=–t*j.*

Решение уравнения состоит из последовательного решения системы уравнений:

Ly = b,

Dz = y,

LT x = z.

**Входные данные.**

n = 1800, m = 4, k = 4

**Листинг программы.**

*#include* <iostream>

*#include* <vector>

*#include* <stdlib.h>

*#include* <ctime>

*#include* <cmath>

*#include* <chrono>

*#include* <string>

int n;

*// функция генерации матрицы*

*#define* K *4*

std::*vector*<std::*vector*<double>> GenerateMatrix(*const* int size)

{

    std::*vector*<std::*vector*<double>> matrix(size, std::*vector*<double>());

*for* (int i = size - *1*; i >= *0*; --i)

    {

        double diagElement = *0*;

*for* (int j = size - *1*; j > i; --j)

        {

            diagElement += matrix[j][i];

        }

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

        {

*if* (i == j)

            {

                matrix[i].push\_back(*0.0f*);

*continue*;

            }

            matrix[i].push\_back((double)rand() / RAND\_MAX \* -*1000.0f*);

            diagElement += matrix[i][j];

        }

*if* (i == *0*)

        {

            matrix[i][i] = -diagElement + std::pow(*10*, *2* - K);

        }

*else*

        {

            matrix[i][i] = -diagElement;

        }

    }

*return* matrix;

}

std::*vector*<std::*vector*<double>> CreateMatrixForGauss(*const* std::*vector*<std::*vector*<double>> *&*matrix)

{

    std::*vector*<std::*vector*<double>> matrixForGauss(n, std::*vector*<double>(n));

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

    {

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

        {

*if* (i >= j)

            {

                matrixForGauss[i][j] = matrix[i][j];

            }

*else*

            {

                matrixForGauss[i][j] = matrix[j][i];

            }

        }

    }

*return* matrixForGauss;

}

*// фунция создания вектора x*

*#define* M *4*

std::*vector*<int> GetVector(*const* int size)

{

    std::*vector*<int> vector(size);

*for* (int i = *0*; i < size; ++i)

    {

        vector[i] = M + i;

    }

*return* vector;

}

*// функция умножения матрицы на вектор*

template <typename T>

std::*vector*<double> MatrixVectorMultiply(*const* std::*vector*<std::*vector*<double>> *&*matrix, *const* std::*vector*<T> *&*vector)

{

*const* int size = vector.size();

    std::*vector*<double> result(size);

*for* (int i = *0*; i < size; i++)

    {

        double sum = *0*;

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

        {

            sum += matrix[i >= j ? i : j][i >= j ? j : i] \* static\_cast<double>(vector[j]);

        }

        result[i] = sum;

    }

*return* result;

}

*// функция представления матрицы A в виде A=LDL^T*

void LdltRtRDecomposition(std::*vector*<std::*vector*<double>> *&*matrix)

{

    std::*vector*<double> t(n);

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

    {

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

        {

            t[i] = matrix[i][k];

            matrix[i][k] /= matrix[k][k];

*for* (int j = k + *1*; j <= i; ++j)

            {

                matrix[i][j] -= matrix[i][k] \* t[j];

            }

        }

    }

}

*// функция решения системы Ly=b*

std::*vector*<double> SolveLyEqB(*const* std::*vector*<std::*vector*<double>> *&*lMatrix, *const* std::*vector*<double> *&*bVector)

{

    std::*vector*<double> y(n);

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

    {

        double sum = *0.0*;

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

        {

            sum += lMatrix[i][j] \* y[j];

        }

        y[i] = bVector[i] - sum;

    }

*return* y;

}

*// функция решения системы Dz=y*

std::*vector*<double> SolveDzEqY(*const* std::*vector*<std::*vector*<double>> *&*DMatrix, *const* std::*vector*<double> *&*yVector)

{

    std::*vector*<double> z(n);

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

    {

        z[i] = yVector[i] / DMatrix[i][i];

    }

*return* z;

}

*// функция решения системы L^Tx=z*

std::*vector*<double> SolveLTxEqZ(*const* std::*vector*<std::*vector*<double>> *&*ltMatrix, *const* std::*vector*<double> *&*zVector)

{

    std::*vector*<double> x(n);

*for* (int i = n - *1*; i >= *0*; i--)

    {

        double sum = *0.0*;

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

        {

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

        }

        x[i] = zVector[i] - sum;

    }

*return* x;

}

*// функция решения СЛАУ на основе LDL^T разложения*

std::*vector*<double> SolveSystem(std::*vector*<std::*vector*<double>> *&*matrix, *const* std::*vector*<double> *&*vector)

{

    LdltRtRDecomposition(matrix);

*const* std::*vector*<double> y = SolveLyEqB(matrix, vector);

*const* std::*vector*<double> z = SolveDzEqY(matrix, y);

*const* std::*vector*<double> x = SolveLTxEqZ(matrix, z);

*return* x;

}

template <typename T>

void PrintVector(*const* std::*vector*<T> *&*vector, *const* int numberOfElements = *0*, *const* std::string *&*message = "")

{

*if* (message != "")

    {

        std::cout << message << ' ';

    }

*const* int border = numberOfElements != *0* ? numberOfElements : vector.size();

*for* (int i = *0*; i < border; ++i)

    {

        std::cout << vector[i] << ' ';

    }

    std::cout << '*\n*';

}

*// функция подсчета квадратичной (евклидовой нормы)*

template <typename T>

double CalculateEuclideanNorm(*const* std::*vector*<T> *&*vector)

{

    double sumOfSquares = *0.0*;

*// подсчет суммы квадратов координат вектора*

*for* (auto element : vector)

    {

        sumOfSquares += static\_cast<double>(element \* element);

    }

*// извелечение корня из суммы квадратов и возвращение результата*

*return* std::sqrt(sumOfSquares);

}

*// функция подсчета относительной погрешности*

double GetRelativeError(*const* std::*vector*<int> *&*originalVector, *const* std::*vector*<double> *&*calculatedVector)

{

    std::*vector*<double> temp(n);

*// подсчет вектора x-x\**

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

    {

        temp[i] = originalVector[i] - calculatedVector[i];

    }

*// подсчет норм и возвращение результата*

*return* CalculateEuclideanNorm(temp) / CalculateEuclideanNorm(originalVector);

}

*// функция шага прямого хода метода Гауса*

void MakeMove(std::*vector*<std::*vector*<double>> *&*matrix, std::*vector*<double> *&*vector, *const* int k)

{

*// обход всех нижестоящих уравнений системы*

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

    {

*// получение коэффициента lik*

        double lik = matrix[i][k] / matrix[k][k];

        matrix[i][k] = *0.0*;

*// получение новой системы*

*for* (int j = k + *1*; j < n; j++)

        {

            matrix[i][j] -= lik \* matrix[k][j];

        }

        vector[i] -= lik \* vector[k];

    }

}

*// функция осуществления обратного хода метода Гауса*

std::*vector*<double> GetGaussResult(*const* std::*vector*<std::*vector*<double>> *&*matrix, *const* std::*vector*<double> *&*vector)

{

*// создание вектора (x) для хранения результата*

    std::*vector*<double> solution(n);

*// получение элемента xn*

    solution[n - *1*] = vector[n - *1*] / matrix[n - *1*][n - *1*];

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

    {

*// получение элемента xi*

        double sum = *0.0*;

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

        {

            sum += matrix[i][j] \* solution[j];

        }

        solution[i] = (vector[i] - sum) / matrix[i][i];

    }

*// возвращение результата*

*return* solution;

}

std::*vector*<double> GaussWithoutSelectingLeadingElement(*const* std::*vector*<std::*vector*<double>> *&*matrix, *const* std::*vector*<double> *&*vector)

{

*// создание копий данных для предотвращения их изменения*

    std::*vector*<std::*vector*<double>> matrixCopy(matrix);

    std::*vector*<double> vectorCopy(vector);

*// проход по каждому уравнению системы*

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

    {

*// прямой ход метода Гауса для текущего шага*

        MakeMove(matrixCopy, vectorCopy, k);

    }

*// обратный ход метода Гауса и возвращение результата*

*return* GetGaussResult(matrixCopy, vectorCopy);

}

int main()

{

    std::ios\_base::sync\_with\_stdio(*false*);

    std::cin.tie(*nullptr*);

    std::srand(static\_cast<unsigned int>(std::time(*nullptr*)));

*// ввод данных*

    std::cin >> n;

*// генерация задачи*

    std::*vector*<std::*vector*<double>> matrix = GenerateMatrix(n);

    std::*vector*<std::*vector*<double>> matrixForGauss = CreateMatrixForGauss(matrix);

*const* std::*vector*<int> vector = GetVector(n);

*const* std::*vector*<double> b = MatrixVectorMultiply(matrix, vector);

*// решение СЛАУ с использованием LDL^T разложения*

    std::chrono::high\_resolution\_clock::time\_point start = std::chrono::high\_resolution\_clock::now();

*const* std::*vector*<double> x = SolveSystem(matrix, b);

    std::chrono::high\_resolution\_clock::time\_point end = std::chrono::high\_resolution\_clock::now();

*const* int time1 = std::chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

*// решение той же СЛАУ с помощью метода Гауса*

    start = std::chrono::high\_resolution\_clock::now();

*const* std::*vector*<double> x2 = GaussWithoutSelectingLeadingElement(matrixForGauss, b);

    end = std::chrono::high\_resolution\_clock::now();

*const* int time2 = std::chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

*// вывод результатов*

    PrintVector(x, *5*, "vector x calculated using LDL^T decomposition");

*// подсчет и вывод относительной погрешности*

    std::cout << "RelativeError: " << GetRelativeError(vector, x) << '*\n*';

*// вывод времени выполнения*

    std::cout << "Time (LDL^T decomposition): " << time1 << "ms";

    std::cout << "Time (Gauss): " << time2 << "ms";

*return* *0*;

}

**Выходные данные.**

vector x calculated using LDL^T decomposition 4.00133 5.00133 6.00133 7.00133 8.00133

RelativeError: 1.27343e-06

Time (LDL^T decomposition): 6811ms

Time (Gauss): 12611ms

**Выводы.**

Применение LDLT-разложения для работы с симметричными матрицами дает выигрыш во времени примерно в два раза.