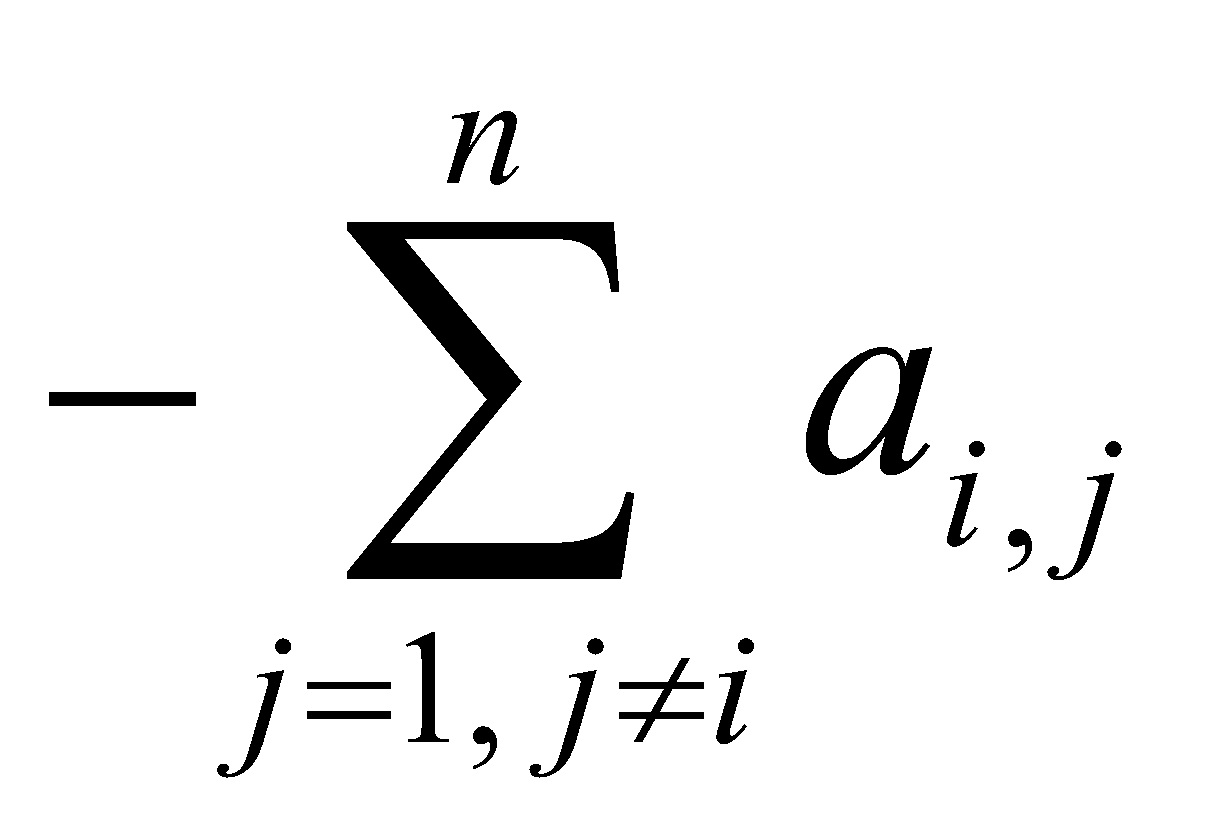
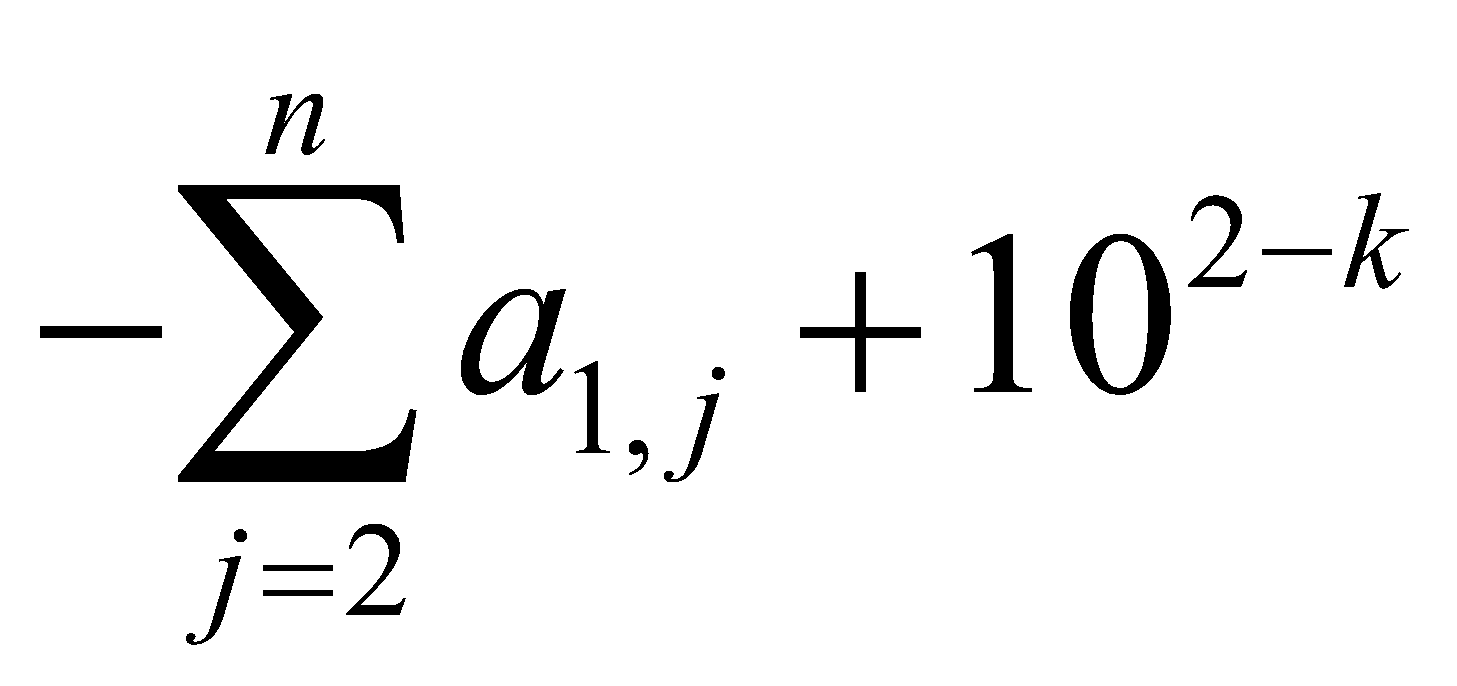
**Лабораторная работа 5 «Метод сопряженных градиентов»**

Выполнил студент 3 курса 4 группы ФПМИ БГУ Видевич Александр

**Задача.**

Разработать программу численного решения СЛАУ методом сопряженных градиентов. Результаты вычислительных экспериментов сравнить с аналогичными результатами, полученными с помощью разработанной ранее программы численного решения СЛАУ на основе LDLT-разложения.

Матрицу (порядка *n*) системы сформировать следующим образом:

* недиагональные элементы *ai,j*, *i<j*, выбираются случайным образом из диапазона от 0 до *–*100; 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*.

Случай 1. Программно реализовать псевдокод алгоритма CG для рассматриваемого примера (в качестве языка программирования выбрать C или C++). Взять *x*0*=*0; *lit=*50.

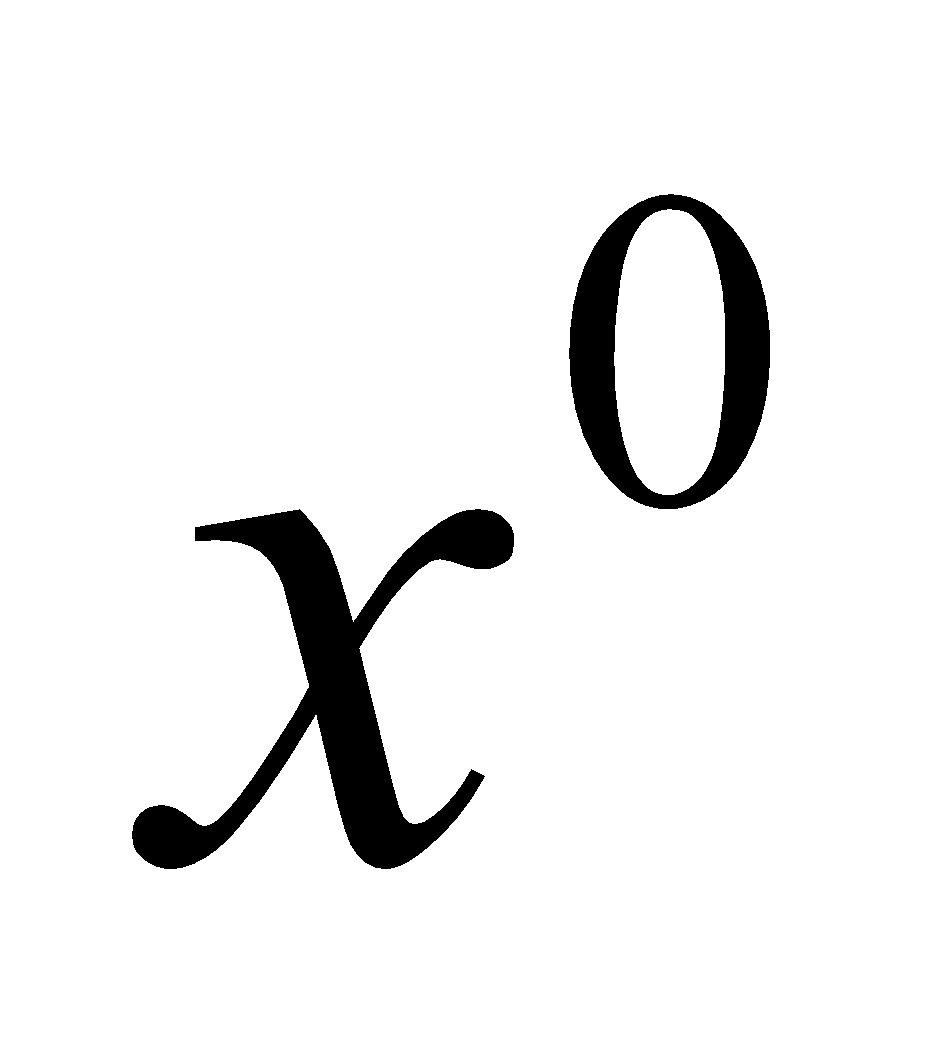
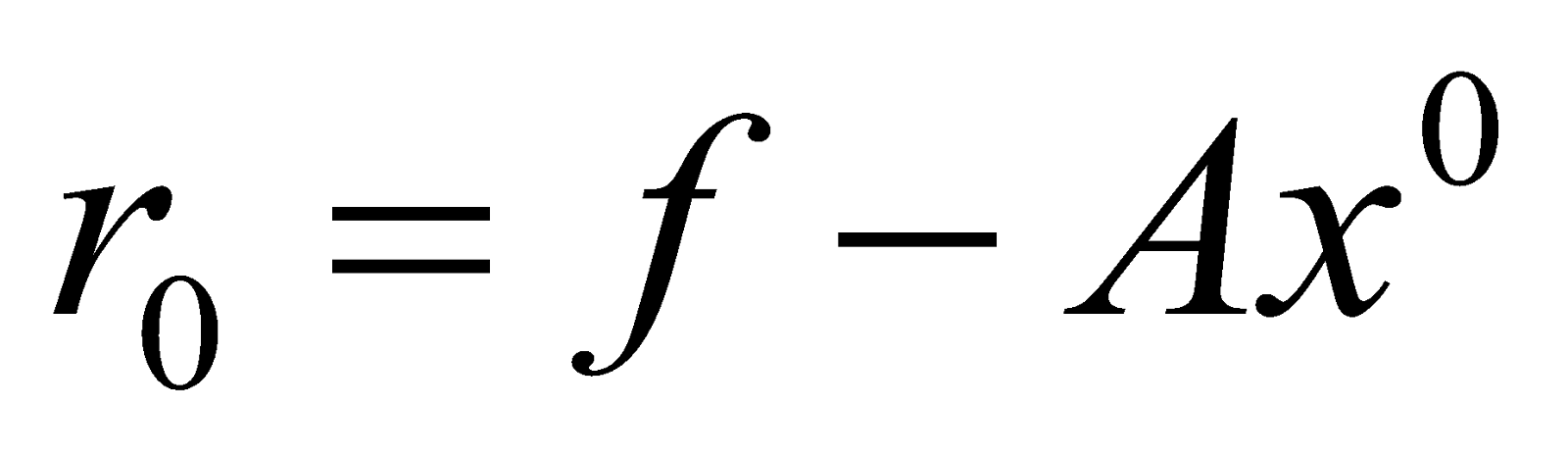
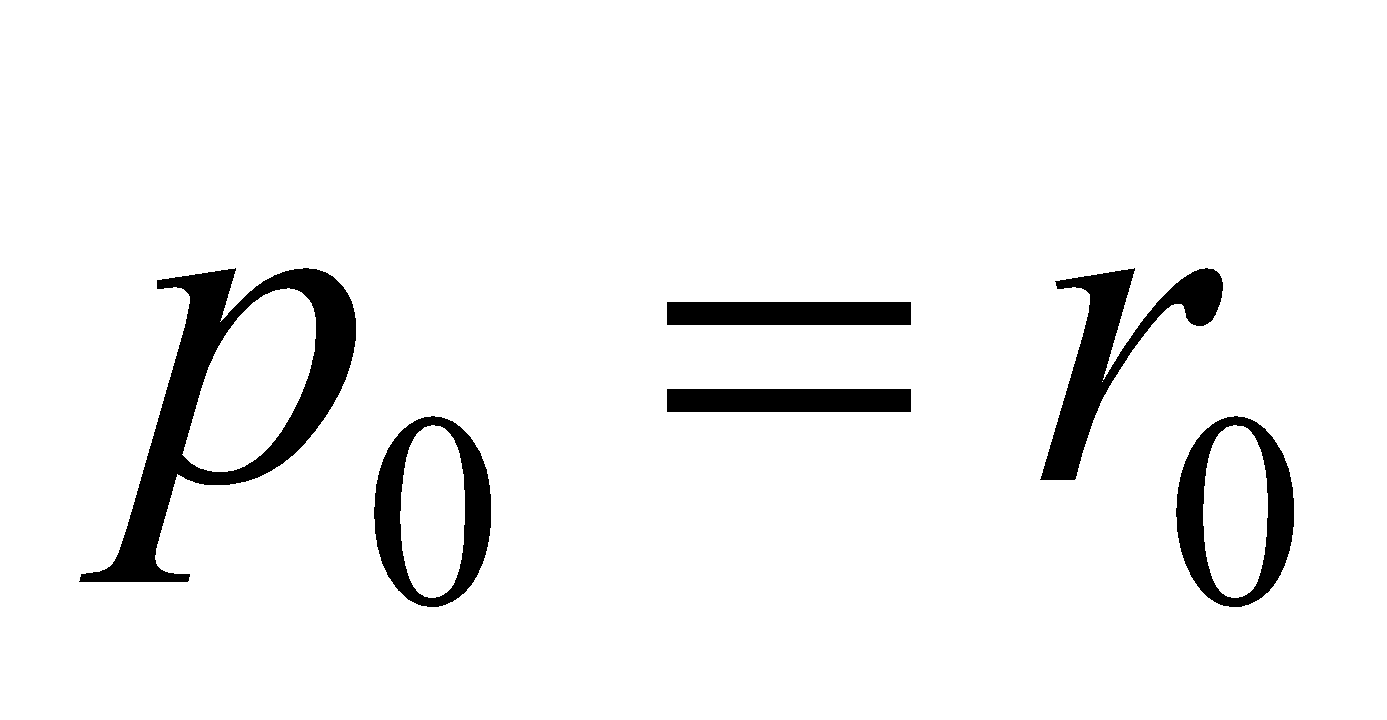
Случай 2. Получить приближение *x\** к точному решению с помощью разработанной ранее программы численного решения СЛАУ на основе LDLT-разложения

Для вычислений выбрать параметры:

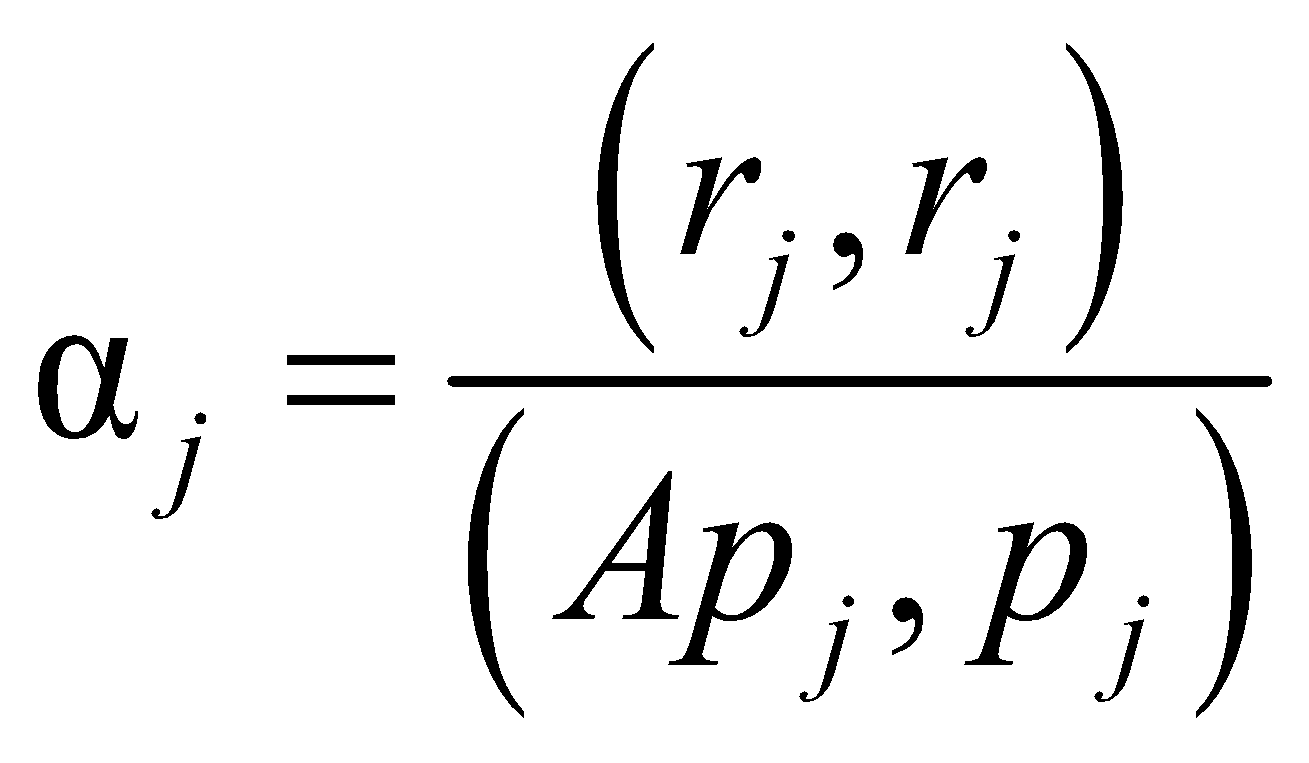
* *n* – одно из чисел в пределах от 1500 до 2000;
* *m* – номер в списке студенческой группы;
* *k* – номер студенческой группы.

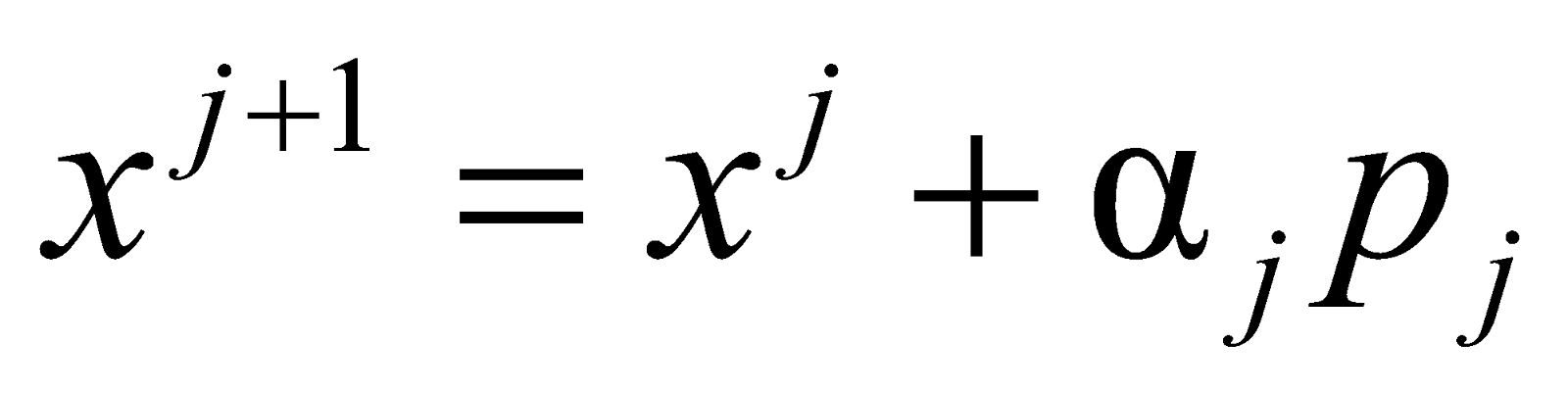
**Алгоритм сопряженных градиентов и соответствующий псевдокод.**

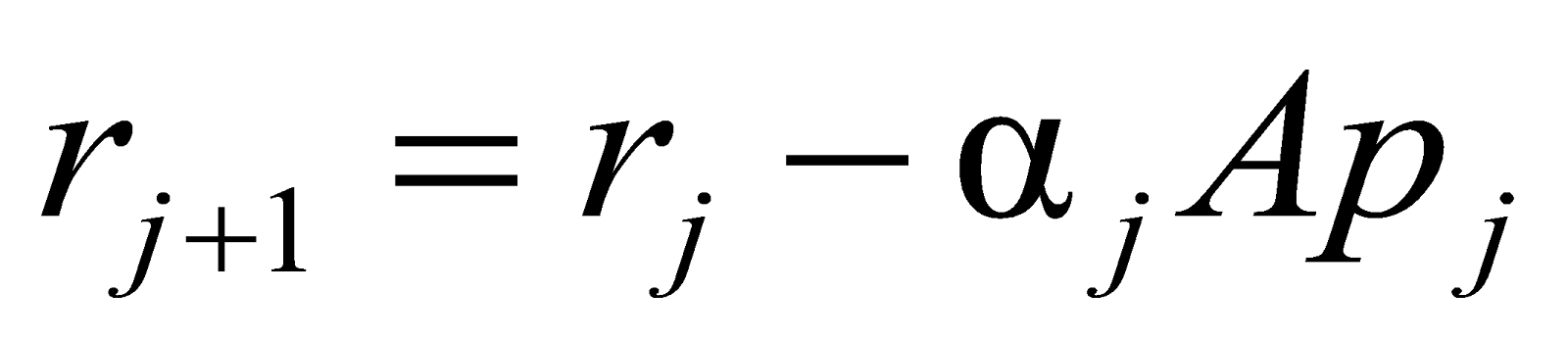
Алгоритм CG:

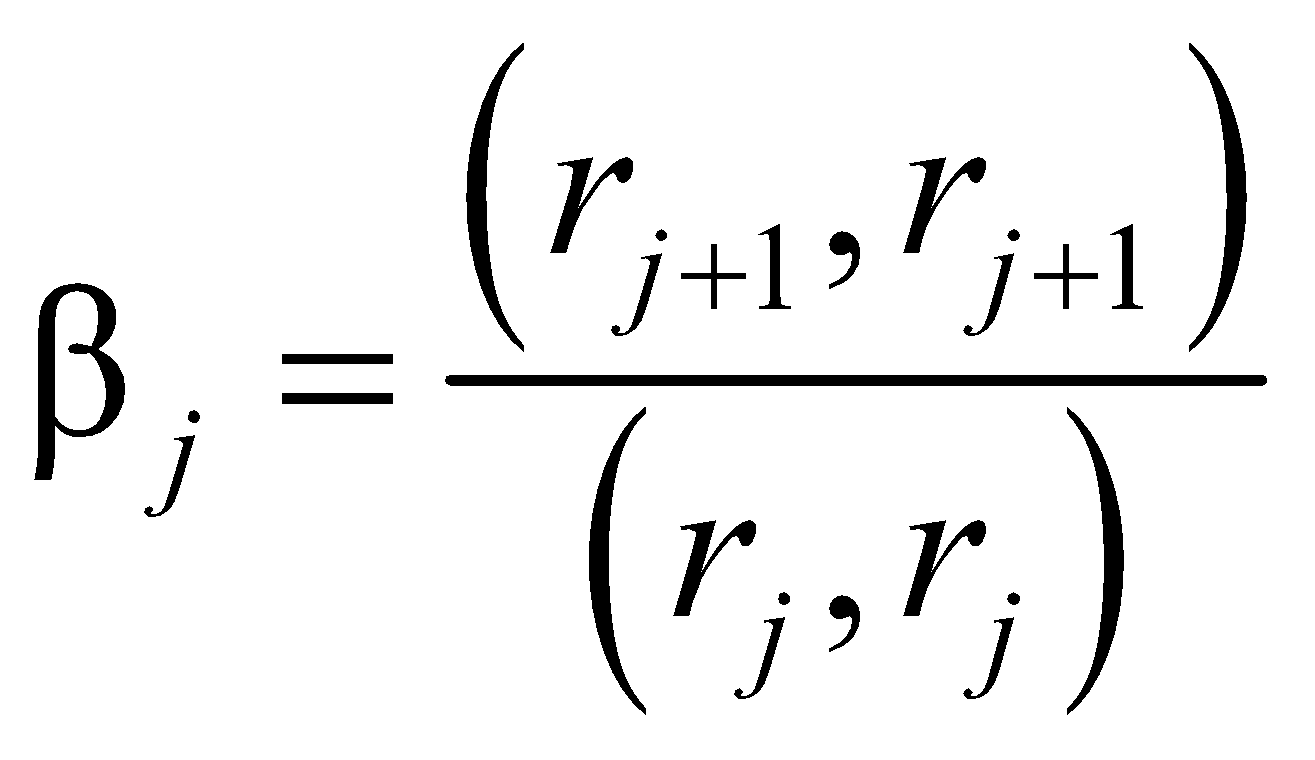
Выбрать , вычислить , положить 

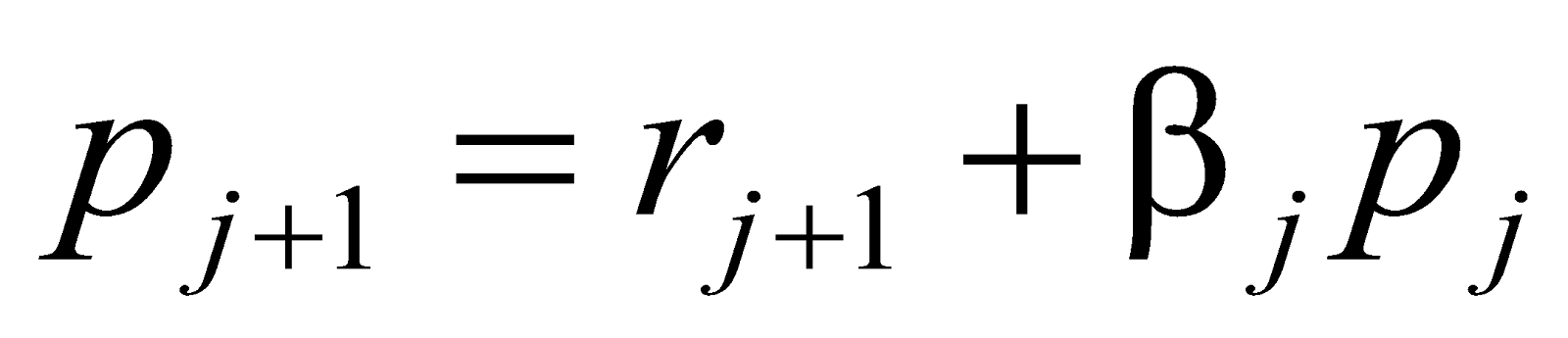
*j=*0,1,2, …(пока не достигнут критерий остановки):

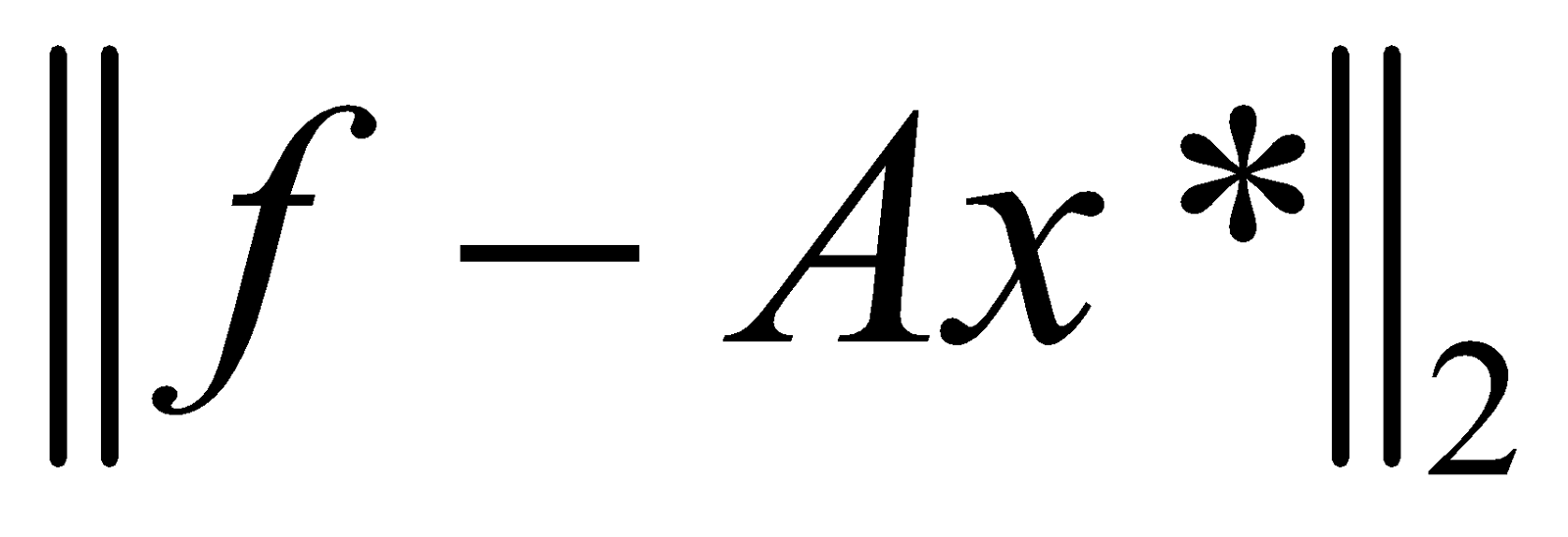
 // вспомогательный коэффициент

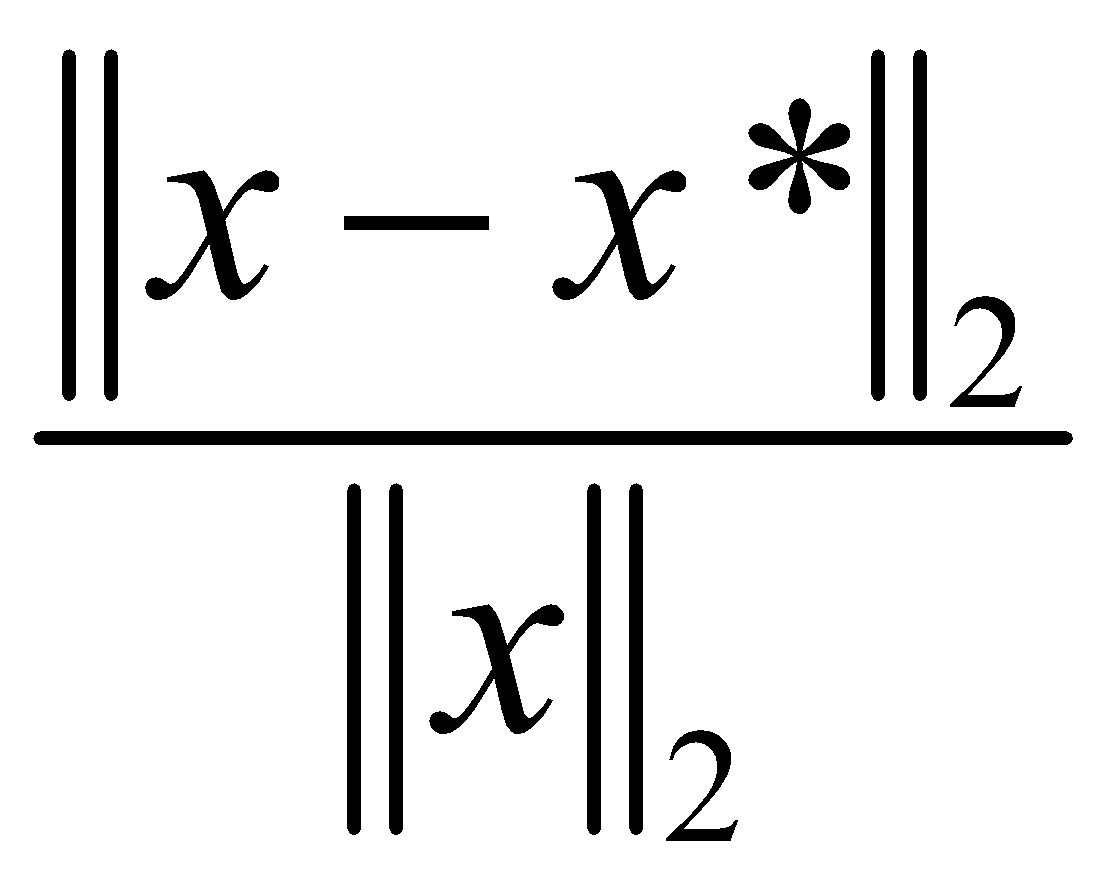
 // приближение на текущей итерации

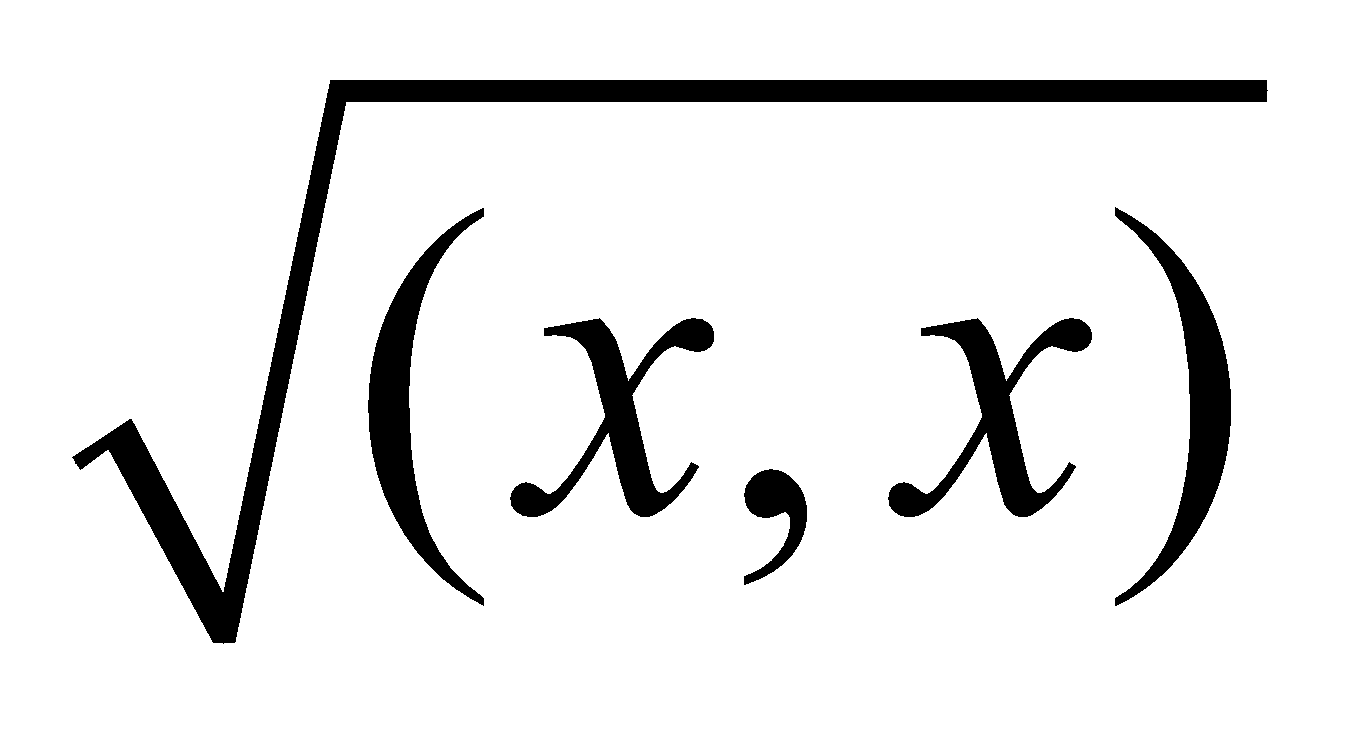
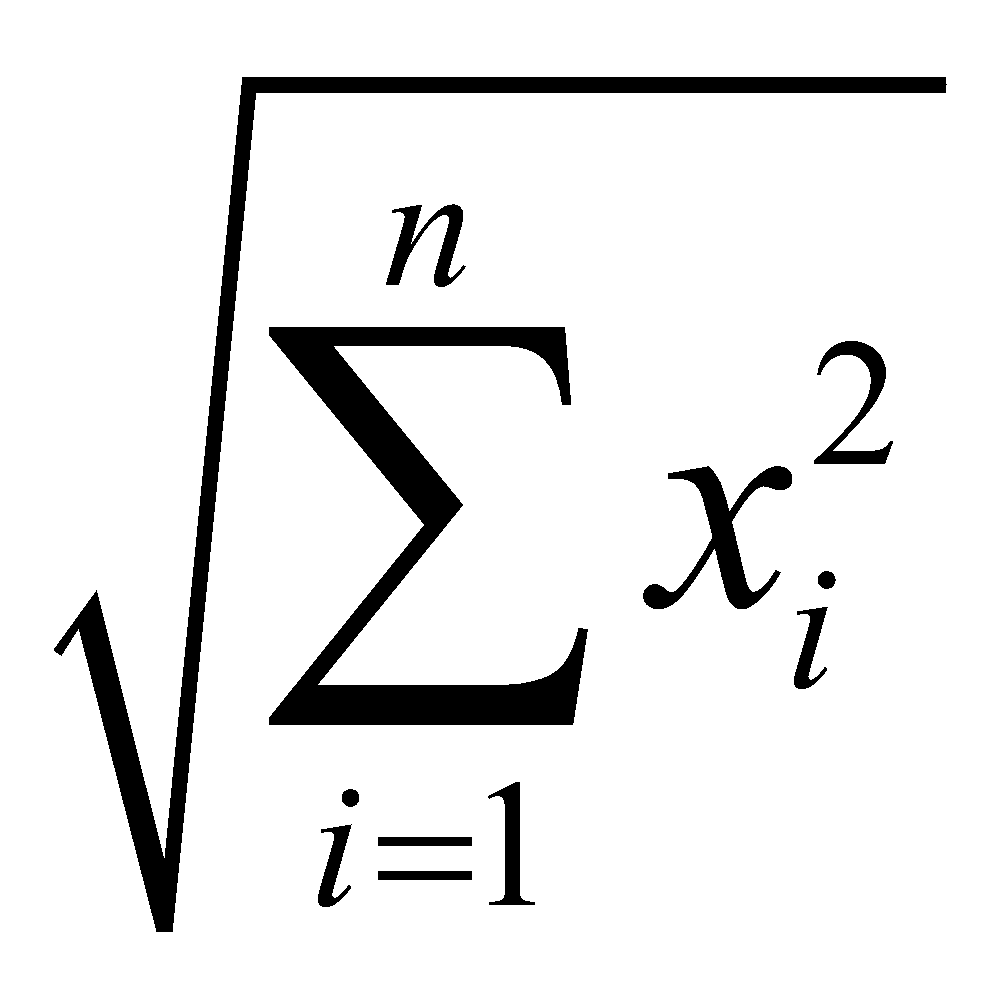
 // невязка

 // вспомогательный коэффициент

 // градиент

Норма вектора невязки 

Относительная погрешность: 

Сферическая (или евклидова) норма: |*x*||2==.

**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;

}

*// фунция создания вектора x*

*#define* M *4*

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

{

    std::*vector*<double> 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 для вычисления скалярного произведения векторов*

double CalculateScalarProduct(*const* std::*vector*<double> *&*vector1, *const* std::*vector*<double> *&*vector2)

{

    double result = *0*;

*for* (size\_t i = *0*; i < vector1.size(); ++i)

    {

        result += vector1[i] \* vector2[i];

    }

*return* result;

}

*#define* L\_IT *50*

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

{

    std::*vector*<double> xl(n, *0*);

    std::*vector*<double> rl = vector, pl = vector;

    double scalRlRl = CalculateScalarProduct(rl, rl);

*for* (int i = *0*; i < L\_IT; ++i)

    {

        double rlRL = scalRlRl;

        std::*vector*<double> apl = MatrixVectorMultiply(matrix, pl);

        double scalAplPl = CalculateScalarProduct(apl, pl);

        double alpha = rlRL / scalAplPl;

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

        {

            xl[j] += alpha \* pl[j];

            rl[j] -= alpha \* apl[j];

        }

        scalRlRl = CalculateScalarProduct(rl, rl);

        double beta = scalRlRl / rlRL;

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

        {

            pl[j] = rl[j] + beta \* pl[j];

        }

    }

*return* xl;

}

*// 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 нормы вектора невязки*

double GetNormOfResidualVector(*const* std::*vector*<std::*vector*<double>> *&*matrix, *const* std::*vector*<double> *&*calculatedVector, *const* std::*vector*<double> *&*f)

{

    std::*vector*<double> ax = MatrixVectorMultiply(matrix, calculatedVector);

*// подсчет вектора f-Ax\**

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

    {

        ax[i] = f[i] - ax[i];

    }

*// подсчет нормы вектора и возвращение результата*

*return* CalculateEuclideanNorm(ax);

}

*// функция подсчета относительной погрешности*

double GetRelativeError(*const* std::*vector*<double> *&*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);

}

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);

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

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

*// решение СЛАУ методом сопряженных градиентов*

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

*const* std::*vector*<double> res = RunCG(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 СЛАУ с использованием LDL^T разложения*

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

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

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

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

*// вывод результатов*

    PrintVector(res, *5*, "vector x calculated using CG algorithm");

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

*// подсчет и вывод нормы ветора невязки*

    std::cout << "Norm of residual vector (CG): " << GetNormOfResidualVector(matrix, res, b) << '*\n*';

    std::cout << "Norm of residual vector (LDL^T): " << GetNormOfResidualVector(matrix, x, b) << '*\n*';

*// подсчет и вывод относительной погрешности*

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

    std::cout << "RelativeError(LDL^T): " << GetRelativeError(vector, x) << '*\n*';

*// вывод времени выполнения*

    std::cout << "Time (CG): " << time1 << "ms*\n*";

    std::cout << "Time (LDL^T): " << time2 << "ms*\n*";

*return* *0*;

}

**Выходные данные.**

vector x calculated using CG algorithm 3.9983 4.9983 5.9983 6.9983 7.9983

vector x calculated using LDL^T decomposition 4.00283 5.00283 6.00283 7.00283 8.00283

Norm of residual vector (CG): 3.43209e+10

Norm of residual vector (LDL^T): 3.4321e+10

RelativeError(CG): 1.63039e-06

RelativeError(LDL^T): 2.7147e-06

Time (CG): 1180ms

Time (LDL^T): 6877ms

**Выводы.**

Метод сопряженных градиентов значительно превосходит LDLT разложение по скорости.