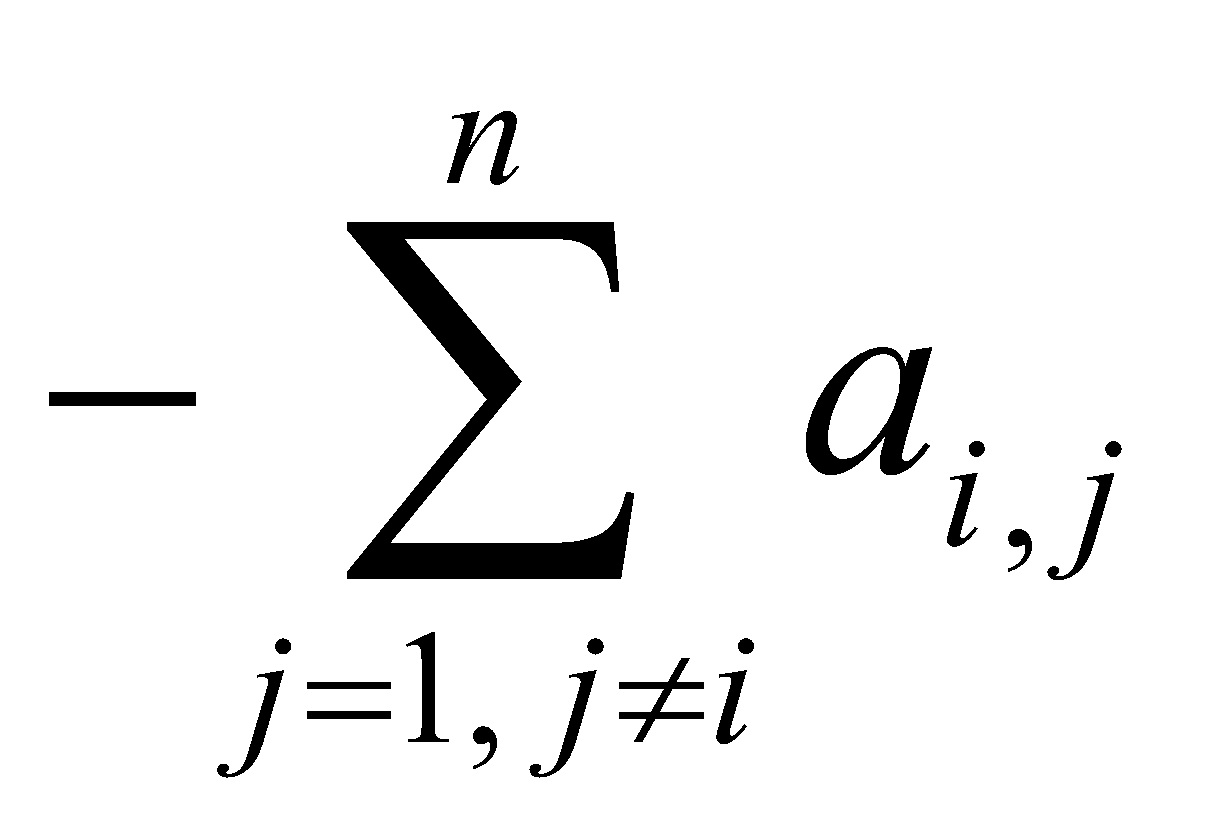
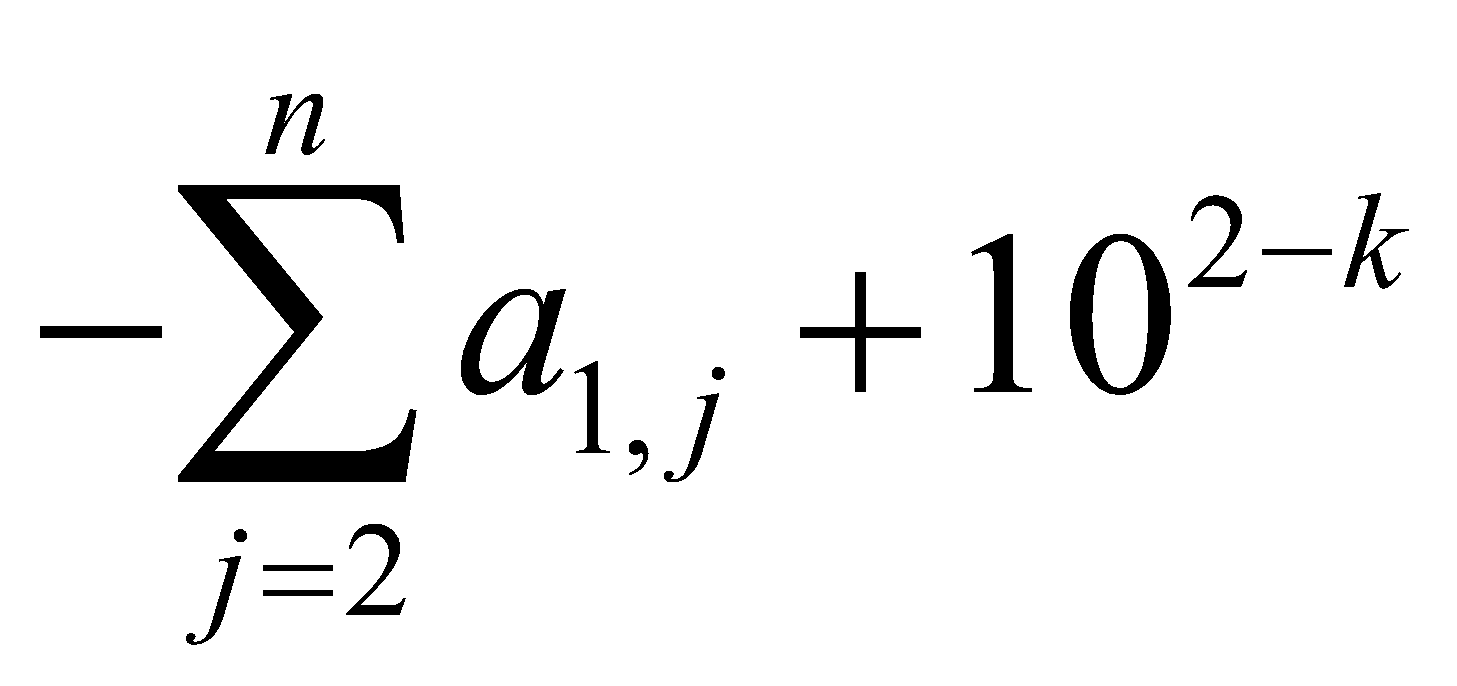
**Лабораторная работа 6 «Предобусловливание»**

Выполнил студент 3 курса 4 группы ФПМИ БГУ Видевич Александр

**Задача.**

Изучить основы понятия предобусловливания. Проверить (для рассматриваемого примера), насколько для итерационного процесса эффективны самые простые предобусловливатели – диагональные.

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 лабораторной работы 5 повторить.

Случай 1Я: ещё раз выполнить случай 1, но перед началом итерационного процесса произвести предобусловливание Якоби.

Случай 1М: ещё раз выполнить случай 1, но перед началом итерационного процесса произвести масштабирование.

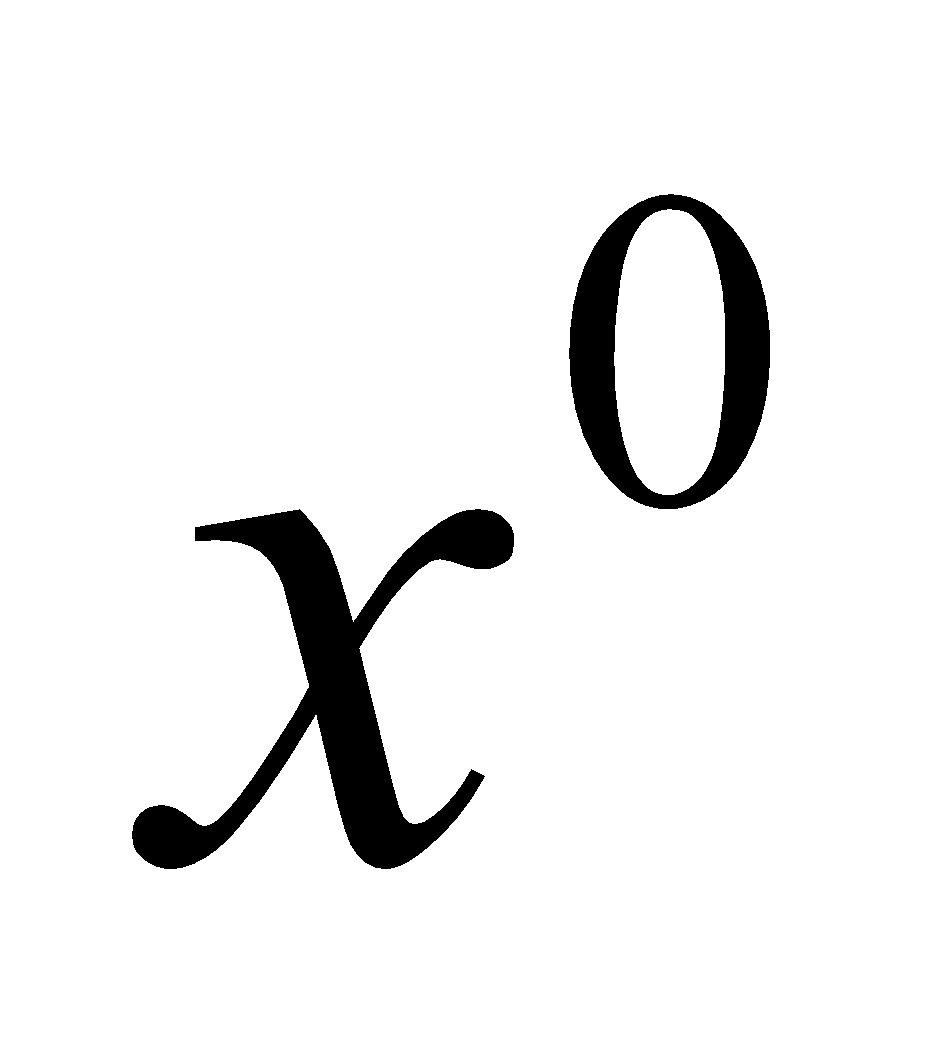
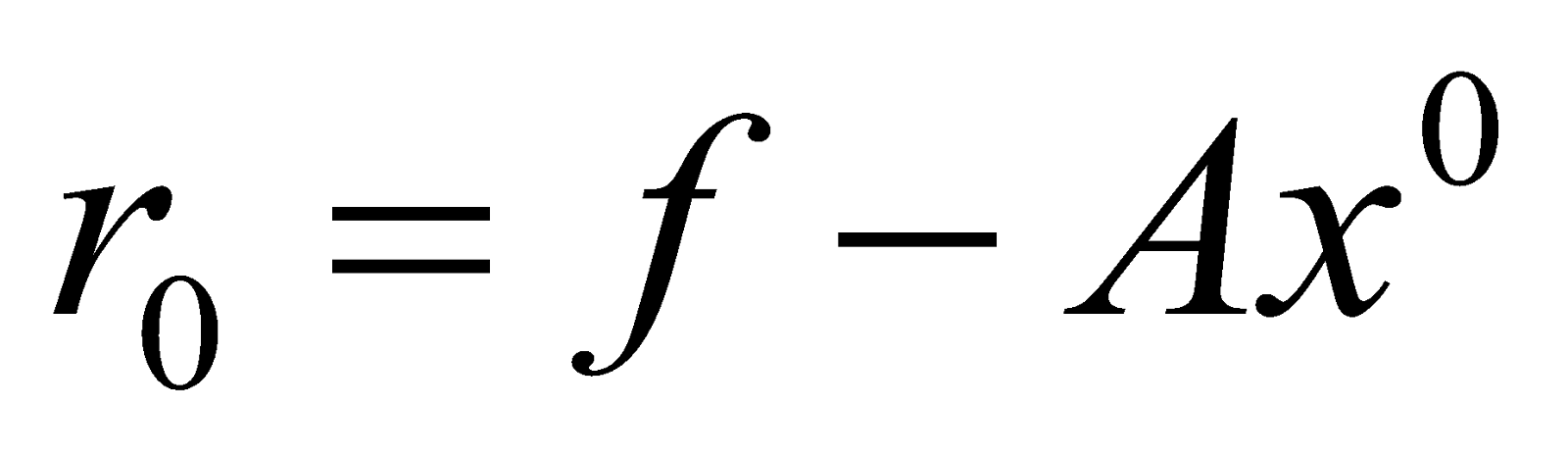
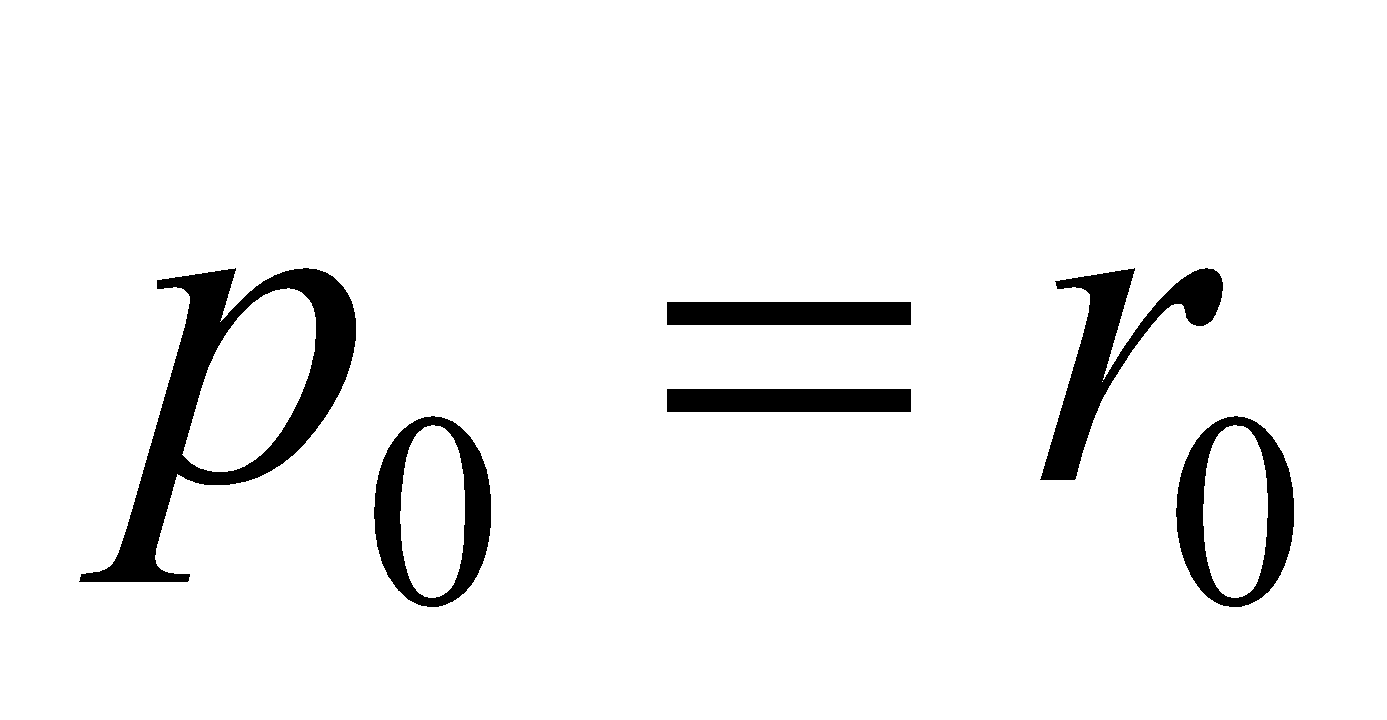
Система уравнений во всех трёх случаях изначально одна и та же

Для вычислений выбрать параметры:

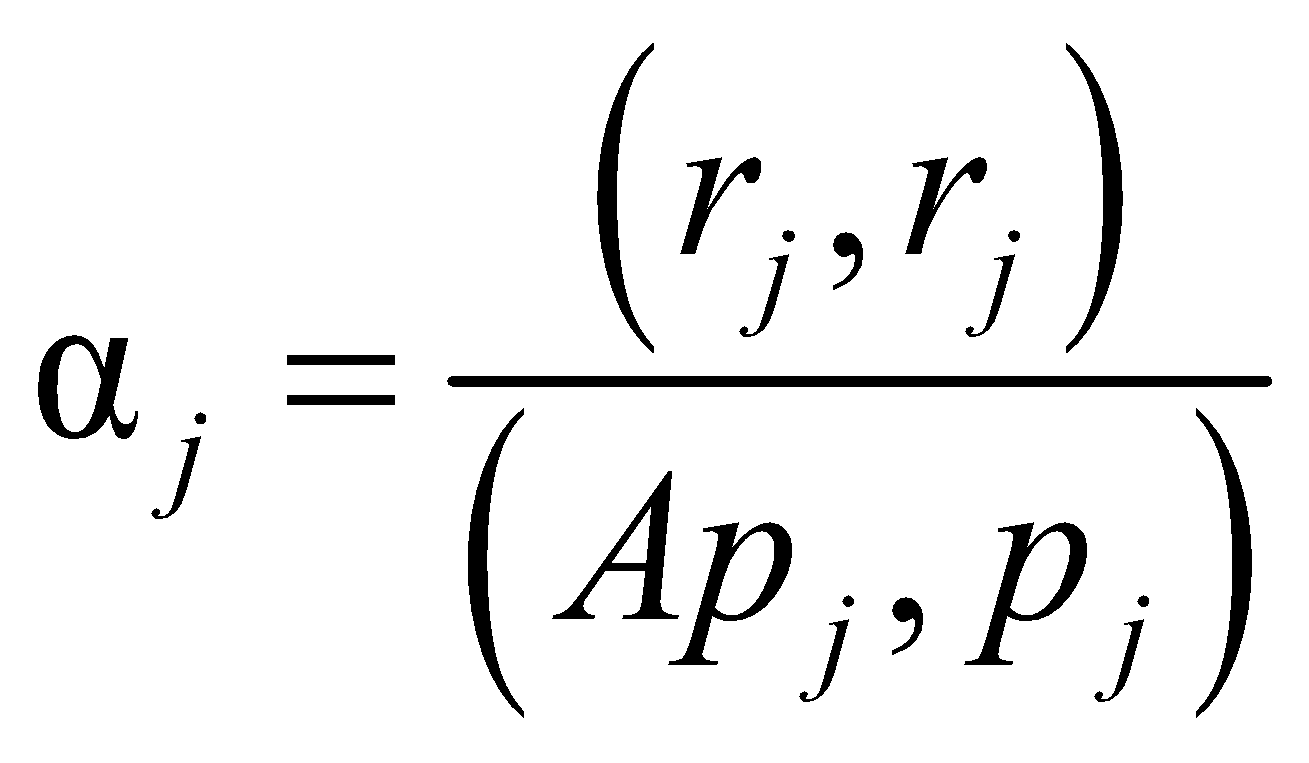
* *m* – номер в списке студенческой группы;
* *k* – номер студенческой группы.

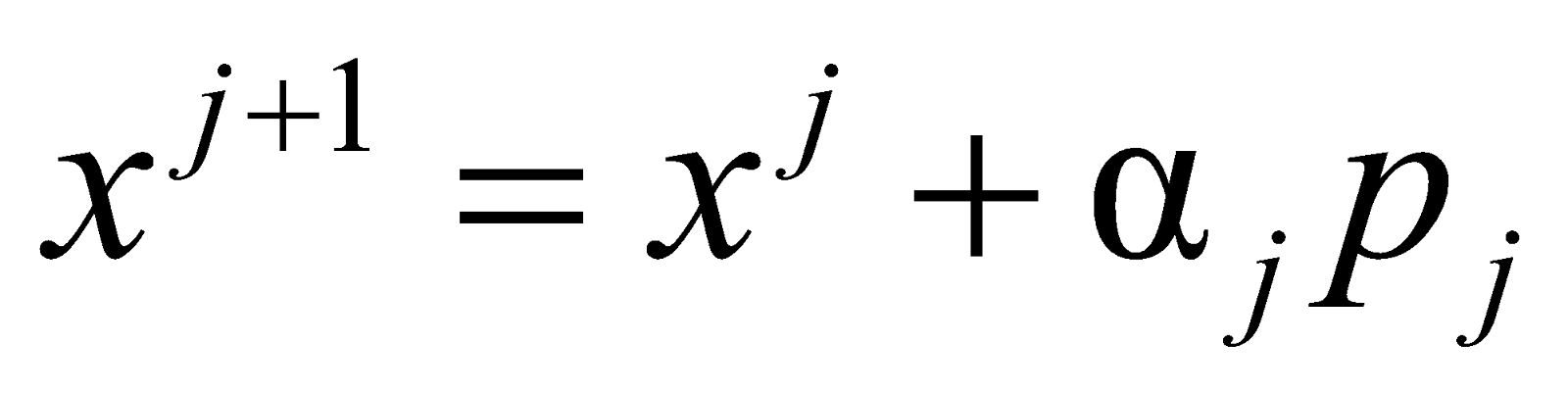
**Алгоритм сопряженных градиентов и соответствующий псевдокод.**

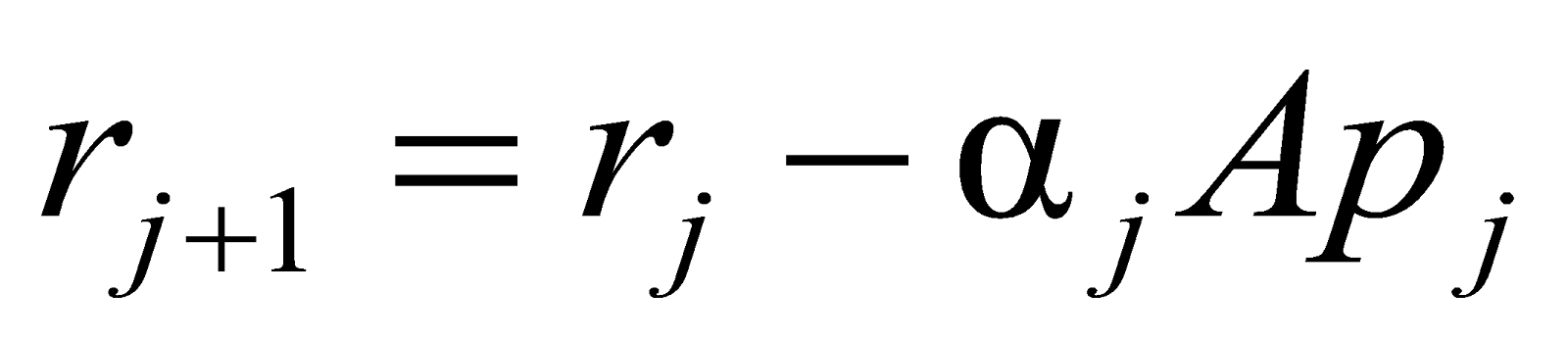
Алгоритм CG:

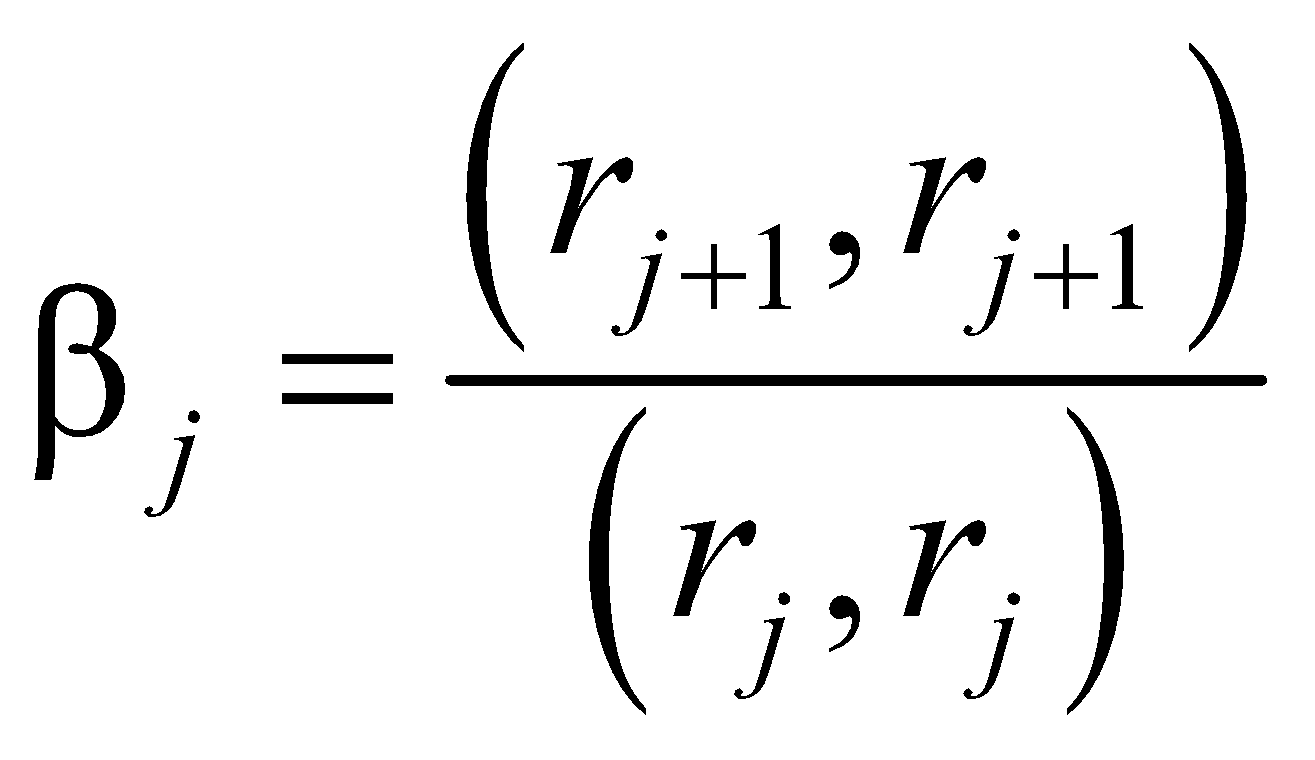
Выбрать , вычислить , положить 

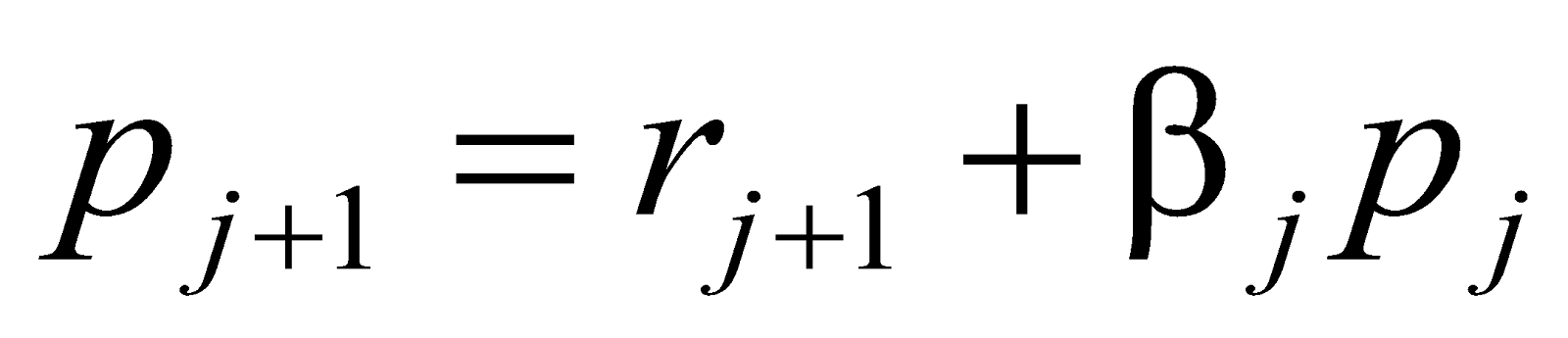
*j=*0,1,2, …(пока не достигнут критерий остановки):

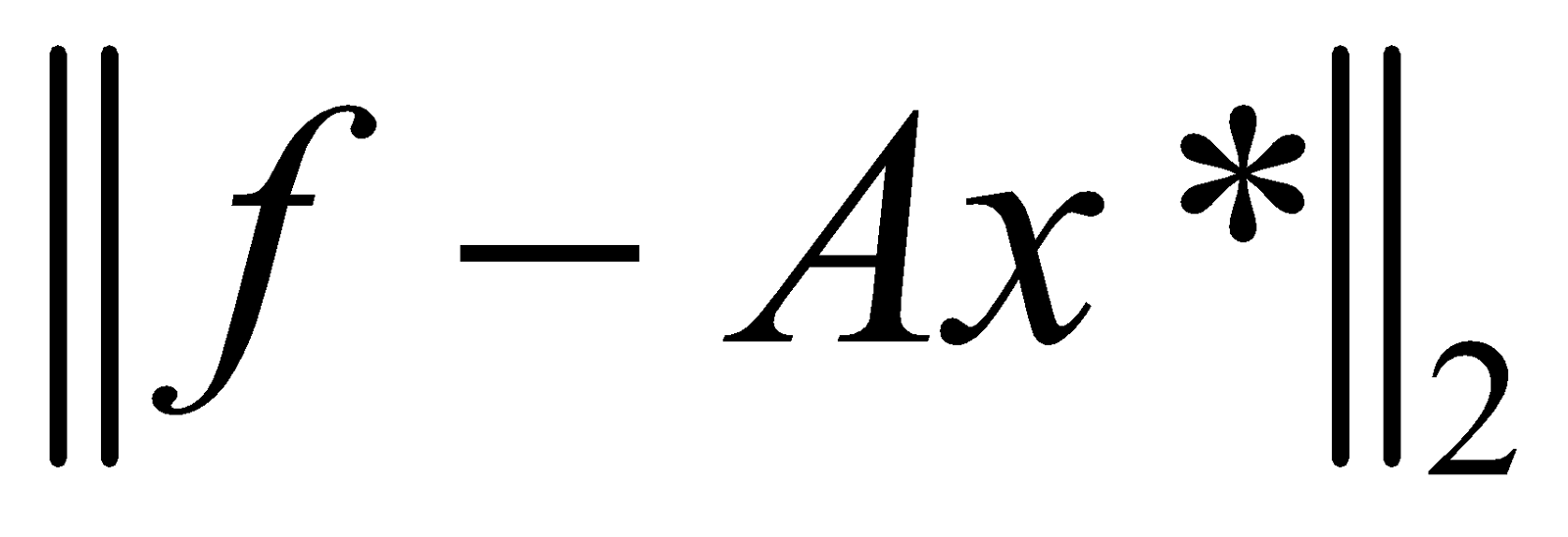
 // вспомогательный коэффициент

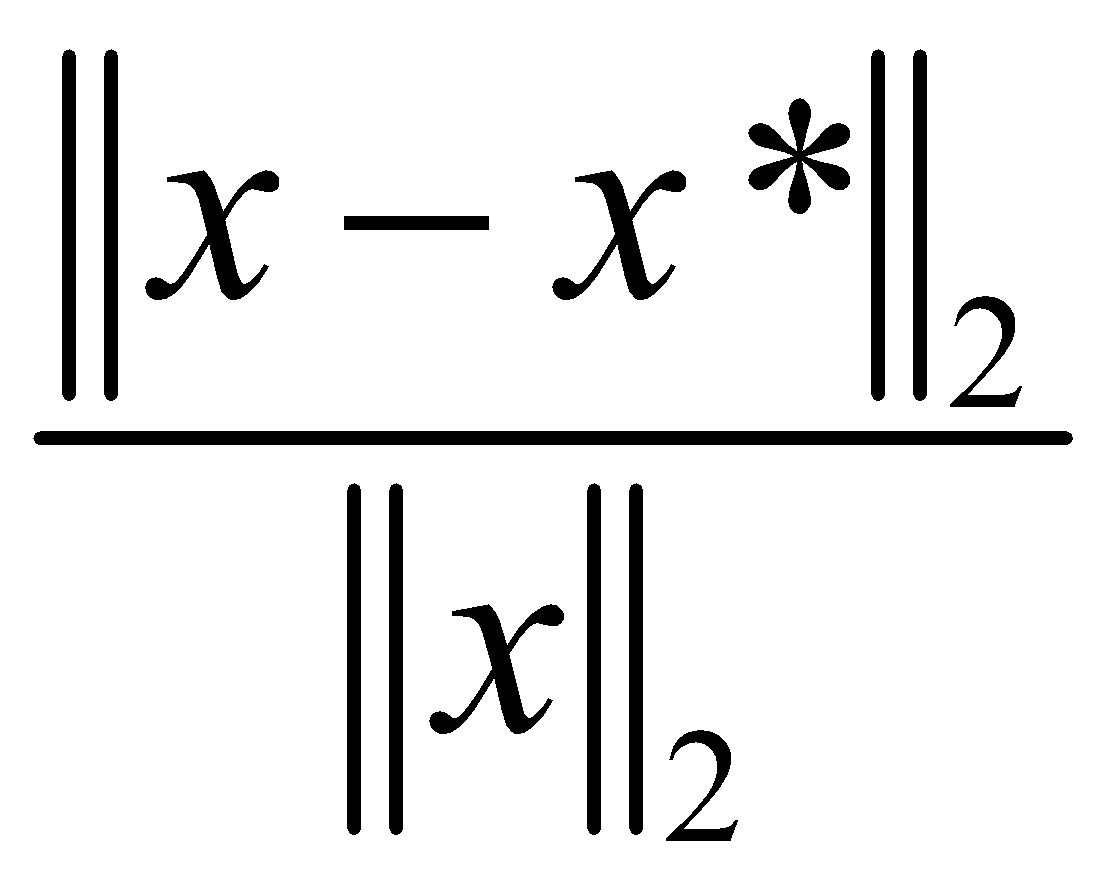
 // приближение на текущей итерации

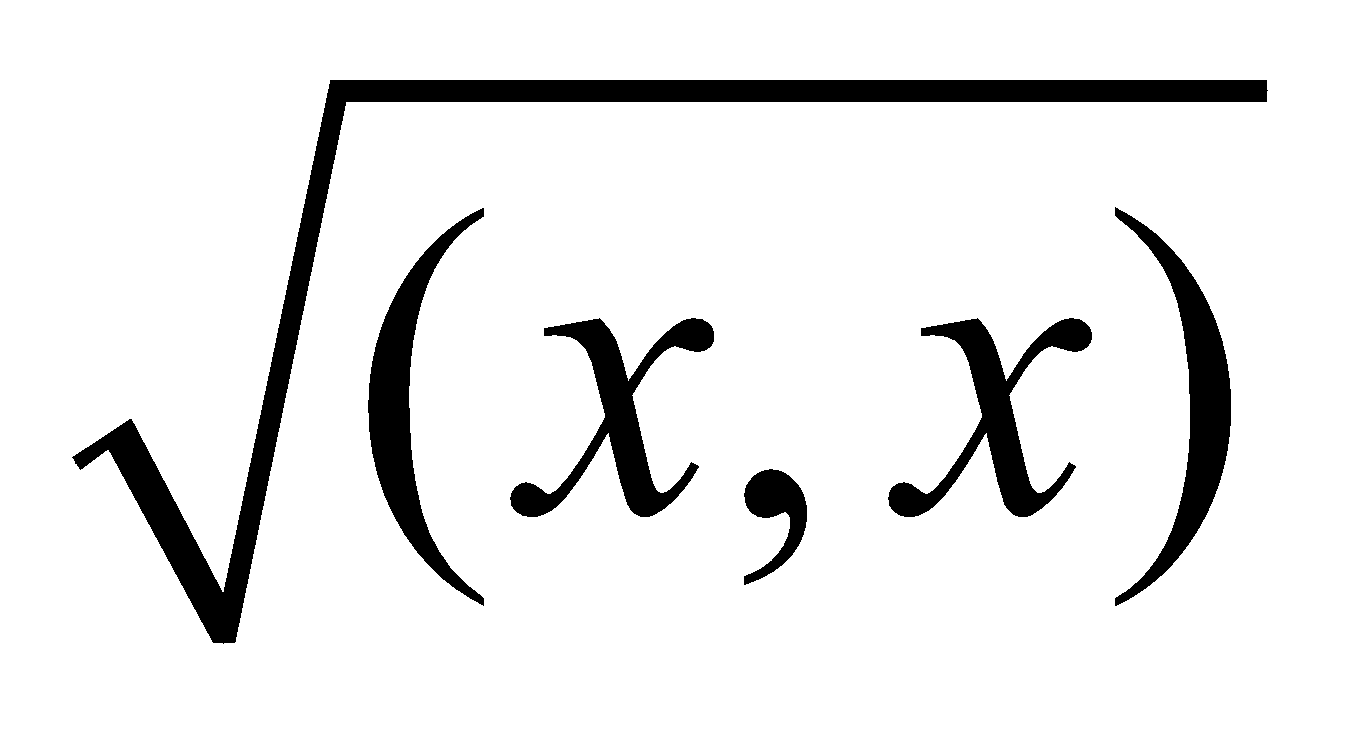
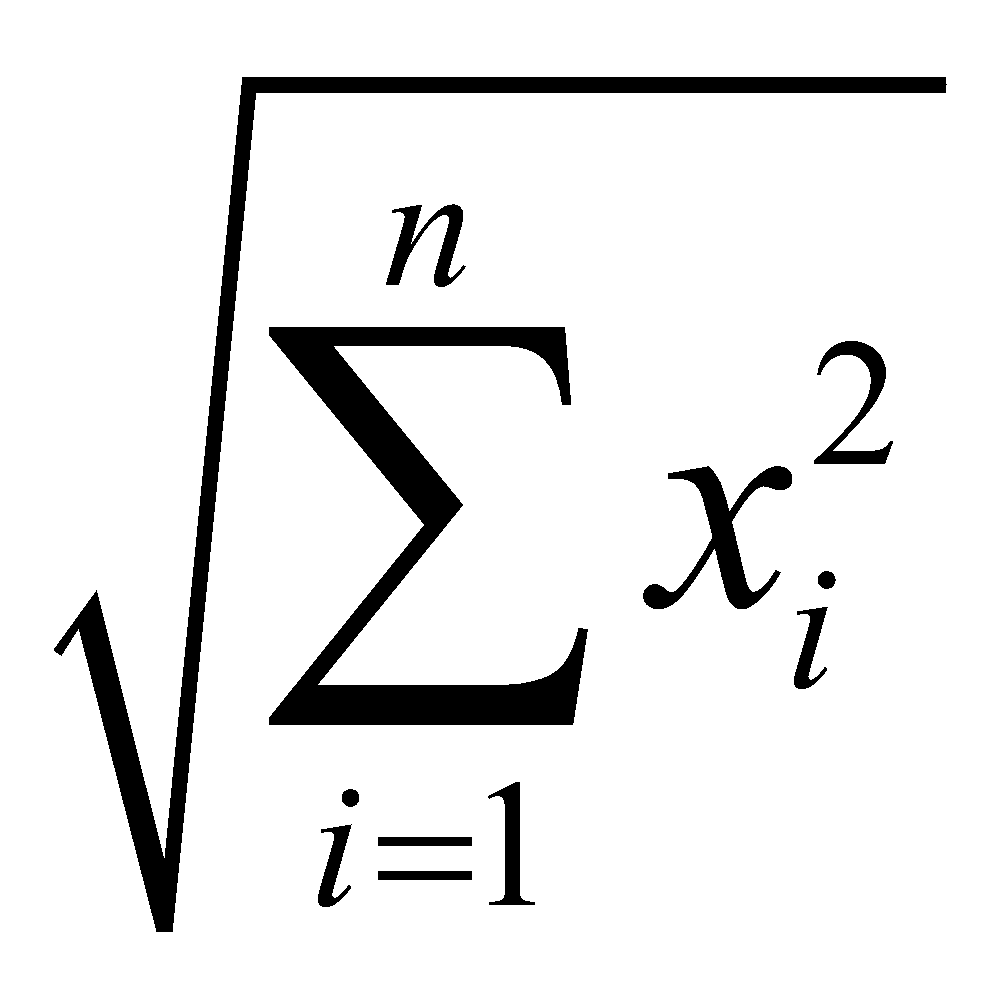
 // невязка

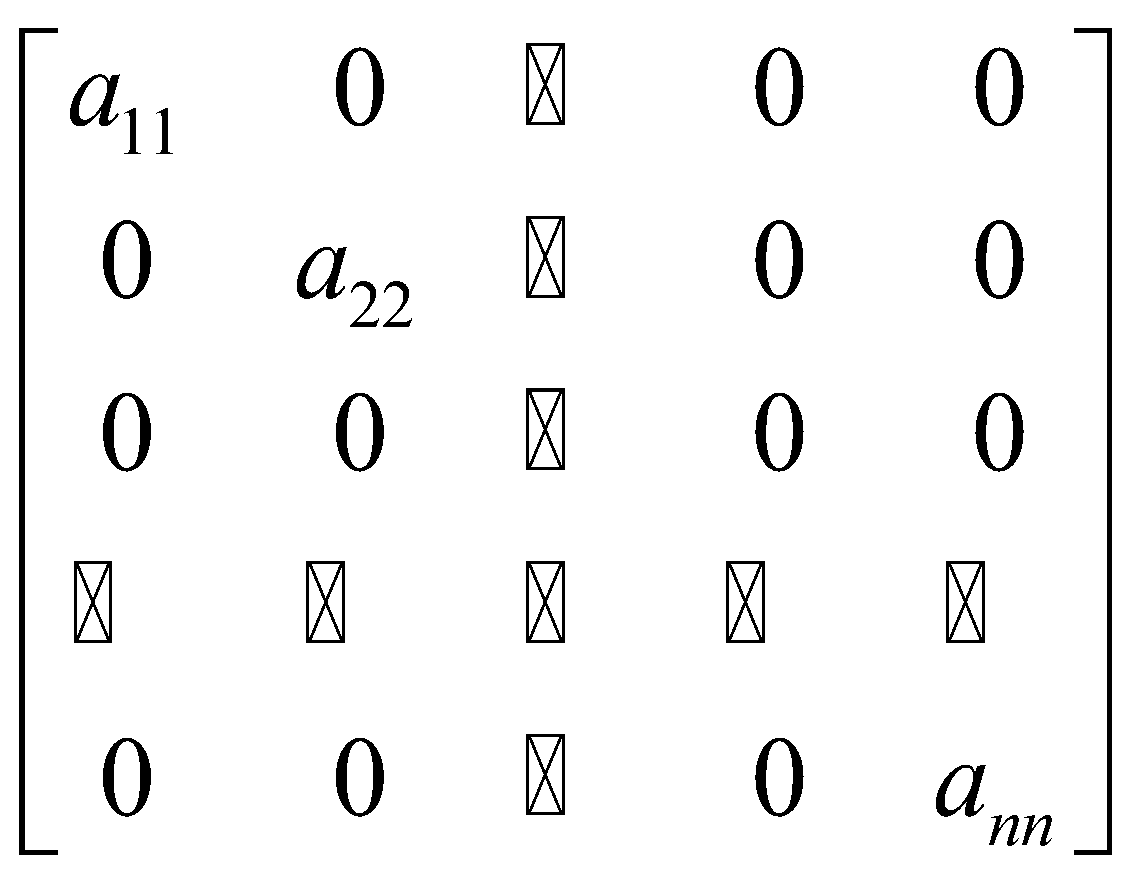
 // вспомогательный коэффициент

 // градиент

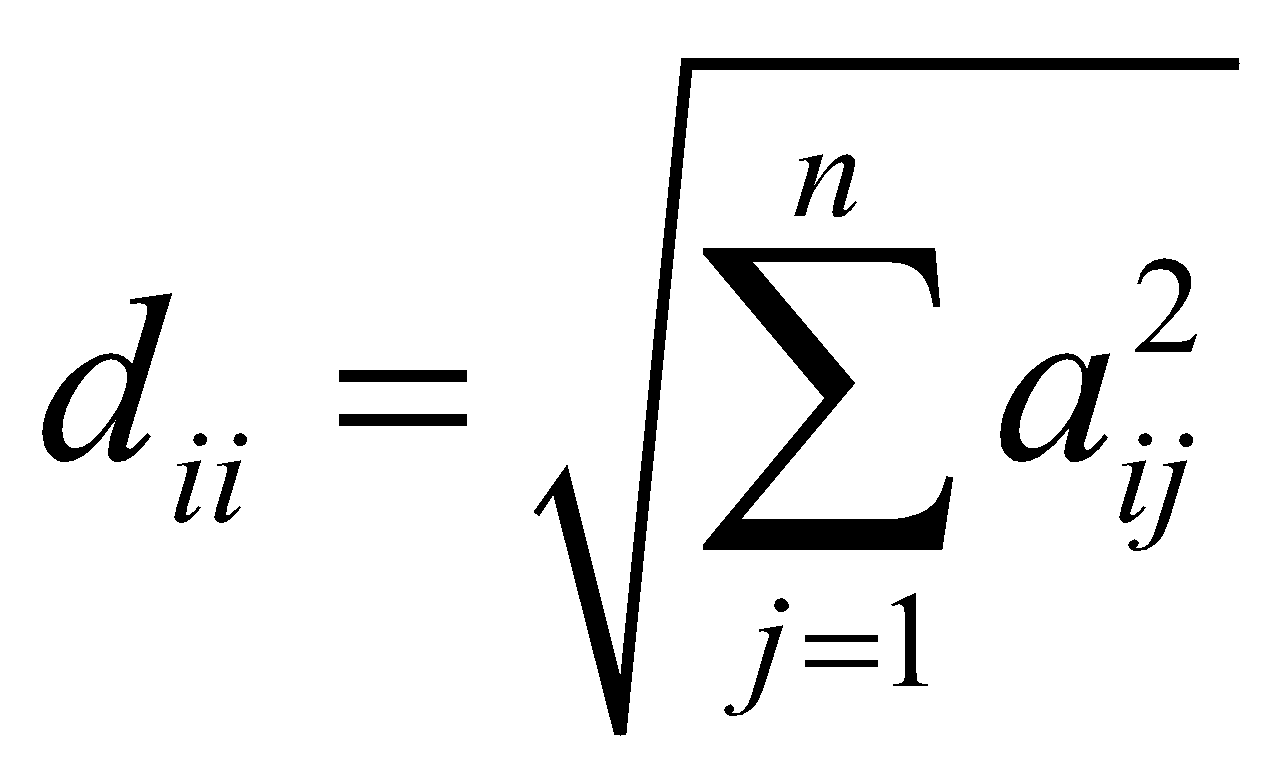
Норма вектора невязки 

Относительная погрешность: 

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

Предобусловливатель Якоби: DA*=**,*

Предобусловливатель масштабирования:

*M=*diag(*d*11,*d*22,…,*dnn*), .

**Входные данные.**

n = 5000, m = 4, k = 4

**Листинг программы.**

*#include* <iostream>

*#include* <vector>

*#include* <functional>

*#include* <ctime>

*#include* <chrono>

*#include* <cmath>

*#include* <iomanip>

int n = *5000*, K = *4*, M = *4*;

double eps = *0.00000001*;

*// функция подсчета квадратичной (евклидовой нормы)*

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

}

*// фунция создания вектора x*

std::*vector*<double> GetVector()

{

    std::*vector*<double> vector(n);

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

    {

        vector[i] = M + i;

    }

*return* vector;

}

*// функция генерации матрицы*

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;

}

*// разность векторов*

template <typename T>

std::*vector*<T> minus(*const* std::*vector*<T> *&*x1, *const* std::*vector*<T> *&*x2)

{

    std::*vector*<T> x(n);

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

    {

        x[i] = x1[i] - x2[i];

    }

*return* x;

}

*// функция умножения матрицы на вектор*

std::*vector*<double>

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

*if* (EuclideanNorm(rl) < eps)

        {

            std::cout << "Exit CG by r norm with number of Iteration: " << i << std::endl;

*break*;

        }

*if* (beta < eps)

        {

            std::cout << "Exit CG by beta" << std::endl;

*break*;

        }

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

        {

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

        }

    }

*return* xl;

}

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

{

    std::*vector*<double> result(n);

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

    {

        double inverse = *1* / matrix[i][i];

        result[i] = vector[i] \* inverse;

    }

*return* result;

}

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

{

    std::*vector*<double> xl(n, *0*);

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

    std::*vector*<double> pl = MakeJacobyPrecondition(rl, matrix);

    double scalMRlRl = CalculateScalarProduct(pl, rl);

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

    {

        double rlRL = scalMRlRl;

        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];

        }

        scalMRlRl = CalculateScalarProduct(MakeJacobyPrecondition(rl, matrix), rl);

        double beta = scalMRlRl / rlRL;

*if* (EuclideanNorm(rl) < eps)

        {

            std::cout << "Exit JCG by r norm with number of Iteration: " << i << std::endl;

*break*;

        }

*if* (beta < eps)

        {

            std::cout << "Exit JCG by beta" << std::endl;

*break*;

        }

        std::*vector*<double> pre\_rl = MakeJacobyPrecondition(rl, matrix);

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

        {

            pl[j] = pre\_rl[j] + beta \* pl[j];

        }

    }

*return* xl;

}

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

{

    std::*vector*<double> result(n);

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

    {

        double sum = *0*;

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

        {

*if* (i >= j)

            {

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

            }

*else*

            {

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

            }

        }

        result[i] = vector[i] \* *1* / (std::sqrt(sum));

    }

*return* result;

}

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

{

    std::*vector*<double> xl(n, *0*);

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

    std::*vector*<double> pl = MakeScalingPrecondition(rl, matrix);

    double scalMRlRl = CalculateScalarProduct(pl, rl);

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

    {

        double rlRL = scalMRlRl;

        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];

        }

        scalMRlRl = CalculateScalarProduct(MakeScalingPrecondition(rl, matrix), rl);

        double beta = scalMRlRl / rlRL;

*if* (EuclideanNorm(rl) < eps)

        {

            std::cout << "Exit SCG  by r norm with number of Iteration: " << i << std::endl;

*break*;

        }

*if* (beta < eps)

        {

            std::cout << "Exit SCG by beta" << std::endl;

*break*;

        }

        std::*vector*<double> pre\_rl = MakeScalingPrecondition(rl, matrix);

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

        {

            pl[j] = pre\_rl[j] + beta \* pl[j];

        }

    }

*return* xl;

}

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*';

}

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* EuclideanNorm(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* EuclideanNorm(temp) / EuclideanNorm(originalVector);

}

int main()

{

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

    std::cin.tie(*nullptr*);

    std::srand(*1234*);

*// ввод данных*

*// std::cin >> n;*

*// генерация задачи*

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

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

*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> resCG = 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();

*// Предобуславливатель Якоби*

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

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

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

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

*// Второй предобуславливатель*

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

*const* std::*vector*<double> res2CG = RunScalingCG(matrix, b);

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

*const* int time3 = std::chrono::duration\_cast<std::chrono::milliseconds>(end - start).count();

*// вывод результатов*

    PrintVector(resCG, *5*, "Result vector (CG)");

    PrintVector(resJacobyCG, *5*, "Result vector (CG+Jacobi)");

    PrintVector(res2CG, *5*, "Result vector (CG+scaling)");

*// подсчет и вывод нормы ветора невязки*

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

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

    std::cout << "Norm of residual vector (CG+scaling): " << GetNormOfResidualVector(matrix, res2CG, b) << '*\n*';

*// подсчет и вывод относительной погрешности*

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

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

    std::cout << "RelativeError (CG+scaling): " << GetRelativeError(vector, res2CG) << '*\n*';

*// вывод времени выполнения*

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

    std::cout << "Time (CG+Jacobi): " << time2 << "ms*\n*";

    std::cout << "Time (CG+scaling): " << time3 << "ms*\n*";

*return* *0*;

}

**Выходные данные.**

Exit CG by r norm with number of Iteration: 25

Exit JCG by r norm with number of Iteration: 22

Exit SCG by r norm with number of Iteration: 22

Result vector (CG) 4.04728 5.04728 6.04728 7.04728 8.04728

Result vector (CG+Jacobi) 3.97695 4.97695 5.97695 6.97695 7.97695

Result vector (CG+scaling) 4.02595 5.02595 6.02595 7.02595 8.02595

Norm of residual vector (CG): 0.000940847

Norm of residual vector (CG+Jacobi): 0.0010014

Norm of residual vector (CG+scaling): 0.000988887

RelativeError (CG): 1.63598e-05

RelativeError (CG+Jacobi): 7.97563e-06

RelativeError (CG+scaling): 8.97919e-06

Time (CG): 5684ms

Time (CG+Jacobi): 4972ms

Time (CG+scaling): 16753ms

**Выводы.**

Предобусловливание увеличивает эффекнтивность работы алгоритма.