МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»

Институт информатики, математики и электроники
Факультет информатики
Кафедра технической кибернетики

Векторные алгоритмы матричных разложений

Отчет по лабораторной работе № 2

Студенты группы 6407 Исаев М. А.
Степаненко С. О.
Проверил Головашкин Д. Л.

САМАРА 2018

СОДЕРЖАНИЕ

ЗАДАНИЕ

В данной работе требуется реализовать разложение Холецкого, используя алгоритмы IJK и JIK, а также блочный алгоритм с использованием скалярного произведения.

ВВЕДЕНИЕ

Основными методами разложения матрицы являются LU, LDM^T, LDL^T, разложение Холецкого и QR алгоритм, который разделяется на метод Хаусхолдера и метод вращений Гивенса, а также метод Шура.

В данной лабораторной работе мы рассмотрим разложение Холецкого, реализованное алгоритмами JIK, IJK, блочную реализацию с использованием скалярного произведения, а так же оценим эффективность применения SSE технологии и сравним результаты с встроенной библиотекой. Для того, чтобы матрицу можно было разложить, необходимо, чтобы она была симметричной, положительно-определённой со строго положительными элементами на диагонали.

Теоретические сведения

Разложение Холецкого — представление симметричной положительно-определённой матрицы A в виде $A = G \times G^T$, где G – нижняя треугольная матрица со строго положительными элементами на диагонали.

IJK – разложение

```
G(1, 1) = sqrt(A(1, 1))
for i = 2 : n
    for j = 1 : i
        G(i, j - 1)  =  A(i,  j - 1) / G(j - 1, j - 1)
        A(i,  j)  =  A(i,  j) - G(i, 1 : j - 1) * G(1 : j - 1, j).T
    end
    G(i, i )  =  sqrt(A(i,  i))
end
```

JIK- разложение

```
for j = 1 : n
    for i = 1 : j - 1
        G(i ,  j) = (A(i,  j)  - (G(1 : i - 1, i) * G(1 : i - 1,  j))) / G(i, i)
    end
    G(j, j) = sqrt( A(j, j) - G(1 : j - 1, j) ^ 2)
end
```

Блочный алгоритм разложения Холецкого

```
for j = 1 : N
    for i = j : N
        S(i, j) = A(i, j) - G(i, 1 : j - 1) * G(j, 1 : j - 1).T
        if i = j then
            G(j, j) * G(j, j).T = S(j, j)
        else  G(i, j) * G(j, j).T = S(i, j)
    end
end
```

Цель эксперимента

Целью эксперимента является выявление зависимости времени работы алгоритма от размера матрицы и выбор наилучшего алгоритма. А также определение оптимального блока для блочной версии алгоритма. Для этого перемножим матрицы различных размерностей всеми алгоритмами и проанализируем полученные результаты.

Инструментарий

В эксперименте использовались следующие средства:

- аппаратные: Intel Core i5 2.4GHz, 8Gb RAM;
- системные: macOS Mojave;
- программные: g++ GNU Compiler Collection version 7.3.0;
- кэш: 3 MB;
- для построения графиков использовался python 3.6.5;
- язык программирования: C++ стандарта C++11.

Данное оборудование удовлетворят всем необходимым аппаратным и программным требованиям, т.к. оно позволяет выполнять высокопроизводительные вычисления с достаточной степенью точности.
Язык программирования был выбран, исходя из возможности векторной компиляции.

Параметры эксперимента

В качестве параметра эксперимента используется размер матриц. Матрицы инициализируются случайными числами, далее к диагональным элементам матрицы прибавляется такое число, что в ходе преобразования для невырожденных матриц мы получаем симметричную положительно-определённую матрицу. В ходе эксперимента брались матрицы размерностью от 64×64 до 1024×1024, далее время работы алгоритмов существенно возрастает. Проведем серию экспериментов, увеличивая размерность матрицы в два раза, для алгоритмов IJK и JIK. Для блочного алгоритма проведем похожий эксперимент, в котором размерность блока будет изменяться от 2 до 1024 каждый раз увеличивая размерность блока в 2 раза, размерность матрицы 1024×1024. Результаты эксперимента представлены в таблице 1.

Ожидаемые результаты

В ходе вычислительного эксперимента ожидаются следующие результаты: стандартный алгоритм библиотеки Eigen C++ должен показать наилучшее время работы, в силу того, что это библиотека линейной алгебры с операциями для векторов и матриц. Блочный-алгоритм разложения Холецкого должен выполняться быстрее JIK и IJK в силу особенностей представления данных. Далее идёт IJK, а следующим по скорости ожидается алгоритм JIK.

При запуске алгоритмов с SSE модулем, предназначенным для распараллеливания вычислительного процесса между данными, ожидается ускорение в работе каждого алгоритма за счет глобальной оптимизации на уровне функций. Оптимальный размер блока для блочного алгоритма найдём по формуле:

$$blockSize = \sqrt{\frac{cacheSize}{1 \times doubleSize}},$$

- где cacheSize – размер КЭШ памяти;
- doubleSize – размер типа данных double;
- blockSize – размерность блока.

При наших исходных данных ожидаемый оптимальный размер блока 512.

Описание эксперимента

Для выявления зависимости времени работы алгоритма от размера матрицы и выбора наилучшего алгоритма, перемножим матрицы различных размерностей всеми алгоритмами и проанализируем полученные результаты.

В процессе выполнения эксперимента были получены следующие результаты:

Таблица 1 - Время выполнения программы (секунды)

Размер матрицы	IJK	JIK	Eigen
64×64	0.000304	0.000235	0.001204
128×128	0.002254	0.001693	0.006242
256×256	0.016629	0.018804	0.038973
512×512	0.139684	0.243214	0.028045
1024×1024	1.054990	3.151864	1.085821

Таблица 2 - Время выполнения программы с использованием SSE (секунды)

Размер матрицы	IJK	JIK	Eigen
64×64	0.000150	0.000065	0.000082
128×128	0.001165	0.001222	0.000282
256×256	0.010138	0.016873	0.000983
512×512	0.087584	0.229044	0.014285
1024×1024	0.710952	2.209845	0.048326

Ввиду того, что до размера матриц 64×64, время выполнения очень мало, результаты в таблицах не приведены.

Полученные измерения также представлены на рисунке 1.

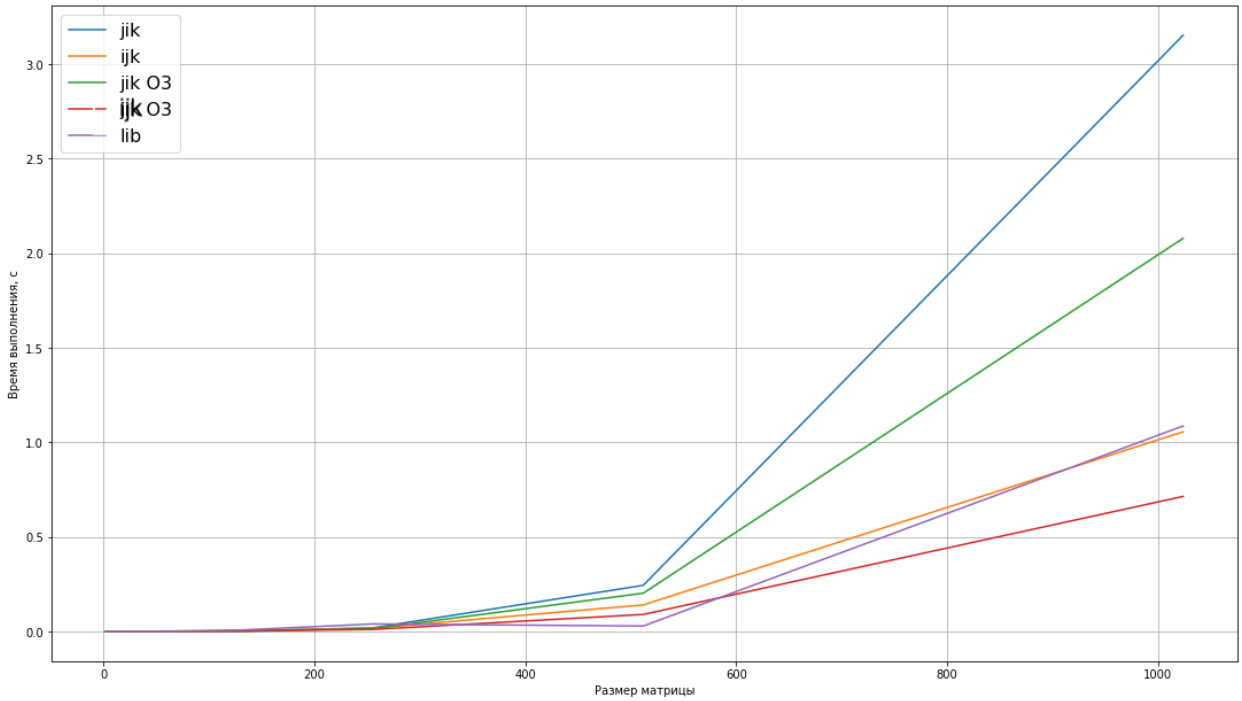

Рисунок 1- График зависимости времени выполнения реализаций алгоритмов от размеров матрицы

Результаты измерений времени блочного алгоритма, в зависимости от размерности блока, представлены в таблице 3 и на рисунках 2-3.

Таблица 3 - Время выполнения реализации блочного алгоритма

Размер блока	С векторизацией	Без векторизации
2×2	57.375400	65.336600
4×4	17.073700	27.139100
8×8	7.132480	11.347800
16×16	3.283650	7.047360
32×32	1.515560	4.978330
64×64	1.053690	4.046390
128×128	0.799695	2.580000
256×256	0.855029	2.956330
512×512	0.535405	1.654200
1024×1024	0.756296	1.738640

Исходя из таблицы, можно отметить, что оптимальным размером блока является 512 для алгоритма с векторизацией и 512 без векторизации.

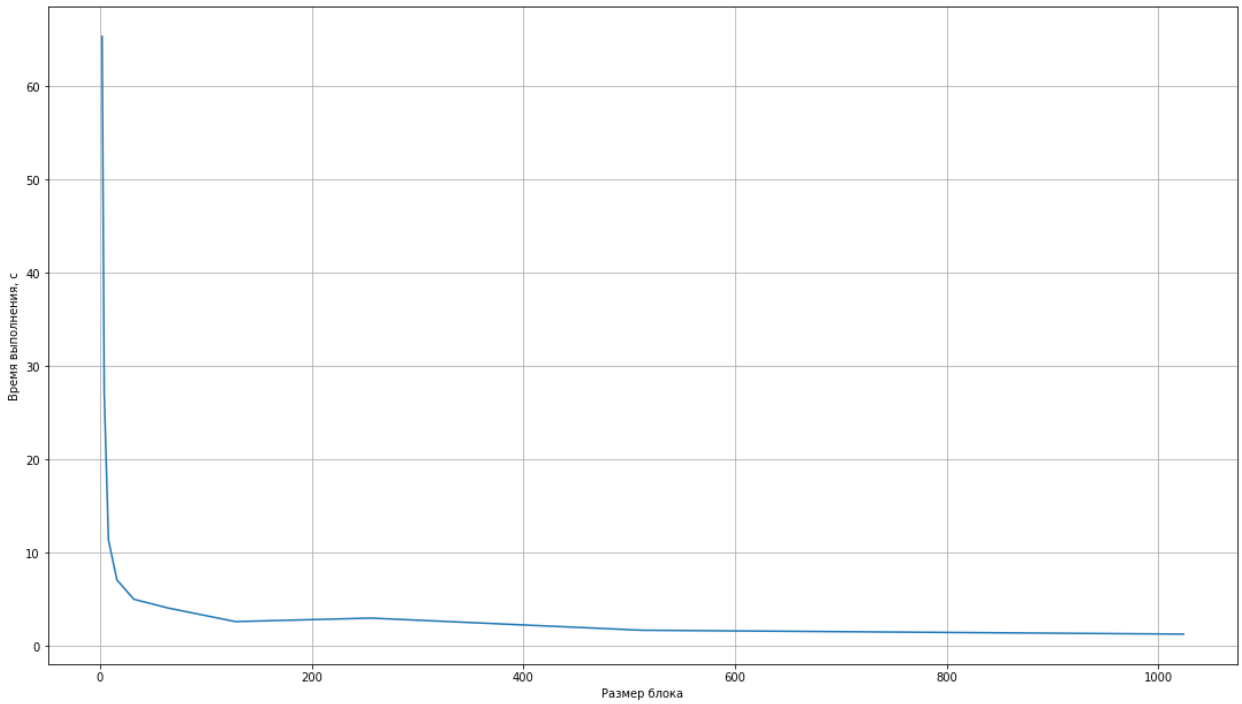

Рисунок 2 - График зависимости времени выполнения реализации блочного алгоритма от размеров блока матрицы без векторизации

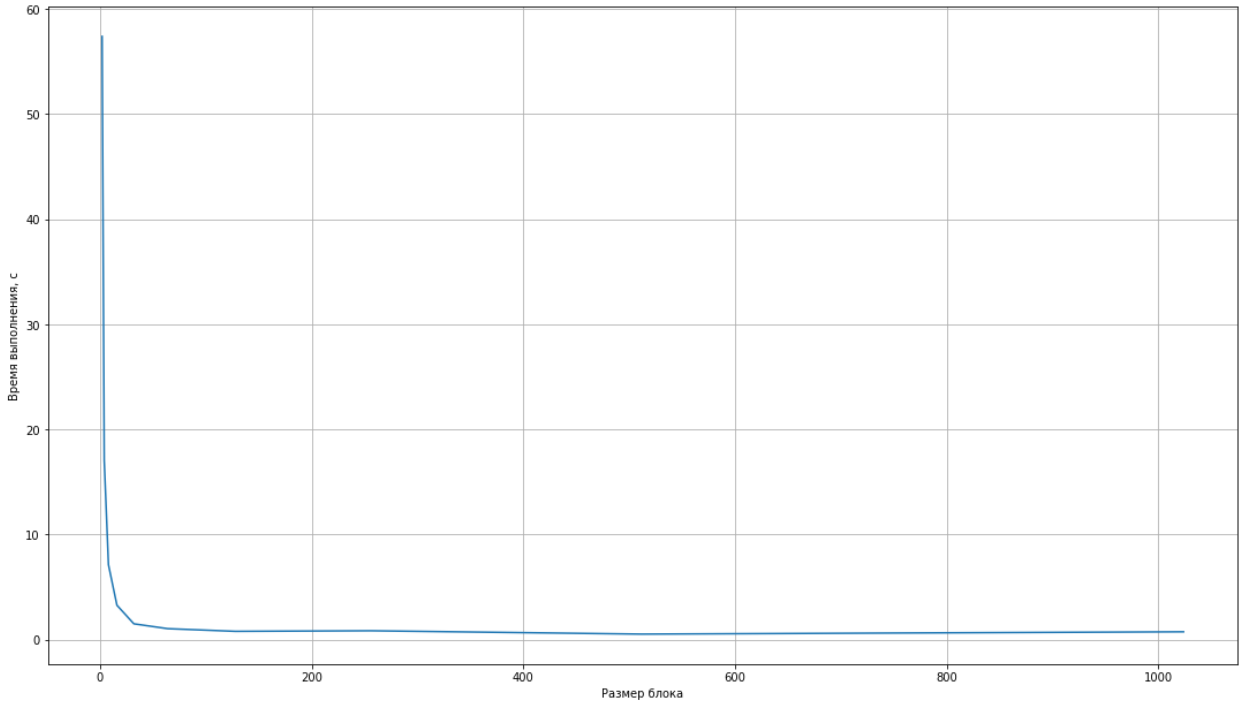

Рисунок 3 – График зависимости времени выполнения реализации блочного алгоритма от размеров блока матрицы с векторизацией

Исходя из графика 1, можно отметить что все алгоритмы работают примерно одинаковое время для матриц меньше размером чем 256×256. Для матриц большего размера обнаруживаем заметное преимущество алгоритма ijk. То же самое наблюдается для этих алгоритмов и при использовании векторизации. Также стоит отметить, что алгоритмы, выполненные с помощью векторизации, требуют меньше времени, чем алгоритмы без векторизации. Исходя из графиков 2 и 3, время выполнения блочной версии алгоритма с использованием скалярного произведения постепенно уменьшается к размеру блока 512. Также, при сравнении блочного и лучшего по времени не блочного алгоритма заметим, что без использования SSE, реализация блочного алгоритма уступает на размере матрицы 1024×1024, иначе сильно выигрывает в скорости.

Анализ результатов

Исходя из результатов эксперимента, можно увидеть, что, алгоритм из библиотеки Eigen показал один из наилучших результатов, ввиду оптимизации стандартной библиотеки для такого рода задач. Также заметим, что реализация алгоритма IJK работает быстрее JIK, ввиду того, что в C++ данные хранятся по строкам.

Как и ожидалось, что при использовании технологии SSE на больших матрицах скорость выполнения растет из-за использования векторных операций, рейтинг быстродействия реализаций алгоритмов при этом остается прежним. Также заметим, что блочный алгоритм имеет наилучшее время выполнения при блоке 512×512, что совпадает с нашими теоретическими ожиданиями и показывает наилучшее время работы среди всех реализаций.

ЗАКЛЮЧЕНИЕ

При выполнении лабораторной работы были проанализированы алгоритмы разложения матриц. Цель эксперимента в ходе лабораторной работы была достигнута. Было выяснено, что на время работы алгоритма влияют многие факторы: загруженность процессора и оперативной памяти, размер матрицы. Оптимальный блок блочного алгоритма 512×512. Из алгоритмов jik и ijk лучшее время показал ijk.

Для стандартных алгоритмов важно то, каким образом хранятся матрицы. Использование SSE модуля улучшает время выполнения программы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Бизли Д. Python Подробный справочник. –2010. – 591 С.
2. Голуб Дж. Матричные вычисления / Дж. Голуб, Ч.Ван. Лоун; пер. с англ. Ю.М. Нечепуренко, А.Ю. Романова, А.В. Собянина, Е.Е. Тыртышникова; по ред. В.В. Воеводина. – М.: Мир, 1999. – 548 с
3. Вильямс Алгоритмы. Введение в разработку и анализ — М.: 2006. 189–195. — 576 с.

Приложение А

Код программы на языке C++

```
#include <iostream>
#include <ctime>
#include <stdlib.h>
#include <math.h>
#include <Eigen/Cholesky>

using namespace std;
using namespace Eigen;

typedef Matrix<double, Dynamic, Dynamic, RowMajor> RowMatrixXi;

double** init(int);
double** transpose(double**, int);
double** choleskyIJK(double**, int);
double** choleskyJIK(double**, int);
double** cholesky(double**, int, int);
double** zeromatrix(int);
double** addition(double**, double**, int, bool);
double **solver(double**, int, int, int);
void fetch_block(double**, double**, int, int, int);
void multiply(double**, double**, double**, int);
void copyMtoM(double**, double**, int, int, int);
double** choleskyEigen(double**, int);

double** choleskyEigen(double** a, int n)
{
    clock_t start = clock();

    Map<RowMatrixXi> eig(&a[0][0], n, n);
    LLT<RowMatrixXi> ggtOfA(eig);
    MatrixXd g = ggtOfA.matrixL();

    return (double**) g.data();
}

void fetch_block(double** a, double** block, int r, int ib, int
jb)
{
    for (int i = ib; i < ib + r; i++)
    {
        for (int j = jb; j < jb + r; j++)
        {
            block[i - ib][j - jb] = a[i][j];
        }
    }
}
```

```
double** pull_block(double** a, int r, int ib, int jb)
{
    double** block = zeromatrix(r);

    for (int i = ib; i < ib + r; i++)
    {
        for (int j = jb; j < jb + r; j++)
        {
            block[i - ib][j - jb] = a[i][j];
        }
    }

    return block;
}

double** choleskyJIK(double** matrix, int n)
{

    double** lower = (double**) calloc(n, sizeof(double*));

    for(int i = 0; i < n; i++)
    {
        lower[i] = (double*) calloc(n, sizeof(double));
        for(int j = 0; j < n; j++)
        {
            lower[i][j] = matrix[i][j];
        }
    }

    double tmp;

    clock_t start = clock();

    for (int j = 0; j < n; ++j)
    {
        for (int i = 0; i < j; ++i)
        {
            for (int k = j; k< n; ++k)
            {
                lower[k][j] -=  lower[k][i] * lower[j][i];
            }
        }

        tmp = lower[j][j];

        for (int i = j; i < n; ++i)
        {
            lower[i][j] /= sqrt(tmp);
        }

    }
```

```
    return lower;
}

double** choleskyIJK(double** matrix, int n)
{

    double** lower = (double**) calloc(n, sizeof(double*));

    for(int i = 0; i < n; i++)
    {
        lower[i] = (double*) calloc(n, sizeof(double));

        for(int j = 0; j <= i; j++)
        {
            lower[i][j] = matrix[i][j];
        }
    }

    clock_t start = clock();

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

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

            int sum = 0;

            if (j == i)
            {

                for (int k = 0; k < j; k++)
                    sum += pow(lower[j][k], 2);

                lower[j][j] = sqrt(matrix[j][j] - sum);

            } else
            {
                for (int k = 0; k < j; k++)
                    sum += (lower[i][k] * lower[j][k]);

                lower[i][j] = (matrix[i][j] - sum) / lower[j]
[j];
            }

        }
    }

    return lower;
}
```

```
double** solve(double** U, int n, double** B)
{
    double** X = zeromatrix(n);
    double* b = (double*) malloc(n * sizeof(double));

    double sum;

    for (int k = 0; k < n; ++k)
    {
        sum = 0;
        for (int j = 0; j < n; ++j)
        {
            b[j] = B[k][j];
        }

        X[k][0] = b[0] / U[0][0];

        for (int i = 1; i < n; ++i)
        {
            for (int j = 0; j < i; ++j)
            {
                sum += X[k][j] * U[j][i];
            }
            X[k][i] = (b[i] - sum) / U[i][i];
        }
    }
    return X;
}

double** cholesky(double** a, int n, int r)
{
    double** s = zeromatrix(r);
    double** g = zeromatrix(n);
    double** tempo = zeromatrix(r);

    clock_t start = clock();

    for(int j = 0; j < n/r; j++)
    {
        for(int i = 0; i < n/r; i++)
        {
            fetch_block(a, s, r, i, j);

            for (int k = 0; k < j - 1; k++)
            {
                multiply(pull_block(g, r, i, k), pull_block(g,
r, k, j), tempo, r);
                tempo = addition(tempo, tempo, r, true);
            }
```

```
                s = addition(s, tempo, r, false);

                if(i == j)
                {
                    tempo = choleskyIJK(s, r);
                    copyMtoM(tempo, g, r, j, j);
                }
                else
                {
                    tempo = transpose(pull_block(g, r, j, j), r);
                    tempo = solve(s, r, tempo);
                    copyMtoM(tempo, g, r, i, j);
                }

                for(int k = 0; k < r; k++)
                {
                    for(int v = 0; v < r; v++)
                    {
                        a[i + k][j + v] = g[i + k][j + v];
                    }
                }

            }
        }

        free(g);
        free(s);
        free(tempo);

        return a;
    }

    double** init(int n)
    {

        double sum = 0;

        double** matrix = (double**) malloc(n * sizeof(double*));

        if(matrix != NULL){

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

                matrix[i] = (double*) malloc(n * sizeof(double));

                if(matrix[i] != NULL)
                {

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

```
                    matrix[i][j] = 1 + rand() % 5;
                    matrix[j][i] = matrix[i][j];
                }
            }
        }
    }

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

        for(int j = 0; j < n; j++)
        {
            if(i != j)
                sum += matrix[i][j];
        }

        matrix[i][i] = sum + 1;
    }

    return matrix;
}

double** zeromatrix(int n)
{

    double** matrix = (double**) calloc(n, n * sizeof(double*));

    for(int i = 0; i < n; i++){
        matrix[i] = (double*) calloc(n, sizeof(double));
    }

    return matrix;
}


double** transpose(double** matrix, int n)
{

    double** transposed = zeromatrix(n);

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

        for(int j = 0; j < n; j++)
            transposed[i][j] = matrix[j][i];
    }

    return transposed;
}
```

```
void copyMtoM(double** source, double** destination, int r, int
ib, int jb)
{
    for(int i = ib; i < ib + r; i++)
    {
        for(int j = jb; j < jb + r; j++)
        {
            destination[i][j] = source[i - ib][j - jb];
        }
    }
}

double** addition(double** a, double** b, int n, bool sign)
{
    double** result = zeromatrix(n);

    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < n; j++)
        {
            result[i][j] = (sign == true ? a[i][j] + b[i][j] :
a[i][j] - b[i][j]);
        }
    }

    return result;
}

void multiply(double** a, double** b, double** c, int n){

    for (int j = 0; j < n; j++)
    {
        for (int i = 0; i < n; i++)
        {
            c[i][j] = 0;

            for (int k = 0; k < n; k++)
            {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }

}
```