Университет ИТМО

Факультет ПииКТ

**Вычислительная математика**

Лабораторная работа 1

Вариант – метод Гаусса с выбором главного элемента

Выполнил: Давтян Давид Арменович

Группа: P3201

Преподаватель: Перл Ольга Вячеславовна

Санкт-Петербург, 2020г

Описание используемого метода

Метод Гаусса с выбором главного элемента является точным методом решения систем линейных алгебраических уравнений. Он заключается в том, чтобы привести матрицу с треугольному виду, чтобы под главной диагональю были нули. После этого можно очень просто вычислить все неизвестные.

Метод Гаусса релаизуют в два эапа: прямой ход и обратный ход.

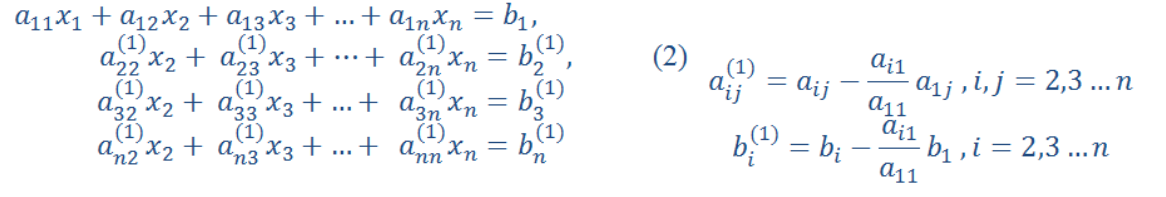
Во время прямого хода мы последовательно исключаем неизвестные из уравнений системы с помощью преобразований матрицы. Это делается путём вычитания строк друг из друга и умножения на некоторые числа.

Прямой ход прозодит до тех пор, пока матрица не станет треугольной.

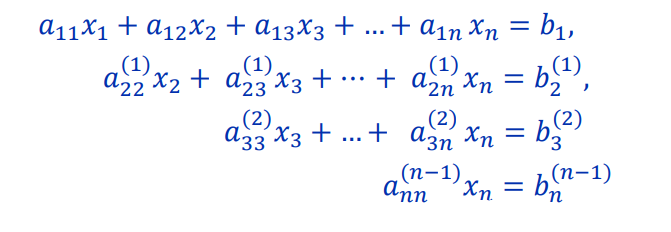
Обратный ход называется обратным, потому что мы (внезапно) движемся в обратном направлении: от n-ого уравнения к первому, последовательно вычисляя неизвестные. Сначала мы вычисляем xn из последнего уравнения, после этого мы можем вычислить предпоследнее (n-1) уравнение, потому что имеем значение последней неизвестной. Так продолжается со всеми остальными уравнениями, пока не будет найдено последнее неизвестное.

Как именно исключаются неизвестные уравнения: на первом шаге исключаем x1 из всех последующих уравнений, для этого следует умножить первое уравнение на (-a21/a11) и прибавить ко второму, затем умножить первое уравнение на (-a31/a11) и прибавить к третьему, и т.д. На втором шаге делаем те же действия, но делим не на a11, а на a22, проходя по всем уравнениям ниже a22.

Прямой ход, 1 шаг:



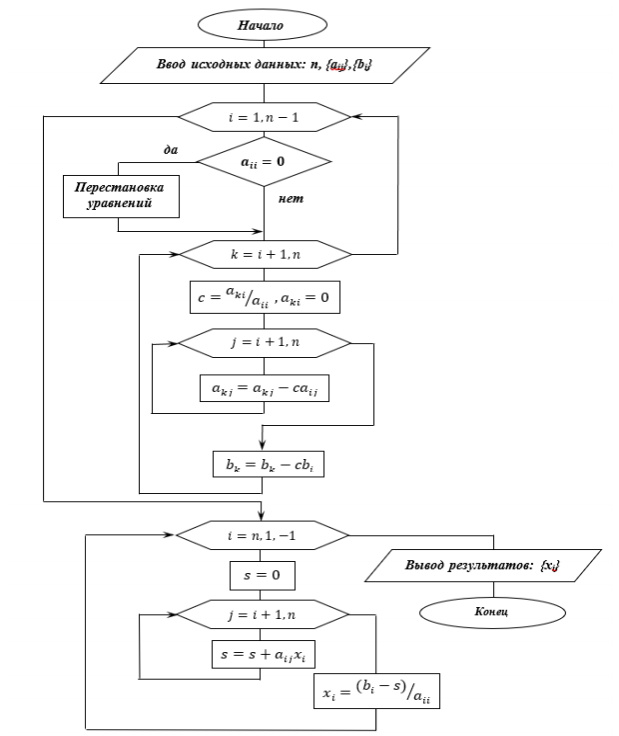
Вот что должно получиться после прямого хода:



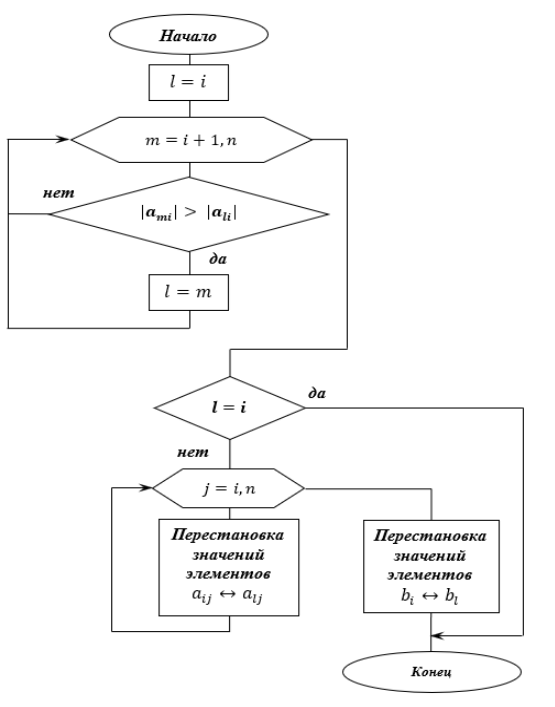
Обратный ход:

Блок-схема

Метод Гаусса



Выбор главного элемента



package ru.david.compmath;

import ru.david.compmath.matrix.Matrix;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.\*;

public class MatrixUtils {

public static double getDeterminant(Matrix matrix) {

double[][] data = matrix.getData();

if (data.length == 1)

return data[0][0];

else if (data.length == 2)

return data[0][0]\*data[1][1] - data[0][1]\*data[1][0];

else if (data.length == 3)

return data[0][0]\*data[1][1]\*data[2][2] +

data[2][0]\*data[0][1]\*data[1][2] +

data[1][0]\*data[2][1]\*data[0][2] -

data[2][0]\*data[1][1]\*data[0][2] -

data[1][0]\*data[0][1]\*data[2][2] -

data[0][0]\*data[1][2]\*data[2][1];

else if (data.length < 10) {

double result = 0;

for (int i = 0; i < data.length; i++)

result += data[i][0] == 0 ? 0 : (i % 2 == 0 ? 1 : -1) \* data[i][0] \* getDeterminant(subMatrix(matrix, i, 0));

return result;

} else {

ExecutorService threadPool = Executors.newFixedThreadPool(1);

List<Future<Double>> futures = new ArrayList<>();

double result = 0;

for (int i = 0; i < data.length; i++) {

int a = i;

if (Math.round(data[i][0]\*10000000)/10000000.0 == 0) continue;

futures.add(CompletableFuture.supplyAsync(

() -> (a % 2 == 0 ? 1 : -1) \* data[a][0] \* getDeterminant(subMatrix(matrix, a, 0)),

threadPool

));

}

for (Future<Double> future : futures) {

try {

result += future.get();

} catch (InterruptedException | ExecutionException e) {

e.printStackTrace();

}

}

return result;

}

}

public static double[] getDiscrepancy(Matrix matrix, double[] solution) {

double[][] data = matrix.getData();

double[] freeMembers = matrix.getFreeMembers();

double[] result = new double[data.length];

for (int i = 0; i < data.length; i++) {

double interResult = 0;

for (int j = 0; j < data[i].length; j++)

interResult += data[i][j] \* solution[j];

result[i] = interResult - freeMembers[i];

}

return result;

}

public static Matrix subMatrix(Matrix matrix, int rowIndexToDelete, int columnIndexToDelete) {

if (matrix.getData().length <= 1)

return new Matrix(new double[0][0], new double[0]);

double[][] data = matrix.getData();

double[] freeMembers = matrix.getFreeMembers();

double[][] newData = new double[matrix.getData().length-1][matrix.getFreeMembers().length-1];

double[] newFreeMembers = new double[matrix.getFreeMembers().length-1];

for (int row = 0; row < newData.length; row++) {

newFreeMembers[row] = freeMembers[row < rowIndexToDelete ? row : row+1];

for (int column = 0; column < newData[row].length; column++)

newData[row][column] = data

[row < rowIndexToDelete ? row : row+1]

[column < columnIndexToDelete ? column : column+1];

}

return new Matrix(newData, newFreeMembers);

}

public static double[] solveGaussian(Matrix m) {

Matrix matrix = m.clone();

makeTriangular(matrix);

if (getDeterminant(matrix) == 0)

return null;

triangularToDiagonal(matrix);

double[] result = new double[m.getData().length];

double[] freeMembers = matrix.getFreeMembers();

double[][] data = matrix.getData();

for (int i = 0; i < result.length; i++)

result[i] = freeMembers[i]/data[i][i];

return result;

}

public static int makeTriangular(Matrix matrix){

int switchRowsCount = 0;

double[][] data = matrix.getData();

for (int i = 0; i < data.length; i++) {

int mainElem = getMainElementRowIndex(matrix, i);

if (mainElem != i) {

switchRowsCount++;

switchRows(matrix, i, mainElem);

}

if (data[i][i] == 0)

continue;

for (int j = i+1; j < data.length; j++)

addRows(matrix, i, j, -1/data[i][i]\*data[j][i]);

}

return switchRowsCount;

}

public static void triangularToDiagonal(Matrix matrix) {

double[][] data = matrix.getData();

for (int i = data.length-1; i >= 0; i--) {

for (int row = i-1; row >= 0; row--) {

addRows(matrix, i, row, -1/data[i][i]\*data[row][i]);

}

}

}

public static void addRows(Matrix matrix, int rowIndex1, int rowIndex2, double multiplier) {

double[] row1 = matrix.getData()[rowIndex1];

double[] row2 = matrix.getData()[rowIndex2];

double[] freeMembers = matrix.getFreeMembers();

for (int i = 0; i < row1.length; i++)

row2[i] += multiplier \* row1[i];

freeMembers[rowIndex2] += multiplier \* freeMembers[rowIndex1];

}

public static void switchRows(Matrix matrix, int row1, int row2) {

double[][] data = matrix.getData();

double[] buffer = data[row1];

data[row1] = data[row2];

data[row2] = buffer;

double[] freeMembers = matrix.getFreeMembers();

double freeMemberBuffer = freeMembers[row1];

freeMembers[row1] = freeMembers[row2];

freeMembers[row2] = freeMemberBuffer;

}

public static int getMainElementRowIndex(Matrix matrix, int index) {

double[][] data = matrix.getData();

int result = index;

for (int i = index+1; i < data.length; i++) {

if (Math.abs(data[i][index]) > Math.abs(data[result][index]))

result = i;

}

return result;

}

}

Примеры

-1.00 3.00 -3.00 | -2.00

-4.00 2.00 -4.00 | 0.00

-1.00 3.00 -4.00 | 4.00

**Определитель**: -10.0000

**Результат**:

x1 = 3.2;

x2 = -5.6;

x3 = -6.0;

**Невязки**:

x1 = 0.00000000000000355271367880050100;

x2 = 0.00000000000000000000000000000000;

x3 = 0.00000000000000355271367880050100;

70.00 19.00 -21.00 -94.00 -37.00 -42.00 | -39.00

95.00 -96.00 16.00 31.00 4.00 15.00 | -13.00

35.00 69.00 13.00 30.00 14.00 -29.00 | -84.00

-27.00 59.00 -92.00 -27.00 89.00 33.00 | 64.00

-88.00 94.00 -35.00 -28.00 92.00 -51.00 | 44.00

33.00 55.00 73.00 25.00 -13.00 83.00 | -3.00

**Определитель**: -1384668505103.9995

**Результат**:

x1 = -0.58599;

x2 = -0.51125;

x3 = 0.24227;

x4 = -0.73148;

x5 = 0.66865;

x6 = 0.6476;

**Невязки**:

x1 = 0.00000000000000000000000000000000;

x2 = 0.00000000000001065814103640150300;

x3 = 0.00000000000000000000000000000000;

x4 = 0.00000000000001421085471520200400;

x5 = 0.00000000000000710542735760100200;

x6 = 0.00000000000002131628207280300600;

31.00 -36.00 -56.00 98.00 -10.00 | 65.00

-40.00 8.00 -26.00 -6.00 -37.00 | 18.00

-95.00 28.00 76.00 14.00 -28.00 | 29.00

39.00 94.00 -14.00 58.00 -16.00 | 21.00

-64.00 87.00 -19.00 55.00 36.00 | 93.00

**Определитель**: -7924928688.0000

**Результат**:

x1 = -0.56481;

x2 = 0.03348;

x3 = -0.36016;

x4 = 0.67646;

x5 = 0.27475;

**Невязки**:

x1 = 0.00000000000000000000000000000000;

x2 = 0.00000000000000710542735760100200;

x3 = 0.00000000000000710542735760100200;

x4 = -0.00000000000001065814103640150300;

x5 = 0.00000000000000000000000000000000;

Вывод

**Метод Гаусса с выбором главного элемента**, являющийся точным методом решения систем линейных алгебраических уравнений, который использует формулы для вычисления неизвестных, поэтому позволяет получить решение за конечное число операций. Преимуществом метода Гаусса с выбором главного элемента является универсальность – методом Гаусса можно решить подавляющее количество линейных систем. Недостатком метода Гаусса можно считать потребление памяти, и погрешности. Так как нужно хранить всю матрицу, иногда это представляет проблему, если паяти мало, а матрица большая. Так как вычисления на следующем шаге используют результаты предыдущего, происходит накапливание погрешностей. Метод Гаусса следует применять при решении систем с определителем, не близким к нулю и не разреженной матрицей, иначе эффективность алгоритма будет не такой высокой, как хотелось бы (представьте себе огромную матрицу, на 95% состоящую из нулей!).

При сравнении **метода Гаусса с выбором главного элемента** с **обычным методом Гаусса** окажется, что различий очень мало: в первом случае мы меняем местами строки таким образом, чтобы элемент стал максимальным по модулю (по столбцу или по строке), а во втором, если , то меняем местами строки таким образом, чтобы этот элемент был ненулевым.

Метод Гаусса с выбором главного элемента считается более усовершенствованным, так как с ним медленнее накапливается погрешность из-за использования относительно больших чисел (никто не любит делить околонулевое число, потому что от этого возникают большие погрешности).

Насчёт сравнения метода Гаусса с методом простой итерации и методом Гаусса-Зейделя, то последние методы являются не точными, а итерационными. Это значит, что решение ищется не по формулам, а по-другому: задается начальное приближение, а затем производятся итерации (этапы вычисления), в результате которых получают новые приближения. Так делают до тех пор, пока не будет получен результат с нужной точностью. В отличие от метода Гаусса, эти методы не требуют хранения всей матрицы, а только нескольких векторов, а также в них не накапливается погрешность, т.к. точность вычислений итераций зависит только от предыдущей итерации.