Университет ИТМО

**Факультет: ПИиКТ**

**Дисциплина: Вычислительная математика**

**Лабораторная работа №1**

Вариант “Метод простых итераций”

Выполнили: **Кудлаков Роман**

Группа: **P3231**

Преподаватель: **Перл Ольга Вячеславовна**

Санкт-Петербург, 2021

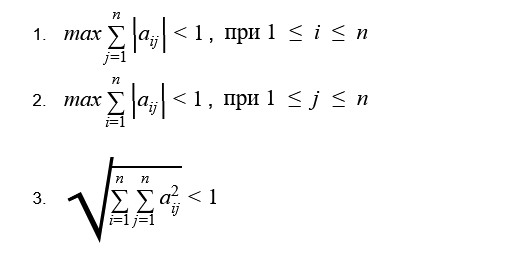
**Метод** **простой** **итерации**, называемый также **методом** последовательного приближения, - это математический алгоритм нахождения значения неизвестной величины путем постепенного ее уточнения.

Первым действием пытаемся привести матрицу к диагональному преобладанию. Для этого можно использовать перестановки строк и столбцов, а также матричные преобразования.

Если диагональное преобразование было получено, то теперь в каждой строке разделим все элементы на элемент, стоящий на диагонали.

Теперь перенесем все элементы из левой части уравнения в правую, кроме диагональных элементов.

Далее, чтобы воспользоваться методом нужно проверить выполняется ли одно из достаточных условий сходимости в правой части уравнения (столбец свободных членов не учитывается):

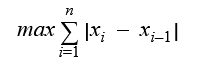


На следующем шаге обозначим наше начальное приближение присвоив ему столбец свободных членов.

Теперь подставим в правую часть каждого уравнения значение из начального приближения и получим новые значения неизвестных. Дальше будем подставлять уже новые, только что полученные значения, в уравнения.

Так будем продолжать делать до тех пор, пока не получим точность меньше, чем данная.

Расчет точности на i-том шаге:



**Листинг кода**

class **ProblemAnswer** {

public:

**ProblemAnswer**(vector<double> unknownVariables, int numOfIterations, vector<double> inaccuracy) {

       setUnknownVariables(unknownVariables);

       setNumOfIterations(numOfIterations);

       setInaccuracy(inaccuracy);

    }

**ProblemAnswer**(int numOfIterations){

       setNumOfIterations(numOfIterations);

    }

    void **setUnknownVariables**(vector<double> &unknownVariables) {

       this->unknownVariables = unknownVariables;

    }

    void **setNumOfIterations**(int numOfIterations) {

       this->numOfIterations = numOfIterations;

    }

    void **setInaccuracy**(vector<double> inaccuracy) {

       this->inaccuracy = inaccuracy;

    }

    vector<double> **getUnknownVariables**() {

       return unknownVariables;

    }

    int **getNumOfIterations**() {

       return numOfIterations;

    }

    vector<double> **getInaccuracy**() {

       return inaccuracy;

    }

    void **printAnswer**(int precision) {

       talkToUser("Number of Iterations:");

       cout << this->numOfIterations << "\n";

       talkToUser("UnknownVariables:");

       printRow(this->unknownVariables, precision);

       talkToUser("InaccuracyOfResult:");

       printRow(this->inaccuracy, precision);

    }

private:

    vector<double> unknownVariables;

    int numOfIterations;

    vector<double> inaccuracy;

};

int **findMaxAbsInRow**(vector<double> &row) {

    double max = abs(row[0]);

    int pos = 0;

    for (int i = 0; i < row.size() - 1; ++i) {

       if (max < abs(row[i])) {

           max = abs(row[i]);

           pos = i;

       }

    }

    return pos;

}

void **swapColumn**s(vector<vector<double>> &matrix, int firstColumnNum, int secondColumnNum) {

    if (firstColumnNum == secondColumnNum) return;

    for (int i = 0; i < matrix.size(); ++i) {

       double temp = matrix[i][firstColumnNum];

       matrix[i][firstColumnNum] = matrix[i][secondColumnNum];

       matrix[i][secondColumnNum] = temp;

    }

}

bool **tryToMakeDiagonalPredominance**(vector<vector<double>> &matrix) {

    for (int thisRow = 0; thisRow < matrix.size() - 1; ++thisRow) {

       int columnNum = findMaxAbsInRow(matrix[thisRow]);

       if (matrix[thisRow][columnNum] == 0 || columnNum < thisRow) return false;

       swapColumns(matrix, thisRow, columnNum);

}

    return true;

}

void **divideByDiagonal**(vector<vector<double>> &matrix) {

    for (int i = 0; i < matrix.size(); ++i) {

       for (int j = 0; j < matrix[i].size(); ++j) {

           if (i == j) { continue; }

           matrix[i][j] /= matrix[i][i];

       }

       matrix[i][i] = 0;

    }

}

double **countConvergeCoefByRows**(vector<vector<double>> &matrix) {

    double max = 0;

    for (int i = 0; i < matrix.size(); ++i) {

       double sum = 0;

       for (int j = 0; j < matrix.size(); ++j) {

           sum += matrix[i][j];

       }

       if (max < sum) max = sum;

    }

    if (max < 1) return max;

    return 0;

}

double **countConvergeCoefByColumns**(vector<vector<double>> &matrix) {

    double max = 0;

    for (int j = 0; j < matrix.size(); ++j) {

       double sum = 0;

       for (int i = 0; i < matrix.size(); ++i) {

           sum += matrix[i][j];

       }

       if (max < sum) max = sum;

    }

    if (max < 1) return max;

    return 0;

}

double **countConvergeCoefByEachValue**(vector<vector<double>> &matrix) {

    double sum = 0;

    for (int i = 0; i < matrix.size(); ++i) {

       for (int j = 0; j < matrix.size(); ++j) {

           sum += pow(matrix[i][j], 2);

       }

    }

    sum = sqrt(sum);

    if (sum < 1) return sum;

    return 0;

}

double **countConvergeCoef**(vector<vector<double>> &matrix) {

    double convergeCoef = 0;

    convergeCoef = countConvergeCoefByRows(matrix);

    if (convergeCoef) return convergeCoef;

    convergeCoef = countConvergeCoefByColumns(matrix);

    if (convergeCoef) return convergeCoef;

    convergeCoef = countConvergeCoefByEachValue(matrix);

    return convergeCoef;

}

vector<double> **makeIteration**(vector<vector<double>> &matrix, vector<double> &prevResults) {

    vector<double> results;

    for (int i = 0; i < matrix.size(); ++i) {

       double sum = 0;

       for (int j = 0; j < matrix.size(); ++j) {

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

       }

       sum = (-sum) + matrix[i][matrix.size()];

       results.push\_back(sum);

    }

    return results;

}

double **countPrecision**(vector<double> &prevResults, vector<double> &currentResults) {

    double max = abs(prevResults[0] - currentResults[0]);

    for (int i = 1; i < prevResults.size(); ++i) {

       if (max < abs(prevResults[0] - currentResults[0])) {

           max = abs(prevResults[0] - currentResults[0]);

       }

    }

    return max;

}

vector<double> **countInaccuracyVector**(vector<double> &result, vector<double> &prevStep, double convergeCoef) {

    vector<double> inaccuracyVector;

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

       double inaccuracy = convergeCoef / (1 - convergeCoef) \* abs(result[i] - prevStep[i]);

       inaccuracyVector.push\_back(inaccuracy);

    }

    return inaccuracyVector;

}

ProblemAnswer **simpleOperationMethod**(vector<vector<double>> &matrix, int precision) {

    double convergeCoef = countConvergeCoef(matrix);

    if (!convergeCoef) {

       ProblemAnswer answer(0);

       return answer;

    }

    vector<double> currentResults(matrix.size(), 0);

    for (int i = 0; i < matrix.size(); ++i) {

       currentResults[i] = matrix[i][matrix.size()];

    }

    int numOfIterations = 0;

    double currentPrecision = 2;

    vector<double> previousResults;

    while (currentPrecision >= pow(0.1, precision)) {

       previousResults = currentResults;

       currentResults = makeIteration(matrix, previousResults);

       currentPrecision = countPrecision(previousResults, currentResults);

       ++numOfIterations;

    }

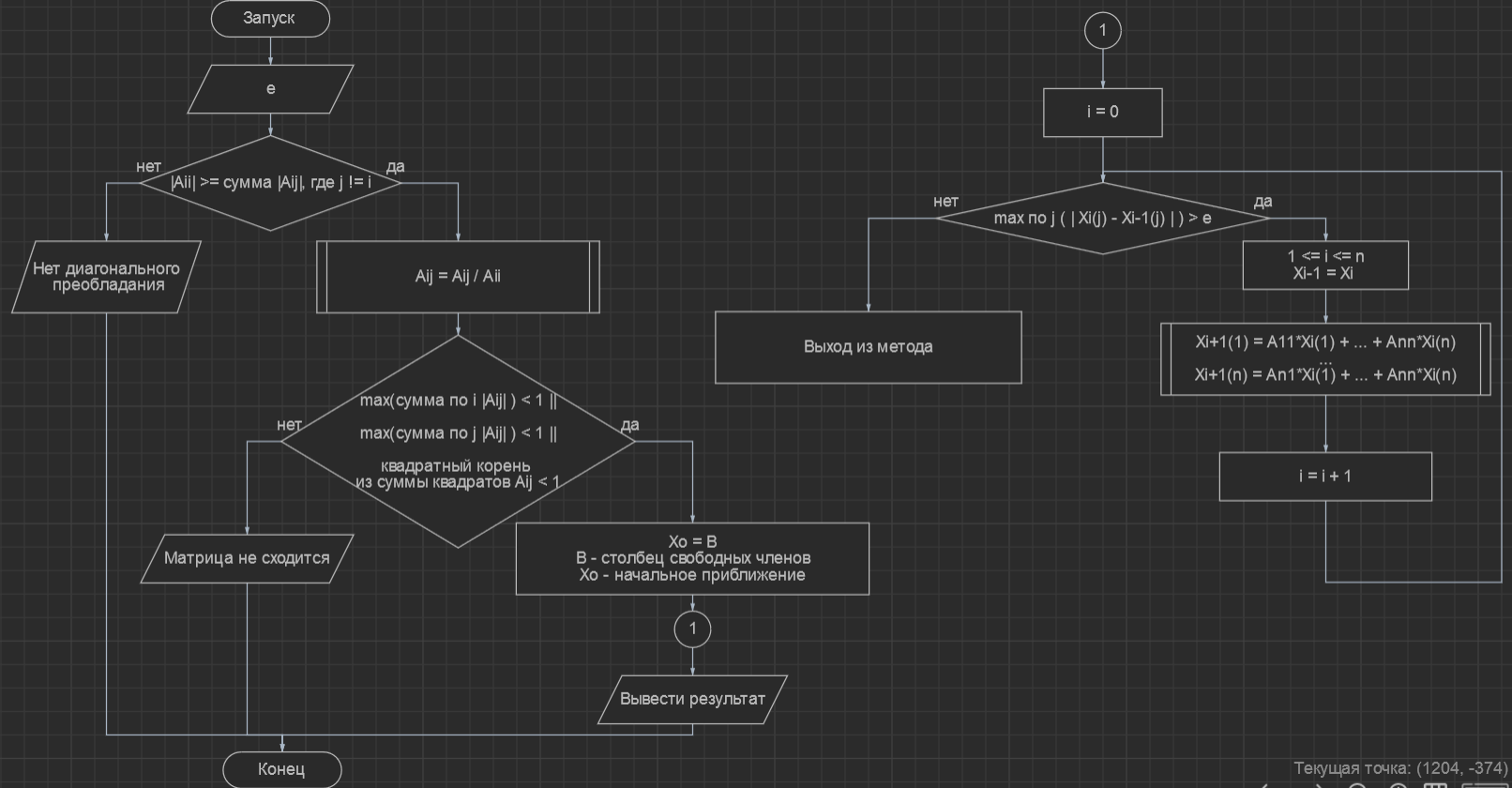
    vector<double> inaccuracy = countInaccuracyVector(currentResults, previousResults, convergeCoef);

    ProblemAnswer answer(currentResults, numOfIterations, inaccuracy);

    return answer;

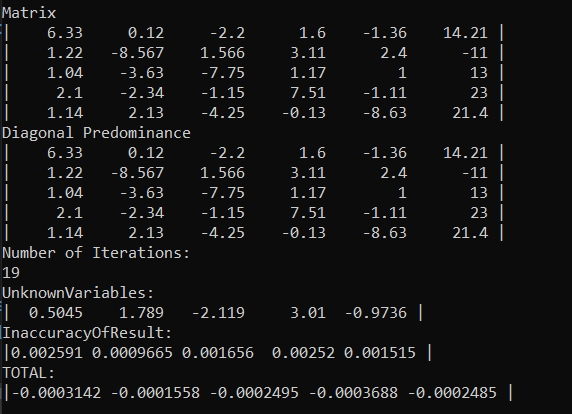
}

**Блок-схема алгоритма**



**Тесты и результаты**

**Тест 1.**

****

**Тест 2.**

****

**Вывод.**

В ходе лабораторной работы изучен метод простых итераций. Данный метод используется только для СЛАУ, в которых выполняется один из достаточных признаков сходимости. Сложность данного метода O(N2\*k), где k - количество итераций.

Плюсы данного метода в том, что можно получить результат максимально близкий к точному, но чем больше точность, тем больше итераций данному методу надо будет совершить. Также на больших размерах матриц метод простых итераций будет быстрее нежели прямые методы.

Минусы метода в том, что на маленьких размерах матрицы он работает очень долго по сравнению с прямыми методами. Обязательно должен выполняться хотя бы один признак сходимости, таким образом прямые методы могут находить решения в большем количестве матриц, чем итерационные методы.