Московский авиационный институт
(Национальный исследовательский университет)
Институт №8 «Информационные технологии и прикладная математика»
Кафедра вычислительной математики и программирования

Лабораторная работа № 1

по курсу «Численные методы».

Тема: «Методы решения задач линейной алгебры».

Студент: Вельтман Л.Я.
Группа: 80-307Б
Вариант: 7
Оценка:

Москва, 2020

Постановка задачи.

Реализовать методы для

решения СЛАУ:

1. LUP-разложение
2. Метод прогонки
3. Метод простых итераций (3.1) и метод Зейделя (3.2)

нахождения собственных векторов и собственных значений:

4. Метод вращений (Якоби)
5. QR-алгоритм

Описание методов.

1. Матрица А разлагается на произведение LU, где L – нижняя треугольная, U – верхняя треугольная. Для вычисления U выполнятся прямой ход метода Гаусса для которого вычисляются коэффициенты μ. $a_{ij}^{(k)} = a_{ij}^{(k-1)} - \mu_i^{(k)} a_{kj}^{(k-1)}, \quad \mu_i^{(k)} = \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}, \quad i = \overline{k+1,n}, \; j = \overline{k,n}.$

   Из этих коэффициентов строится матрица L.

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ \mu_2^{(1)} & 1 & 0 & 0 & 0 & 0 \\ \mu_3^{(1)} & \mu_3^{(2)} & 1 & 0 & 0 & 0 \\ \dots & \dots & \dots \mu_{k+1}^{(k)} & 1 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \mu_n^{(1)} & \mu_n^{(2)} & \mu_n^{(k)} & \mu_n^{(k+1)} \quad \dots & \mu_n^{(n-1)} & 1 \end{pmatrix}$$

   Матрица $A^{(n-1)}$ будет искомой U. Далее решается система Lz = b, а затем система Ux = z. Эти действия эквивалентны обратному ходу метода Гаусса. Время работы O(n^3).

2. А – трехдиагональная матрица. b – коэффициенты элементов главной диагонали, a – под главной, c – над, d – элементы вектора .

$$a_1 = 0 \begin{cases} b_1x_1 + c_1x_2 = d_1 \\ a_2x_1 + b_2x_2 + c_2x_3 = d_2 \\ \qquad a_3x_2 + b_3x_3 + c_3x_4 = d_3 \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \\ \qquad\qquad a_{n-1}x_{n-2} + b_{n-1}x_{n-1} + c_{n-1}x_n = d_{n-1} \\ \qquad\qquad\qquad a_nx_{n-1} + b_nx_n = d_n, \qquad c_n = 0, \end{cases}$$

   Решение будем искать в виде

$$x_i = P_i x_{i+1} + Q_i, \qquad i = \overline{1,n}$$.

В ходе прямого хода определяются прогоночные коэффициенты P и Q.

$$P_i = \frac{-c_i}{b_i + a_i P_{i-1}}, \quad Q_i = \frac{d_i - a_i Q_{i-1}}{b_i + a_i P_{i-1}}, \quad i = \overline{2,n-1};$$

(1.4)

$$P_1 = \frac{-c_1}{b_1}, \quad Q_1 = \frac{d_1}{b_1}, \text{ так как } a_1 = 0, \quad i = 1;$$

(1.5)

$$P_n = 0, \text{ т.к. } c_n = 0, \; Q_n = \frac{d_n - a_n Q_{n-1}}{b_n + a_n P_{n-1}}, \quad i = n.$$

Затем обратным ходом вычисляется искомый вектор x.

$$\begin{cases} x_n = P_n x_{n+1} + Q_n = 0 \cdot x_{n+1} + Q_n = Q_n \\ x_{n-1} = P_{n-1} x_n + Q_{n-1} \\ x_{n-1} = P_{n-2} x_{n-1} + Q_{n-2} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ x_1 = P_1 x_2 + Q_1. \end{cases}$$

Для устойчивости необходимо:

$$a_i \neq 0, \quad c_i \neq 0, \; i = \overline{2,n-1}$$

$$|b_i| \geq |a_i| + |c_i|, \qquad i = \overline{1,n},$$

И строгое неравенство достигается хотя бы при одном i. Время работы O(n).

3. (3.1) Применяется преимущественно для разреженных матриц. СЛАУ

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{cases}$$

приводится к эквивалентному виду

$$\begin{cases} x_1 = \beta_1 + \alpha_{11}x_1 + \alpha_{12}x_2 + \ldots + \alpha_{1n}x_n \\ x_2 = \beta_2 + \alpha_{21}x_1 + \alpha_{22}x_2 + \ldots + \alpha_{2n}x_n \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ x_n = \beta_n + \alpha_{n1}x_1 + \alpha_{n2}x_2 + \ldots + \alpha_{nn}x_n \end{cases}$$

Один из способов такого приведения – способ Якоби. Поменяем строки в матрице так, чтобы на диагонали не было нулей. Далее

$$\beta_i = \frac{b_i}{a_{ii}};\ \alpha_{ij} = -\frac{a_{ij}}{a_{ii}},\ i,j = \overline{1,n},\ i \neq j;\quad \alpha_{ij} = 0,\ i = j,\ i = \overline{1,n}.$$

Используя его, получаем итерационную формулу для метода простых итераций

$$\begin{cases} x^{(0)} = \beta \\ x^{(1)} = \beta + \alpha x^{(0)} \\ x^{(2)} = \beta + \alpha x^{(1)} \\ \ldots\ldots\ldots\ldots\ldots\ldots \\ x^{(k)} = \beta + \alpha x^{(k-1)}. \end{cases}$$

Оценка погрешности

$$\left\| x^{(k)} - x^* \right\| \le \varepsilon^{(k)} = \frac{\|\alpha\|}{1 - \|\alpha\|} \left\| x^{(k)} - x^{(k-1)} \right\|$$

Достаточное условие сходимости $\|\alpha\| < 1$.

(3.2) Применяется преимущественно для разреженных матриц. Является ускоренной версией предыдущего алгоритма. Ускорение достигается благодаря использованию на текущей итерации элементов вектора решения, уже вычисленных на текущей итерации (значения остальных компонент берутся из предыдущей итерации).

$$\begin{cases} x_1^{k+1} = \beta_1 + \alpha_{11} x_1^k + \alpha_{12} x_2^k + \ldots + \alpha_{1n} x_n^k \\ x_2^{k+1} = \beta_2 + \alpha_{21} x_1^{k+1} + \alpha_{22} x_2^k + \ldots + \alpha_{2n} x_n^k \\ x_3^{k+1} = \beta_3 + \alpha_{31} x_1^{k+1} + \alpha_{32} x_2^{k+1} + \alpha_{33} x_3^k + \ldots + \alpha_{3n} x_n^k \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ x_n^{k+1} = \beta_n + \alpha_{n1} x_1^{k+1} + \alpha_{n2} x_2^{k+1} + \ldots + \alpha_{nn-1} x_{n-1}^{k+1} + \alpha_{nn} x_n^k \end{cases}.$$

Матрица α представляется в виде B + C, где B – нижняя треугольная(b[i][i] == 0), C – верхняя треугольная.

Итерационная формула:

$$x^{k+1} = \beta + B\,x^{k+1} + C\,x^k$$

Достаточное условие сходимости $\|\alpha\| < 1$. Оценка погрешности

$$\varepsilon^{(k)} = \frac{\|C\|}{1 - \|\alpha\|} \left\| x^{(k)} - x^{(k-1)} \right\|$$

.

4. A – симметрическая. Ищется преобразование подобия $\Lambda = U^{-1} A U$, где Λ – диагональная. Ее диагональные элементы будут собственными значениями. Векторы матрицы U –

собственными векторами. На каждой итерации ищется максимальный по модулю элемент (его мы будем обнулять) $A^{(k)}(\left|a_{ij}^{(k)}\right|=\max_{l<m}\left|a_{lm}^{(k)}\right|)$, в соответствии с ним строится матрица вращения

$$U^k = \begin{pmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & 0 & \\ & & 1 & & & & & & \\ & & & \cos\varphi^{(k)} & \cdots & -\sin\varphi^{(k)} & \cdots & & i \\ & & & & 1 & & & & \\ & & & & & \ddots & & & \\ & & & & & & 1 & & \\ & & & \sin\varphi^{(k)} & \cdots & \cos\varphi^{(k)} & \cdots & & j \\ & & & & & & 1 & & \\ & 0 & & & & & & \ddots & \\ & & & & & & & & 1 \end{pmatrix} \begin{matrix} \\ \\ \\ i \\ \\ \\ \\ j \\ \\ \\ \end{matrix}$$

.

Угол выбирается следующим образом $\varphi^{(k)} = \frac{1}{2} arctg \frac{2a_{ij}^{(k)}}{a_{ii}^{(k)} - a_{jj}^{(k)}}$. Итерационная формула $A^{(k+1)} = U^{(k)T} A^{(k)} U^{(k)}$. Критерий останова – корень суммы квадратов поддиагональных элементов меньше заданной точности. Собственные значения – это диагональные элементы конечной матрицы A. Собственные векторы – векторы матрицы $U = U^{(0)}U^{(1)}...U^{(k)}$.

5. Ищется представление A = QR, где Q –ортогональная, R – верхняя треугольная, с помощью преобразования Хаусхолдера $H = E - \frac{2}{v^T v} v v^T$. Затем матрицы R и Q перемножаются в обратном порядке. То есть каждая итерация состоит из двух шагов A[k] = Q[k]R[k], A[k + 1] = R[k]Q[k]. Итерации продолжаются пока: для вещественных собственных значений – сумма поддиагональных элементов столбца не станет меньше или равной заданной точности, для комплексных |λ[k] – λ[k -1]| не станет меньше заданной точности. Комплексные λ находятся из решений квадратных уравнений, вещественные стоят на диагонали последней полученной матрицы A. Критерий сходимости $\left(\sum_{l=m+1}^{n}(a_{lm}^{(k)})^2\right)^{1/2} \le \varepsilon$.

Общая информация.

Данная работа состоит из 5 модулей, которые позволяют решать различные задачи решения СЛАУ и нахождения собственных векторов и собственных значений. Технические детали реализации: все алгоритмы написаны на C++.

Запуск программы.

Чтобы воспользоваться программой, необходимо скомпилировать файл main.cpp и запустить полученный исполняемый файл, например, для g++ на Windows:

g++ -std=c++11 main.cpp -o lab1

lab1.exe

При запуске программы выдается меню для выбора алгоритма.

Входные данные вводятся в консоль. Выходные данные выводятся в поток вывода.

Входные данные и результаты.

Вариант 1.

A Matrix 4x4: 1 -5 -7 1 1 -3 -9 -4 -2 4 2 1 -9 9 5 3 Vector b -75.0 -41.0 18.0 29.0	S O L U T I O N: L Matrix 4x4: 1 0 0 0 1 1 0 0 -2 -3 1 0 -9 -18 5.22222 1 U Matrix 4x4: 1 -5 -7 1 0 2 -2 -5 0 0 -18 -12 0 0 0 -15.3333 Inverse matrix: Matrix 4x4: 0.0289855 -0.0326087 0.525362 -0.228261 0.00724638 0.0543478 0.568841 -0.119565 -0.108696 -0.0652174 -0.282609 0.0434783 0.246377 -0.152174 0.34058 -0.0652174 x solution: x1 = 2 x2 = 4 x3 = 7 x4 = -8 Determinant: 552

Matrix 5x5:
15 8 0 0 0
2 -15 4 0 0
0 4 11 5 0
0 0 -3 16 -7
0 0 0 3 8
92.0 -84.0 -77.0 15.0 -11.0
// matrix
// vector
	S O L U T I O N:
	X1 = 4 X2 = 4 X3 = -8 X4 = -1 X5 = -1

Matrix 4x4: 29 8 9 -9 -7 -25 0 9 1 6 16 -2 -7 4 -2 17	SimpleIterationsMethod S O L U T I O N: x1 = 6.99975 x2 = 5.9998 x3 = -9.00018 x4 = -2.99987 Iterations: 17 ZeidelMethod S O L U T I O N: x1 = 6.99987 x2 = 5.99993 x3 = -9 x4 = -3.00003 Iterations: 7
Matrix 3x3: -6 6 -8 6 -4 9 -8 9 -2	Sobstv values: x1 = 0.770697 x2 = -19.3441 x3 = 6.57345 Iterations = 5 Sobstv vectors: x1 = 0.762047 0.62257 -0.178021 x2 = -0.595912 0.566727 -0.568955 x3 = -0.253325 0.539655 0.802869
in5.txt Matrix 3x3: 9 0 2 -6 4 4 -2 -7 5	**out5.txt** S O L U T I O N: l1 = 10.0271 l2 = 3.98646 + 6.09621i l3 = 3.98646 -6.09621i Iterations: 29

Выводы:

В ходе выполнения данной лабораторной работы я изучила новые для себя алгоритмы такие как метод прогонки, метод простых итерация и метод Зейделя, метод вращений и алгоритм QR. Алгоритм LU разложения использует в себе метод Гаусса, а с ним я познакомилась на курсе Линейной алгебры. Для себя я сделала следующие выводы: для систем с трехдиагональной матрицей коэффициентов, которые удовлетворяют условиям применения метода прогонки, лучше использовать именно его; для разреженных матриц лучше применять метод Зейделя, ведь он почти всегда быстрее метода простых итераций; во всех остальных случаях поможет метод Гаусса. Также при нахождении собственных значений для симметрических матриц выгоднее применять метод вращений Якоби, для других – QR алгоритм.

Исходный код.

```
//main.cpp
#include <iostream>
#include "matrix.hpp"
#include "lab1_1.hpp"
#include "lab1_2.hpp"
#include "lab1_3.hpp"
#include "lab1_4.hpp"
#include "lab1_5.hpp"


void help() {
   std::cout << " ========== M E N U ========== " << std::endl;
   std::cout << "Enter parameter: " << std::endl;
   std::cout << 1 << " -- to solve SLAU by LU separating matrix with Gauss method." << std::endl;
   std::cout << "Find determinant, back matrix, L and U matrix. " << std::endl;
   std::cout << 2 << " -- to solve SLAU by Thomas algorithm (progonka). " << std::endl;
   std::cout << 3 << " -- to solve SLAU by simple ittearation and Zeidel methods." << std::endl;
   std::cout << "Analyze the number of iterations which are required to achieve a given accuracy. " <<
std::endl;
   std::cout << 4 << "-- to find vectors and values of simmetric matrix by rotate method." << std::endl;
   std::cout << 5 << "-- to find values of matrix by QR-method." << std::endl;
}

int main(int argc, const char * argv[]) {

   std::cout << "LABORATORY WORK №1" << std::endl;
   std::cout << "Variant №7" << std::endl;
   help();

   int choice;
   while (std::cin >> choice) {
      if (choice == 1) {
         if (!lab1_1()) {
            std::cout << "Something wrong! Try again!\n";
         }
      }
      else if (choice == 2) {
         if (!lab1_2()) {
            std::cout << "Something wrong! Try again!\n";
         }
```

```
        }
        else if (choice == 3) {
            if (!lab1_3()) {
                std::cout << "Something wrong! Try again!\n";
            }
        }
        else if (choice == 4) {
            if (!lab1_4()) {
                std::cout << "Something wrong! Try again!\n";
            }
        }
        else if (choice == 5) {
            if (!lab1_5()) {
                std::cout << "Something wrong! Try again!\n";
            }
        }
        else {
            std::cout << "Error input!\n";
        }
        help();
    }
    return 0;
}


//matrix.hpp

#ifndef matrix_hpp
#define matrix_hpp

#include <stdio.h>
#include <iostream>
#include <vector>
#include <cmath>


class Matrix {
public:
    Matrix();
    Matrix(int32_t n, int32_t m);

    friend std::ostream& operator<<(std::ostream& os, const Matrix& matrix);
    std::vector<double>& operator[](const int32_t index);
    const std::vector<double>& operator[](const int index) const;
    friend const Matrix operator*(const Matrix& left, double rightNumber);
    friend const Matrix operator*(double leftNumber, const Matrix& right);
    friend const std::vector<double> operator*(const Matrix& left, const std::vector<double>& right);
    //friend const Matrix operator*(double leftNumber, Matrix& right);
    friend const Matrix operator*(const Matrix& left, const Matrix& right);
    friend const Matrix operator-(const Matrix& left, const Matrix& right);

    int32_t getRow() const;
```

```
    int32_t getColumn() const;
    std::vector<std::vector<double>> getMatrix();
    void getKoefficients(std::vector<std::vector<double>>&);
    double get_norm() const;
    double get_upper_norma() const;

    void unitary();
    void transpose();

    bool isSquareMatrix() const;
    bool isTridiagonalMatrix() const;
    bool isSimmetricalMatrix() const;

    ~Matrix();
private:
    std::vector<std::vector<double>> _matrix;
    int32_t row, column;
};

#endif /* matrix_hpp */

//matrix.cpp
#include "matrix.hpp"

Matrix::Matrix() {
    row = 0;
    column = 0;
}

Matrix::Matrix(int32_t n, int32_t m) {
    try {
        if (n < 0 || m < 0) {
            throw 1;
        }
        row = n;
        column = m;
        _matrix.assign(row, std::vector<double>(column, 0));
    }
    catch (int32_t i) {
        std::cout << "Error №" << i << ": size of matrix must be > 0!" << std::endl;
    }
}

std::ostream& operator<<(std::ostream& os, const Matrix& m) {
    os << "Matrix " << m.row << "x" << m.column << ": " << std::endl;
    for (int i = 0; i < m.row; ++i) {
        for (int j = 0; j < m.column; ++j) {
            os << '\t';
            os << m[i][j];
        }
        os << std::endl;
    }
```

```
    os << std::endl;
    return os;
}

std::vector<double>& Matrix::operator[](const int32_t index) {
    return _matrix[index];
}

const std::vector<double>& Matrix::operator[](const int index) const {
    return _matrix[index];
}

const Matrix operator*(const Matrix& left, double rightNumber){
    Matrix result = left;
    for(int i = 0; i < result.getRow(); ++i){
        for(int j = 0; j < result.getColumn(); ++j){
            result[i][j] *= rightNumber;
        }
    }
    return result;
}

const Matrix operator*(double leftNumber, const Matrix& right) {
    return right * leftNumber;
}

const Matrix operator*(const Matrix& left, const Matrix& right) {
    if (left.getColumn() != right.getRow()) {
        throw "Wrong sizes of matrixes to multiply!";
    }
    Matrix result(left.getRow(), right.getColumn());
    for (int i = 0; i < result.getRow(); ++i) {
        for (int j = 0; j < result.getColumn(); ++j) {
            for (int m = 0; m < left.getColumn(); ++m) {
                result[i][j] += left[i][m] * right[m][j];
            }
        }
    }
    return result;
}

const std::vector<double> operator*(const Matrix& left, const std::vector<double>& right) {
    if (left.getColumn() != right.size()) {
        throw "Wrong sizes of matrixes to multiply!";
    }
    std::vector<double> res(left.getRow(), 0.0);
    for(int i = 0; i < left.getRow(); ++i){
        for(int j = 0; j < left.getColumn(); ++j){
            res[i] += left[i][j] * right[j];
        }
    }
    return res;
```

```
}

const Matrix operator-(const Matrix& left, const Matrix& right){
   if(left.getRow() != right.getRow() || left.getColumn() != right.getColumn()){
      throw "Wrong minus! Sizes of matrix not equal!";
   }
   Matrix ans(left.getRow(), left.getColumn());
   for(int i = 0; i < ans.getRow(); ++i){
      for(int j = 0; j < ans.getColumn(); ++j){
         ans[i][j] = left._matrix[i][j] - right._matrix[i][j];
      }
   }
   return ans;
}

int32_t Matrix::getRow() const {
   return row;
}

int32_t Matrix::getColumn() const {
   return column;
}

std::vector<std::vector<double>> Matrix::getMatrix() {
   return _matrix;
}

void Matrix::getKoefficients(std::vector<std::vector<double>>& koef) {
   try {
      if (!isSquareMatrix()) {
         throw 2;
      }
      koef.assign(row, std::vector<double>(3, 0.0));

      for (int i = 0; i < row; ++i) {
         koef[i][0] = i - 1 < 0 ? 0.0 : _matrix[i][i - 1];
         koef[i][1] = _matrix[i][i];
         koef[i][2] = i + 1 == column ? 0.0 : _matrix[i][i + 1];
      }
   } catch (int32_t i) {
      std::cout << "Error №" << i << ": it's not a square matrix!" << std::endl;
      return;
   }
}


void Matrix::unitary() {
   for (int i = 0; i < row; ++i) {
      for (int j = 0; j < column; ++j) {
         _matrix[i][j] = i != j ? 0.0 : 1.0;
      }
   }
```

```
}

void Matrix::transpose() {
   std::vector<std::vector<double>> tmp(column, std::vector<double>(row));
   for (int i = 0; i < row; ++i) {
      for (int j = 0; j < column; ++j) {
         tmp[j][i] = _matrix[i][j];
      }
   }
   _matrix = tmp;
}


bool Matrix::isSquareMatrix() const {
   return row == column;
}

bool Matrix::isTridiagonalMatrix() const {
   try {
      if(!isSquareMatrix()) {
         throw 2;
      }
      for (int i = 0; i < row; ++i) {
         for (int j = 0; j < column; ++j) {
            if ((abs(i - j) > 1) && _matrix[i][j]){
               return false;
            }
         }
      }
      return true;
   }
   catch (int32_t i) {
      std::cout << "Error №" << i << ": it's not a square matrix!" << std::endl;
      return false;
   }
}

bool Matrix::isSimmetricalMatrix() const{
   try {
      if(!isSquareMatrix()) {
         throw 2;
      }
      for (int i = 0; i < row; ++i){
         for (int j = i + 1; j < column; ++j) {
            if(_matrix[i][j] != _matrix[j][i]) {
               return false;
            }
         }
      }
      return true;
   }
   catch (int32_t i) {
```

```
        std::cout << "Error №" << i << ": it's not a square matrix!" << std::endl;
        return false;
    }
}

double Matrix::get_norm() const {
    double max = 0.0;
    for (int i = 0; i < row; ++i) {
        double ans = 0.0;
        for (int j = 0; j < column; ++j) {
            ans += abs(_matrix[i][j]);
        }
        max = max > ans ? max : ans;
    }
    return max;
}

double Matrix::get_upper_norma() const{
    double max = 0.0;
    for (int i = 0; i < row; ++i){
        double ans = 0.0;
        for (int j = 0; j <= i; ++j) {
            ans += abs(_matrix[i][j]);
        }
        max = max > ans ? max : ans;
    }
    return max;
}

Matrix::~Matrix() {}

//lab1_1.hpp

#ifndef lab1_1_h
#define lab1_1_h


#include "matrix.hpp"

void NewSolutionUsingLUDecomposition(Matrix& L, Matrix& U, std::vector<double>& b,
std::vector<double>& x) {
    x.resize(U.getColumn());
    std::vector<double> z(L.getColumn());
    z[0] = b[0];
    for (int i = 1; i < L.getColumn(); ++i) {
        z[i] = b[i];
        for (int j = 0; j < i; ++j){
            z[i] -= (L[i][j] * z[j]);
        }
    }

    for (int i = U.getColumn() - 1; i >= 0; --i) {
```

```
        x[i] = z[i];
        for (int j = i + 1; j < U.getColumn(); ++j) {
            x[i] -= (x[j] * U[i][j]);
        }
        x[i] /= U[i][i];
    }
}

void Gauss(Matrix& L, Matrix& U, Matrix& matrix, std::vector<double>& b, std::vector<double>& x) {

    U = matrix;
    L = Matrix(matrix.getRow(), matrix.getColumn());
    for (int k = 0; k < U.getRow(); ++k) {
        if (U[k][k] == 0.0){
            throw "LU not exist! Try simple gauss method or enter lines in another order!";
        }
        L[k][k] = 1.0;
        for (int i = k + 1; i < U.getRow(); ++i) {
            double m_ik = U[i][k] / U[k][k];
            L[i][k] = m_ik;

            for (int j = k; j < U.getRow(); ++j) {
                U[i][j] = U[i][j] - m_ik * U[k][j];
            }
        }
    }
    NewSolutionUsingLUDecomposition(L, U, b, x);
}


void InvertMartix(Matrix& L, Matrix& U, Matrix& invA, std::vector<double>& b) {
    std::vector<double> tmp(b.size());
    for (int i = 0; i < invA.getRow(); ++i) {
        tmp[i] = 1.0;
        NewSolutionUsingLUDecomposition(L, U, tmp, invA[i]);
        tmp[i] = 0.0;
    }
    invA.transpose();
}

void InitMatrix1(Matrix& A) {
    A = Matrix(4, 4);
    A[0][0] = 1.0;
    A[0][1] = -5.0;
    A[0][2] = -7.0;
    A[0][3] = 1.0;

    A[1][0] = 1.0;
    A[1][1] = -3.0;
    A[1][2] = -9.0;
    A[1][3] = -4.0;
```

```
    A[2][0] = -2.0;
    A[2][1] = 4.0;
    A[2][2] = 2.0;
    A[2][3] = 1.0;

    A[3][0] = -9.0;
    A[3][1] = 9.0;
    A[3][2] = 5.0;
    A[3][3] = 3.0;
}

void InitVectorB1(std::vector<double>& b) {
    b.resize(4);
    b[0] = -75.0;
    b[1] = -41.0;
    b[2] = 18.0;
    b[3] = 29.0;
}

void Solution(Matrix& L, Matrix& U, Matrix& invA, std::vector<double>& x, double determinant) {
    std::cout << "S O L U T I O N:" << std::endl;
    std::cout << "L" << std::endl;
    std::cout << L << std::endl;
    std::cout << "U" << std::endl;
    std::cout << U << std::endl;
    std::cout << "Inverse matrix:" << std::endl;
    std::cout << invA << std::endl;
    std::cout << "x solution:" << std::endl;
    for (int i = 0; i < x.size(); ++i) {
        std::cout << "x" << i + 1 << " = " << x[i] << " ";
    }
    std::cout << std::endl;
    std::cout << "Determinant: " << determinant << std::endl;
}

bool lab1_1() {
    Matrix L, U, A;
    InitMatrix1(A);
    std::cout << "A" << std::endl;
    std::cout << A << std::endl;

    Matrix invA(A.getColumn(), A.getRow());

    std::vector<double> b, x;
    InitVectorB1(b);

    Gauss(L, U, A, b, x);
    Matrix ans;
    InitMatrix1(ans);
   ans = L * U;
   InvertMartix(L, U, invA, b);
    double determinant = 1.0;
```

```
    for (int i = 0; i < U.getRow(); ++i) {
        determinant *= U[i][i];
     }

    Solution(L, U, invA, x, determinant);

    return true;
}


#endif /* lab1_1_h */


//lab1_2.hpp

#ifndef lab1_2_hpp
#define lab1_2_hpp

#include "matrix.hpp"

/*
    Метод прогонки (англ. tridiagonal matrix algorithm)
    или алгоритм Томаса (англ. Thomas algorithm)
    используется для решения систем линейных уравнений вида
    Ax = F, где A — трёхдиагональная матрица.
 */

void ThomasAlgorithm(std::vector<std::vector<double>>& koef, std::vector<double>& d,
std::vector<double>& x) {
    std::vector<double> p(d.size()), q(d.size());
    p[0] = -koef[0][2] / koef[0][1];
    q[0] = d[0] / koef[0][1];
    int n = (int) x.size();
    for (int i = 1; i < n; ++i) {
        p[i] = -koef[i][2] / (koef[i][1] + koef[i][0] * p[i - 1]);
        q[i] = (d[i] - koef[i][0] * q[i - 1]) / (koef[i][1] + koef[i][0] * p[i - 1]);
    }
    //обратный ход
    x[n - 1] = q[n - 1];
    for (int i = n - 2; i >= 0; --i) {
        x[i] = p[i] * x[i + 1] + q[i];
    }
}

void InitMatrix2(Matrix& m) {
    m = Matrix(5, 5);
    m[0][0] = 15.0;
    m[0][1] = 8.0;
    m[0][2] = 0.0;
    m[0][3] = 0.0;
    m[0][4] = 0.0;
```

```
    m[1][0] = 2.0;
    m[1][1] = -15.0;
    m[1][2] = 4.0;
    m[1][3] = 0.0;
    m[1][4] = 0.0;

    m[2][0] = 0.0;
    m[2][1] = 4.0;
    m[2][2] = 11.0;
    m[2][3] = 5.0;
    m[2][4] = 0.0;

    m[3][0] = 0.0;
    m[3][1] = 0.0;
    m[3][2] = -3.0;
    m[3][3] = 16.0;
    m[3][4] = -7.0;

    m[4][0] = 0.0;
    m[4][1] = 0.0;
    m[4][2] = 0.0;
    m[4][3] = 3.0;
    m[4][4] = 8.0;
}

void InitVectorB2(std::vector<double>& b) {
    b[0] = 92.0;
    b[1] = -84.0;
    b[2] = -77.0;
    b[3] = 15.0;
    b[4] = -11.0;
}

bool lab1_2() {
    Matrix m;
    InitMatrix2(m);
    std::vector<double> b(m.getRow()), x(m.getRow(), 0.0);
    InitVectorB2(b);

    try {
        if (!m.isTridiagonalMatrix()) {
            throw 3;
        }
        std::vector<std::vector<double>> tmp;
        m.getKoefficients(tmp);
        ThomasAlgorithm(tmp, b, x);
        std::cout << "S O L U T I O N: " << std::endl;
        for (int i = 0; i < x.size(); ++i) {
            std::cout << "X" << i + 1 << " = " << x[i] << " ";
        }
        std::cout << "\n";
```

```
    } catch (int32_t i) {
        std::cout << "Error №" << i << ": matrix must be tridiagonal!" << std::endl;
        return false;
    }

    return true;
}

#endif /* lab1_2_hpp */


//lab1_3.hpp

//
//  lab1_3.hpp
//  lab1
//
//  Created by Лина Вельтман on 26/02/2020.
//  Copyright © 2020 Lina Veltman. All rights reserved.
//

#ifndef lab1_3_hpp
#define lab1_3_hpp

#include "matrix.hpp"

double norma(const std::vector<double>& m) {
    double res = 0.0;
    for (int i = 0; i < m.size(); ++i) {
        res += m[i] * m[i];
    }
    return sqrt(res);
}

std::vector<double> Substraction(std::vector<double>& x, std::vector<double>& new_x) {
    std::vector<double> vec = x;
    for (int i = 0; i < x.size(); ++i) {
        vec[i] -= new_x[i];
    }
    return vec;
}

std::vector<double> Addition(const std::vector<double>& first, const std::vector<double>& second) {
    int32_t size = (int32_t) std::max(first.size(), second.size());
    std::vector<double> res(size, 0.0);
    for (int i = 0; i < size; ++i) {
        res[i] = first[i] + second[i];
    }
    return res;
}
```

```
double Checker(std::vector<double>& x, std::vector<double>& old_x) {
   std::vector<double> vec = Substraction(x, old_x);
   return norma(vec);
}

void ToEquivalentForm(const Matrix& A, Matrix& alpha, std::vector<double>& beta) {
   if (!A.isSquareMatrix()) {
     throw "It's not a square matrix!\n";
   }
   for (int i = 0; i < alpha.getRow(); ++i) {
      if (!A[i][i]) {
         throw "Main diagonal consists 0! Swap the appropriate equation with any other equation!\n";
      }
      for (int j = 0; j < alpha.getRow(); ++j) {
         alpha[i][j] = i != j ? -A[i][j] / A[i][i] : 0.0;
      }
      beta[i] /= A[i][i];
   }
}

int32_t SimpleIterationsMethod(const Matrix A, const std::vector<double> b, std::vector<double>& x,
double accuracy) {
   Matrix alpha = A;
   x.resize(b.size());
   std::vector<double> prev_x(b.size(), 0.0),  beta = b;

   double coeff = 0.0;

   ToEquivalentForm(A, alpha, beta);

   x = beta;

   coeff = alpha.get_norm();
   if (coeff < 1.) {
      coeff /= (1 - coeff);
   } else {
      coeff = 1.0;
   }

   int32_t iter = 0;
   std::cout << coeff * Checker(x, prev_x) << std::endl;
   for (iter = 0; coeff * Checker(x, prev_x) > accuracy; ++iter) {
      prev_x = x;
      x = Addition(beta, alpha * prev_x);
   }
   return iter;
}

int32_t ZeidelMethod(const Matrix A, const std::vector<double> b, std::vector<double>& x, double
accuracy) {
   Matrix alpha = A;
   x.resize(b.size());
```

```
    std::vector<double> prev_x(b.size(), 0.0), beta = b;

    double coeff = 0.0;

    ToEquivalentForm(A, alpha, beta);

    x = beta;

    coeff = alpha.get_norm();
    if (coeff < 1.0) {
        coeff = alpha.get_upper_norma() / (1 - coeff);
    } else {
        coeff = 1.0;
    }

    int32_t iter = 0;
    for (iter = 0; coeff * Checker(x, prev_x) > accuracy; ++iter) {
        x.swap(prev_x);
        x = beta;
        for (int i = 0; i < alpha.getRow(); ++i) {
            for (int j = 0; j < i; ++j){
                x[i] += x[j] * alpha[i][j];
            }
            for (int j = i; j < alpha.getColumn(); ++j) {
                x[i] += prev_x[j] * alpha[i][j];
            }
        }
     }
    return iter;
}


void InitMatrix3(Matrix& m) {
    m = Matrix(4, 4);
    m[0][0] = 29.0;
    m[0][1] = 8.0;
    m[0][2] = 9.0;
    m[0][3] = -9.0;

    m[1][0] = -7.0;
    m[1][1] = -25.0;
    m[1][2] = 0.0;
    m[1][3] = 9.0;

    m[2][0] = 1.0;
    m[2][1] = 6.0;
    m[2][2] = 16.0;
    m[2][3] = -2.0;

    m[3][0] = -7.0;
    m[3][1] = 4.0;
    m[3][2] = -2.0;
```

```
    m[3][3] = 17.0;
}

void InitVectorB3(std::vector<double>& b) {
    b.resize(4);
    b[0] = 197.0;
    b[1] = -226.0;
    b[2] = -95.0;
    b[3] = -58.0;
}

void Answer(std::vector<double>& x, int32_t iter) {
    std::cout << "S O L U T I O N:" << std::endl;
    for (int i = 0; i < x.size(); ++i) {
        std::cout << "x" << i + 1 << " = " << x[i] << " ";
    }
    std::cout << std::endl;
    std::cout << "Iterations: " << iter << std::endl;
}

bool lab1_3() {
    double accuracy = 0.01;

    Matrix m;
    InitMatrix3(m);
    Matrix alpha = m;
    std::cout << m << std::endl;
    std::vector<double> b(m.getRow()), x;
    InitVectorB3(b);

    int iter = SimpleIterationsMethod(alpha, b, x, accuracy);
    std::cout << "SimpleIterationsMethod" << std::endl;
    Answer(x, iter);

    std::cout << "ZeidelMethod" << std::endl;
    iter = ZeidelMethod(alpha, b, x, accuracy);
    Answer(x, iter);

    return true;
}

#endif /* lab1_3_hpp */

//lab1_4.hpp

#ifndef lab1_4_hpp
#define lab1_4_hpp

#include "matrix.hpp"

int32_t RotationMethod(Matrix& A, Matrix& Ui, std::vector<double>& x, double accuracy) {
    Matrix Ak = A;
```

```
    Ui = Matrix(A.getRow(), A.getColumn());
    Ui.unitary();

    x.resize(A.getColumn());

    Matrix U(A.getRow(), A.getColumn());

    int32_t iMax = 0, jMax = 0, iter = 0;
    double condition = 0.0, phi = 0.0, max;
    const double PI = 3.141592653589793;

    if (!A.isSimmetricalMatrix()){
        throw "Matrix not simmetric! Wrong!";
    }
    while(1) {
        max = 0.0;
        iMax = 0;
        jMax = 0;
        for (int i = 0; i < Ak.getRow(); ++i) {
            for (int j = i + 1; j < Ak.getColumn(); ++j) {
                if (max < abs(Ak[i][j])) {
                    max = abs(Ak[i][j]);
                    iMax = i;
                    jMax = j;
                }
            }
        }
        U.unitary();
        phi = Ak[iMax][iMax] != Ak[jMax][jMax] ? atan(2 * Ak[iMax][jMax] / (Ak[iMax][iMax] - Ak[jMax]
[jMax])) / 2 : PI/4;
        U[iMax][jMax] = -sin(phi);
        U[jMax][iMax] = sin(phi);
        U[iMax][iMax] = cos(phi);
        U[jMax][jMax] = cos(phi);

        Ui = Ui * U;

        U.transpose();
        Ak = U * Ak;
        U.transpose();

        Ak = Ak * U;

        condition = 0;
        for (int i = 0; i < Ak.getRow(); ++i) {
            for (int j = i + 1; j < Ak.getColumn(); ++j) {
                condition += Ak[i][j] * Ak[i][j];
            }
        }
        condition = sqrt(condition);
        if (condition < accuracy) {
```

```
            break;
        }
        ++iter;
    }
    for (int i = 0; i < Ak.getRow(); ++i){
        x[i] = Ak[i][i];
    }
    return iter;
}

void InitMatrix4(Matrix& A) {
    A = Matrix(3, 3);
    A[0][0] = -6.0;
    A[0][1] = 6.0;
    A[0][2] = -8.0;

    A[1][0] = 6.0;
    A[1][1] = -4.0;
    A[1][2] = 9.0;

    A[2][0] = -8.0;
    A[2][1] = 9.0;
    A[2][2] = -2.0;
}

bool lab1_4() {
    Matrix A, U;
    InitMatrix4(A);
    std::vector<double> x;
    double accuracy = 0.001;
    int32_t iter = RotationMethod(A, U, x, accuracy);

    std::cout << "Sobstv values:" << std::endl;
    for (int i = 0; i < x.size(); ++i){
        std::cout << "x" << i + 1 << " = " << x[i] << " ";
    }
    std::cout << std::endl;
    std::cout << "Iterations = " << iter << std::endl;
    std::cout << "Sobstv vectors:" << std::endl;
    std::vector<double> vec_x(U.getColumn());
    std::vector<double> solu2(U.getColumn());
    for (int j = 0; j < U.getColumn(); ++j){
            std::cout << "x" << j + 1 << " = " << std::endl;
            for (int i = 0; i < U.getRow(); ++i){
                std::cout << U[i][j] << std::endl;
            }
          std::cout << std::endl;

    }
    for (int j = 0; j < U.getColumn(); ++j){
          std::cout << "x" << j + 1 << " = " << std::endl;
          for (int i = 0; i < U.getRow(); ++i){
```

```
            vec_x[i] = U[i][j];
          }

        std::vector<double> solu = A * vec_x;
        for (int i = 0; i < U.getColumn(); ++i) {
           solu2[i] = x[j] * vec_x[i];
        }

        for (int i = 0; i < U.getColumn(); ++i) {
           std::cout << solu[i] << " = " << solu2[i] << std::endl;
        }

       }

   return true;
}

#endif /* lab1_4_hpp */


//lab1_5.hpp

#ifndef lab1_5_hpp
#define lab1_5_hpp

#include <complex>
#include <utility>
#include "matrix.hpp"


double sign(double x) {
   if (x == 0) {
      return 0.0;
   }
   return x > 0 ? 1.0 : -1.0;
}

void QRDecompositionMethod(Matrix& A, Matrix& Q, Matrix& R) {
   Matrix H;
   Matrix E(A.getRow(), A.getColumn());
   E.unitary();
   // A = QR
   Q = E;
   R = A;
   for (int j = 0; j < A.getRow(); ++j) {
      std::vector<double> v(A.getRow(), 0.0);
      double norma = 0.0;

      v[j] = R[j][j];
      for (int i = j; i < R.getRow(); ++i) {
         norma += R[i][j] * R[i][j];
      }
```

```
        norma = sqrt(norma);
        v[j] += sign(R[j][j]) * norma;
        for (int i = j + 1; i < R.getRow(); ++i) {
            v[i] = R[i][j];
        }

        double sum1 = 0.0;
        for (int i = 0; i < v.size(); ++i) {
            sum1 += v[i] * v[i];
        }

        // Hausdorf matrix
        Matrix sum2(v.size(), v.size());
        for (int i = 0; i < sum2.getRow(); ++i) {
            for (int j = 0; j < sum2.getColumn(); ++j) {
                sum2[i][j] = v[i] * v[j];
            }
        }
        H = E - ((2 / sum1) * sum2);

        // update matrix
        Q = Q * H;
        R = H * R;
    }
}


std::pair<std::complex<double>, std::complex<double>> Roots(double a, double b, double c) {

    std::pair<std::complex<double>, std::complex<double>> res;
    std::complex<double> Discriminant = b * b - 4.0 * a * c;
    res.first = (-b + sqrt(Discriminant)) / (2 * a);
    res.second = (-b - sqrt(Discriminant)) / (2 * a);

    return res;
}


int32_t QRMethod(const Matrix& A, std::vector<std::complex<double>>& x, double accuracy) {
    Matrix Q, R, Ai = A;
    x.resize(Ai.getColumn());
    double criteria = 0.0;
    bool okay = true;
    int32_t iter = 0;

    while (okay) {
        QRDecompositionMethod(Ai, Q, R);
        Ai = R * Q;

        okay = false;
        for (int j = 0; j < Ai.getColumn(); ++j) {
            criteria = 0.0;
```

```
            for (int i = j + 1; i < Ai.getRow(); ++i) {
                criteria += Ai[i][j] * Ai[i][j];
            }
            criteria = sqrt(criteria);

            if (criteria > accuracy) {
                auto lambda = Roots(1.0, -Ai[j][j]-Ai[j+1][j+1], Ai[j][j]*Ai[j+1][j+1]-Ai[j][j+1]*Ai[j+1][j]);
                auto x1 = abs(lambda.first - x[j]);
                auto x2 = abs(lambda.second - x[j + 1]);
                if (std::max(x1, x2) > accuracy) {
                    okay = true;
                }
                x[j] = lambda.first;
                ++j;
                x[j] = lambda.second;
            } else {
                x[j] = Ai[j][j];
            }
        }
        ++iter;
    }
    return iter;
}

void InitMatrix5(Matrix& A) {
    A = Matrix(3, 3);
    A[0][0] = 9.0;
    A[0][1] = 0.0;
    A[0][2] = 2.0;

    A[1][0] = -6.0;
    A[1][1] = 4.0;
    A[1][2] = 4.0;

    A[2][0] = -2.0;
    A[2][1] = -7.0;
    A[2][2] = 5.0;
}

void matrix_init(Matrix& A, int size){
    A = Matrix(size, size);
    for(int i = 0; i < size; ++i){
        for(int j = 0; j < size; ++j){
            std::cin >> A[i][j];
        }
    }
}


bool lab1_5() {
    Matrix A;
    //InitMatrix5(A);
```

```
    matrix_init(A, 4);
    norma(A[0]);
    std::vector<std::complex<double>> x;
    double accuracy = 0.001;

    int iter = QRMethod(A, x, accuracy);

    std::cout << "S O L U T I O N:" << std::endl;
    for (int i = 0; i < x.size(); ++i) {
        std::cout << "l" << i + 1 << " = " << x[i].real() << " ";
        if (x[i].imag()) {
            if (x[i].imag() > 0) {
                std::cout << "+ ";
            }
            std::cout << x[i].imag() << "i";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
    std::cout << "Iterations: " << iter << std::endl;
    return true;
}

#endif /* lab1_5_hpp */
```