Лабораторная работа №2

Решение системы линейных уравнений методом Гаусса и итерационным методом Гаусса-Зейделя

группа Б01-818
Слынко Денис

МФТИ, 2020

Содержание

1. Постановка задачи
2. Результаты решения и данные, характеризующие задачу
3. Особенности реализации
4. Код программы

Постановка задачи

Решить методами Гаусса и Зейделя, найти min, max, определить число обусловленности матрицы $\mu = \|A\| \cdot \|A^{-1}\|$. Сделать печать невязок обоих методов. Указать критерий останова итераций метода Зейделя.

a = 20,

$$
\begin{cases}
ax_1 + x_2 + x_3 + x_4 + x_5 = 1 \\
x_1 + ax_2 + x_3 + x_4 + x_5 + x_6 = 2 \\
x_1 + x_2 + ax_3 + x_4 + x_5 + x_6 + x_7 = 3 \\
x_1 + x_2 + x_3 + ax_4 + x_5 + x_6 + x_7 + x_8 = 4 \\
x_1 + x_2 + x_3 + x_4 + ax_5 + x_6 + x_7 + x_8 + x_9 = 5 \\
x_2 + x_3 + x_4 + x_5 + ax_6 + x_7 + x_8 + x_9 + x_{10} = 6 \\
\ldots \\
x_{k-4} + x_{k-3} + x_{k-2} + x_{k-1} + ax_k + x_{k+1} + x_{k+2} + x_{k+3} + x_{k+4} = k \\
\ldots \\
x_{93} + x_{94} + x_{95} + x_{96} + ax_{97} + x_{98} + x_{99} + x_{100} = 97 \\
x_{94} + x_{95} + x_{96} + x_{97} + ax_{98} + x_{99} + x_{100} = 98 \\
x_{95} + x_{96} + x_{97} + x_{98} + ax_{99} + x_{100} = 99 \\
x_{96} + x_{97} + x_{98} + x_{99} + ax_{100} = 100
\end{cases}
$$

Результаты решения и данные, характеризующие задачу

Решение методом Гаусса:
x1 = 0.0252206
x2 = 0.066562
x3 = 0.106037
x4 = 0.143636
x5 = 0.179352
x6 = 0.214515
x7 = 0.249972
x8 = 0.285628
x9 = 0.321383
x10 = 0.357139
x11 = 0.392865
x12 = 0.428578
x13 = 0.464288
x14 = 0.499999
x15 = 0.535714
x16 = 0.571428
x17 = 0.607143
x18 = 0.642857
x19 = 0.678571
x20 = 0.714286
x21 = 0.75
x22 = 0.785714
x23 = 0.821429
x24 = 0.857143
x25 = 0.892857
x26 = 0.928571
x27 = 0.964286
x28 = 1
x29 = 1.03571
x30 = 1.07143

x31 = 1.10714
x32 = 1.14286
x33 = 1.17857
x34 = 1.21429
x35 = 1.25
x36 = 1.28571
x37 = 1.32143
x38 = 1.35714
x39 = 1.39286
x40 = 1.42857
x41 = 1.46429
x42 = 1.5
x43 = 1.53571
x44 = 1.57143
x45 = 1.60714
x46 = 1.64286
x47 = 1.67857
x48 = 1.71429
x49 = 1.75
x50 = 1.78571
x51 = 1.82143
x52 = 1.85714
x53 = 1.89286
x54 = 1.92857
x55 = 1.96429
x56 = 2
x57 = 2.03571
x58 = 2.07143
x59 = 2.10714
x60 = 2.14286
x61 = 2.17857
x62 = 2.21429
x63 = 2.25

x64 = 2.28571
x65 = 2.32143
x66 = 2.35714
x67 = 2.39286
x68 = 2.42857
x69 = 2.46429
x70 = 2.5
x71 = 2.53571
x72 = 2.57143
x73 = 2.60714
x74 = 2.64286
x75 = 2.67857
x76 = 2.71429
x77 = 2.75
x78 = 2.78571
x79 = 2.82143
x80 = 2.85714
x81 = 2.89285
x82 = 2.92857
x83 = 2.96429
x84 = 3.00002
x85 = 3.03576
x86 = 3.07146
x87 = 3.10704
x88 = 3.14241
x89 = 3.17809
x90 = 3.21462
x91 = 3.25252
x92 = 3.29232
x93 = 3.32209
x94 = 3.34176
x95 = 3.35134
x96 = 3.35085

x97 = 3.57676
x98 = 3.80424
x99 = 4.03275
x100 = 4.26177

Невязка решения методом Гаусса, взятая по норме 1: 2.77556e-17

Решение методом Гаусса-Зейделя:
x1 = 0.0252206
x2 = 0.066562
x3 = 0.106037
x4 = 0.143636
x5 = 0.179352
x6 = 0.214515
x7 = 0.249972
x8 = 0.285628
x9 = 0.321383
x10 = 0.357139
x11 = 0.392865
x12 = 0.428578
x13 = 0.464288
x14 = 0.499999
x15 = 0.535714
x16 = 0.571428
x17 = 0.607143
x18 = 0.642857
x19 = 0.678571
x20 = 0.714286
x21 = 0.75
x22 = 0.785714
x23 = 0.821429
x24 = 0.857143
x25 = 0.892857

x26 = 0.928571
x27 = 0.964286
x28 = 1
x29 = 1.03571
x30 = 1.07143
x31 = 1.10714
x32 = 1.14286
x33 = 1.17857
x34 = 1.21429
x35 = 1.25
x36 = 1.28571
x37 = 1.32143
x38 = 1.35714
x39 = 1.39286
x40 = 1.42857
x41 = 1.46429
x42 = 1.5
x43 = 1.53571
x44 = 1.57143
x45 = 1.60714
x46 = 1.64286
x47 = 1.67857
x48 = 1.71429
x49 = 1.75
x50 = 1.78571
x51 = 1.82143
x52 = 1.85714
x53 = 1.89286
x54 = 1.92857
x55 = 1.96429
x56 = 2
x57 = 2.03571
x58 = 2.07143

x59 = 2.10714
x60 = 2.14286
x61 = 2.17857
x62 = 2.21429
x63 = 2.25
x64 = 2.28571
x65 = 2.32143
x66 = 2.35714
x67 = 2.39286
x68 = 2.42857
x69 = 2.46429
x70 = 2.5
x71 = 2.53571
x72 = 2.57143
x73 = 2.60714
x74 = 2.64286
x75 = 2.67857
x76 = 2.71429
x77 = 2.75
x78 = 2.78571
x79 = 2.82143
x80 = 2.85714
x81 = 2.89285
x82 = 2.92857
x83 = 2.96429
x84 = 3.00002
x85 = 3.03576
x86 = 3.07146
x87 = 3.10704
x88 = 3.14241
x89 = 3.17809
x90 = 3.21462
x91 = 3.25252

x92 = 3.29232
x93 = 3.32209
x94 = 3.34176
x95 = 3.35134
x96 = 3.35085
x97 = 3.57676
x98 = 3.80424
x99 = 4.03275
x100 = 4.26177

Невязка решения методом Гаусса-Зейделя, взятая по норме 1: 1.38778e-17

Число обусловленности матрицы А: $\mu = 1.955004815180$

Минимальные и максимальные собственные значения матрицы итерационного процесса:

$$\lambda_{min} = 0$$

$$\lambda_{max} = 0.0129847222222978109 - 0.0066338025758074 11\,j$$

Особенности реализации

1) Код программной части реализации алгоритма Гаусса и Гаусса-Зейделя написаны на C++; вычисление собственных значений матрицы B итерационного процесса реализовано на python с помощью библиотеки numpy.linalg.
2) Для инвертирования нижней треугольной матрицы L + D в методе Гаусса-Зейделя использовалась формула, полученная из рассмотрения задачи в общем для условия $A^{-1}A = I$, где I – единичная матрица.
3) Условий останова итераций метода Гаусса-Зейделя – норма $\|u_{k+1} - u_k\| < \varepsilon$, где ε – машинный эпсилон, т.е. разница между 1.0L и следующим числом, доступным к представлению типом long double языка C++.
4) Время работы алгоритма Гаусса – 171584 мс
Время работы метода Гаусса-Зейделя – 28692 мс

Код программы

```
// main.cpp
#include <iostream>
#include "matrix.h"
#include <chrono>

int main() {
    int dim = 0;
    std::cin >> dim;
    if (dim <= 0) {
        throw std::invalid_argument("dimension must be positive");
    }
    if (dim < 9) {
        throw std::invalid_argument("dimension doesn't suit the particular problem");
    }

    Matrix<long double> A(dim, dim);
    for (size_t i = 1; i <= 4; ++i) {
        for (size_t j = 1; j <= i + 4; ++j) {
            if (i == j) {
                A.at(i, i) = 20.0L;
                A.at(dim + 1 - i, dim + 1 - i) = 20.0L;
            } else {
                A.at(i, j) = 1.0L;
                A.at(dim + 1 - i, dim + 1 - j) = 1.0L;
            }
        }
    }
    for (size_t i = 5; i <= dim - 4; ++i) {
        for (size_t j = i - 4; j <= i + 4; ++j) {
            if (i == j) {
                A.at(i, i) = 20.0L;
            } else {
                A.at(i, j) = 1.0L;
            }
        }
    }

    Matrix<long double> f(dim, 1);
    for (size_t i = 1; i <= dim; ++i) {
        f.at(i, 1) = static_cast<long double>(i);
    }

    // Condition number of the matrix A
    long double mu = condition_number(A);

    std::chrono::steady_clock::time_point begin_time = std::chrono::steady_clock::now();

    // Gaussian elimination solution
    decltype(auto) gaussian_solution = gaussian_elimination(A, f);
```

```
    decltype(auto) gaussian_residual = f - A * gaussian_solution;

    std::chrono::steady_clock::time_point end_time = std::chrono::steady_clock::now();
    std::cout << "Time elapsed for Gaussian elimination = " <<
              std::chrono::duration_cast<std::chrono::microseconds>(end_time -
begin_time).count() << "[µs]" << std::endl;

    begin_time = std::chrono::steady_clock::now();

    // Gauss-Seidel method solution
    decltype(auto) seidel_solution = gauss_seidel_method(A, f);
    decltype(auto) seidel_residual = f - A * seidel_solution;

    end_time = std::chrono::steady_clock::now();
    std::cout << "Time elapsed for Gauss-Seidel method = " <<
              std::chrono::duration_cast<std::chrono::microseconds>(end_time -
begin_time).count() << "[µs]" << std::endl;

    std::cout << "Solution with Gaussian elimination method is:" << std::endl;
    for (size_t i = 1; i <= dim; ++i) {
        std::cout << "x" << i << " = " << gaussian_solution.at(i, 1) << std::endl;
    }
    std::cout << "residal is:" << std::endl;
    for (size_t i = 1; i <= dim; ++i) {
        std::cout << "r" << i << " = " << gaussian_residual.at(i, 1) << std::endl;
    }

    std::cout << "Solution with Gauss-Seidel method is:" << std::endl;
    for (size_t i = 1; i <= dim; ++i) {
        std::cout << "x" << i << " = " << seidel_solution.at(i, 1) << std::endl;
    }
    std::cout << "residal is:" << std::endl;
    for (size_t i = 1; i <= dim; ++i) {
        std::cout << "r" << i << " = " << seidel_residual.at(i, 1) << std::endl;
    }

    std::cout << "Condition number of the matrix A is " << mu << std::endl;

    return 0;
}


// matrix.h
#pragma once

#include <algorithm>
#include <cmath>
#include <exception>
#include <iomanip>
#include <iostream>
#include <limits>
```

```cpp
#include <numeric>
#include <ostream>
#include <set>
#include <utility>
#include <vector>

const long double pi = 2.0L * std::acos(0.0L);

// TODO: zero number of rows and columns hasn't been tested
template <typename T>
class Matrix {
private:
    std::vector<T> elements_;
    size_t rows_;
    size_t columns_;

public:
    Matrix() : rows_(0), columns_(0) {};
    Matrix(size_t tmp_rows_, size_t tmp_columns_, T initial_value = static_cast<T>(0.0L)) {
        if (tmp_rows_ == 0 || tmp_columns_ == 0)
            throw std::invalid_argument("impossible to create a matrix with 0 number of columns or
rows");
        if (std::numeric_limits<size_t>::max() / tmp_rows_ <= tmp_columns_)
            throw std::invalid_argument("input dimension variables exceed size_t capacity");

        rows_ = tmp_rows_;
        columns_ = tmp_columns_;

        elements_.resize(rows_ * columns_, initial_value);
    };
    Matrix(const Matrix& other) = default;
    Matrix(Matrix&& other) noexcept = default;
    Matrix& operator=(Matrix&& other) noexcept = default;
    Matrix& operator=(const Matrix& other) = default;
    ~Matrix() noexcept = default;
    const T& at(size_t i, size_t j) const {
        if (i > rows_ || i == 0) {
            throw std::out_of_range("number of row");
        }
        if (j > columns_ || j == 0) {
            throw std::out_of_range("number of column");
        }

        return elements_[columns_ * (i - 1) + j - 1];
    };
    T& at(size_t i, size_t j) {
        if (i > rows_ || i == 0) {
            throw std::out_of_range("number of row");
        }
        if (j > columns_ || j == 0) {
            throw std::out_of_range("number of column");
```

```
        }

        return elements_[columns_ * (i - 1) + j - 1];
    };
    size_t get_rows_number() const {
        return rows_;
    };
    size_t get_columns_number() const {
        return columns_;
    };
    void swap_rows(size_t num1, size_t num2) {
        if (num1 > rows_ || num2 > rows_ || num1 == 0 || num2 == 0) {
            throw std::out_of_range("numbers of rows");
        }
        if (num1 == num2) {
            return;
        }

        std::swap_ranges(elements_.begin() + (num1 - 1) * rows_, elements_.begin() + num1 *
rows_,
                         elements_.begin() + (num2 - 1) * rows_);
    }
};

namespace staff_functions {
    template <typename T>
    T naive_determinant(const Matrix<T>& obj) {
        if (obj.get_columns_number() != obj.get_rows_number())
            throw std::invalid_argument("given matrix is not square");

        size_t dim = obj.get_columns_number();
        if (dim == 2)
            return obj.at(1, 1) * obj.at(2, 2) - obj.at(1, 2) * obj.at(2, 1);

        decltype(auto) result = static_cast<T>(0.0L);
        for (size_t i = 1; i <= dim; ++i) {
            result += std::pow(-1.0, i - 1) * obj.at(1, i) * naive_determinant(form_minor_matrix(obj,
1, i));
        }

        return result;
    }

    template <typename T>
    std::pair<Matrix<T>, int> LUP_parces(const Matrix<T>& obj) {
        if (obj.get_columns_number() != obj.get_rows_number()) {
            throw std::invalid_argument("given matrix is not square");
        }

        size_t dim = obj.get_rows_number();
        Matrix<T> C(obj);
```

```
    Matrix<T> P(dim, dim);
    for (size_t i = 1; i <= dim; ++i)
        P.at(i, i) = 1.0;

    int num_of_permutations_in_P = 0;

    for (size_t i = 1; i <= dim; i++) {
        decltype(auto) pivot = static_cast<T>(0.0L);
        size_t pv_row = 0;

        for (size_t row = i; row <= dim; ++row) {
            if (std::fabs(C.at(row, i)) > pivot) {
                pivot = std::fabs(C.at(row, i));
                pv_row = row;
            }
        }

        if (pivot != static_cast<T>(0.0L)) {
            if (pv_row != i)
                ++num_of_permutations_in_P;

            P.swap_rows(pv_row, i);
            C.swap_rows(pv_row, i);

            for (size_t j = i + 1; j <= dim; ++j) {
                C.at(j, i) /= C.at(i, i);

                const T& tmp = C.at(j, i);
                for (size_t k = i + 1; k <= dim; ++k)
                    C.at(j, k) -= tmp * C.at(i, k);
            }
        }
        else {
            throw std::invalid_argument("the matrix is singular");
        }
    }

    return std::make_pair(C, num_of_permutations_in_P);
}

template <typename T>
int is_triangular(const Matrix<T>& obj) {
    if (obj.get_rows_number() != obj.get_columns_number())
        return 0;
    size_t dim = obj.get_columns_number();

    bool flag = true;
    for (size_t i = 1; i < dim; ++i) {
        if (flag) {
            for (size_t j = i + 1; j <= dim; ++j) {
```

```
                if (obj.at(i, j) != static_cast<T>(0.0L)) {
                    flag = false;
                    break;
                }
            }
        }
    }
    if (flag) {
        return 1;
    }

    for (size_t i = 2; i <= dim; ++i) {
        for (size_t j = 1; j < i; ++j) {
            if (obj.at(i, j) != static_cast<T>(0.0L)) {
                return 0;
            }
        }
    }
    return -1;
  }
}

template <typename T>
bool operator== (const Matrix<T>& lhs, const Matrix<T>& rhs) {
  if (lhs.get_rows_number() != rhs.get_rows_number() || lhs.get_columns_number() !=
rhs.get_columns_number())
      return false;
  for (size_t i = 1, rows_number = lhs.get_rows_number(); i <= rows_number; ++i) {
      for (size_t j = 1, columns_number = lhs.get_columns_number();
           j <= columns_number; ++j) {
          if (lhs.at(i, j) != rhs.at(i, j))
              return false;
      }
  }

  return true;
}

template <typename T>
bool operator!= (const Matrix<T>& lhs, const Matrix<T>& rhs) {
  return !(lhs == rhs);
}

template <typename T>
std::ostream& operator<<(std::ostream& os, const Matrix<T>& obj) {
  size_t rows_number = obj.get_rows_number();
  size_t columns_number = obj.get_columns_number();
  for (size_t i = 1; i < rows_number; ++i) {
      for (size_t j = 1; j <= columns_number; ++j)
          os << std::fixed << std::setprecision(12) << std::setw(3) << obj.at(i, j) << " ";
      os << std::endl;
```

```
    }
    for (size_t j = 1; j < columns_number; ++j) {
        os << std::fixed << std::setprecision(12) << std::setw(3) << obj.at(rows_number, j) << "
";
    }
    os << std::fixed << std::setprecision(12) << std::setw(3) << obj.at(rows_number,
columns_number);

    return os;
}

template <typename T>
Matrix<T> operator+ (const Matrix<T>& lhs, const Matrix<T>& rhs) {
    if (lhs.get_rows_number() != rhs.get_rows_number() || lhs.get_columns_number() !=
rhs.get_columns_number())
        throw std::invalid_argument("size conflict of the input matrices");

    Matrix<T> result_matrix(lhs.get_rows_number(), lhs.get_columns_number());

    for (size_t i = 1, rows_number = lhs.get_rows_number(); i <= rows_number; ++i) {
        for (size_t j = 1, columns_number = lhs.get_columns_number();
             j <= columns_number; ++j) {
            result_matrix.at(i, j) = lhs.at(i, j) + rhs.at(i, j);
        }
    }

    return result_matrix;
}

template <typename T>
Matrix<T> operator- (const Matrix<T>& lhs, const Matrix<T>& rhs) {
    if (lhs.get_rows_number() != rhs.get_rows_number() || lhs.get_columns_number() !=
rhs.get_columns_number())
        throw std::invalid_argument("size conflict of the input matrices");

    Matrix<T> result_matrix(lhs.get_rows_number(), lhs.get_columns_number());

    for (size_t i = 1, rows_number = lhs.get_rows_number(); i <= rows_number; ++i) {
        for (size_t j = 1, columns_number = lhs.get_columns_number();
             j <= columns_number; ++j) {
            result_matrix.at(i, j) = lhs.at(i, j) - rhs.at(i, j);
        }
    }

    return result_matrix;
}

template <typename T>
Matrix<T> transpose(const Matrix<T>& other) {
    Matrix<T> result_matrix(other.get_columns_number(), other.get_rows_number());
```

```
    for (size_t i = 1, rows_number = other.get_rows_number(); i <= rows_number; ++i) {
        for (size_t j = 1, columns_number = other.get_columns_number();
             j <= columns_number; ++j) {
            result_matrix.at(j, i) = other.at(i, j);
        }
    }

    return result_matrix;
}

template <typename T>
Matrix<T> operator* (const Matrix<T>& lhs, const Matrix<T>& rhs) {
    if (lhs.get_columns_number() != rhs.get_rows_number())
        throw std::invalid_argument("size conflict of the input matrices");

    Matrix<T> result_matrix(lhs.get_rows_number(), rhs.get_columns_number());

    Matrix<T> rhs_tr = transpose(rhs);
    for (size_t i = 1, rows_number = result_matrix.get_rows_number(); i <= rows_number; ++i)
{
        for (size_t j = 1, columns_number = result_matrix.get_columns_number(); j <=
columns_number; ++j) {
            result_matrix.at(i, j) = static_cast<T>(0.0L);

            for (size_t k = 1; k <= rhs_tr.get_columns_number(); ++k)
                result_matrix.at(i, j) += lhs.at(i, k) * rhs_tr.at(j, k);
        }
    }

    return result_matrix;
}

template <typename T>
Matrix<T> operator* (const Matrix<T>& obj, const T& sqal) {
    Matrix<T> result_matrix(obj);

    for (size_t i = 1, rows_number = result_matrix.get_rows_number(); i <= rows_number; ++i)
{
        for (size_t j = 1, columns_number = result_matrix.get_columns_number(); j <=
columns_number; ++j) {
            result_matrix.at(i, j) *= sqal;
        }
    }

    return result_matrix;
}

template <typename T>
Matrix<T> operator* (const T& sqal, const Matrix<T>& obj) {
```

```
    return obj * sqal;
}

template <typename T>
Matrix<T> form_minor_matrix(const Matrix<T>& obj, size_t row_num, size_t column_num) {
    if (obj.get_rows_number() == 1 || obj.get_columns_number() == 1)
        throw std::invalid_argument("can't calculate for the inputed matrix");
    if (column_num > obj.get_columns_number() || row_num > obj.get_rows_number() ||
        column_num == 0 || row_num == 0) {
        throw std::invalid_argument("incorrect input indexes of the element");
    }

    Matrix<T> result_matrix(obj.get_rows_number() - 1, obj.get_columns_number() - 1);
    for (size_t i = 1, obj_index_row = 1, rows_number = result_matrix.get_rows_number();
         i <= rows_number; ++i, ++obj_index_row) {
        if (i == row_num) {
            ++obj_index_row;
        }

        for (size_t j = 1, obj_index_column = 1, columns_number =
result_matrix.get_columns_number();
             j <= columns_number; ++j, ++obj_index_column) {
            if (j == column_num) {
                ++obj_index_column;
            }

            result_matrix.at(i, j) = obj.at(obj_index_row, obj_index_column);
        }
    }

    return result_matrix;
}



// TODO: fix
template <typename T>
Matrix<T> inverse(const Matrix<T>& obj) {
    if (obj.get_rows_number() != obj.get_columns_number())
        throw std::invalid_argument("matrix must be square");

    Matrix<T> result_matrix(obj.get_columns_number(), obj.get_columns_number());

    decltype(auto) det_obj = static_cast<T>(0.0L);

    decltype(auto) adj = static_cast<T>(0.0L);
    for (size_t i = 1, columns_number = obj.get_columns_number(); i <= columns_number; ++i)
{
        for (size_t j = 1; j <= columns_number; ++j) {
            adj = std::pow(-1.0f, i + j) * determinant(form_minor_matrix(obj, j, i));
            result_matrix.at(i, j) = adj;
```

```
            if (i == 1) {
                det_obj += adj * obj.at(j, i);
            }
        }
    }

    if (det_obj == static_cast<T>(0.0L))
        throw std::invalid_argument("matrix is singular");

    result_matrix = result_matrix * (static_cast<T>(1.0L) / det_obj);

    return result_matrix;
}



// Delete this implementation
template <typename T>
Matrix<T> lower_triangular_inverse_slow(const Matrix<T>& obj) {
    if (staff_functions::is_triangular(obj) == 0) {
        throw std::invalid_argument("matrix must be triangular");
    }

    size_t dim = obj.get_columns_number();
    Matrix<T> result_matrix(dim, dim);

    for (size_t i = 1; i <= dim; ++i) {
        if (obj.at(i, i) == static_cast<T>(0.0L)) {
            throw std::invalid_argument("matrix is singular");
        }
        result_matrix.at(i, i) = static_cast<T>(1.0L) / obj.at(i, i);
    }

    decltype(auto) det = triangular_determinant(obj);
    for (size_t i = 2; i <= dim; ++i) {
        for (size_t j = 1; j < dim; ++j) {
            result_matrix.at(i, j) = std::pow(-1.0f, i + j) * determinant(form_minor_matrix(obj, j, i)) *
(static_cast<T>(1.0L) / det);
        }
    }

    return result_matrix;
}

// TODO: check if the matrix is lower or higher triangular
template <typename T>
Matrix<T> lower_triangular_inverse(const Matrix<T>& obj) {
    if (staff_functions::is_triangular(obj) == 0) {
        throw std::invalid_argument("matrix must be triangular");
    }
```

```cpp
    size_t dim = obj.get_columns_number();
    Matrix<T> result_matrix(dim, dim);

    for (size_t i = 1; i <= dim; ++i) {
        if (obj.at(i, i) == static_cast<T>(0.0L)) {
            throw std::invalid_argument("matrix is singular");
        }
        result_matrix.at(i, i) = static_cast<T>(1.0L) / obj.at(i, i);
    }

    for (size_t i = 2; i <= dim; ++i) {
        for (size_t j = i - 1; j > 0; --j) {
            T tmp = static_cast<T>(0.0L);
            for (size_t k = i; k >= j + 1; --k) {
                tmp += result_matrix.at(i, k) * obj.at(k, j);
            }

            result_matrix.at(i, j) = static_cast<T>(-1.0L) / obj.at(j, j) * tmp;
        }
    }

    return result_matrix;
}

template <typename T>
T determinant(const Matrix<T>& obj) {
    if (obj.get_rows_number() != obj.get_columns_number()) {
        throw std::invalid_argument("matrix must be square");
    }

    try {
        decltype(auto) LUP_res = staff_functions::LUP_parces(obj);
        decltype(auto) result = static_cast<T>(1.0L);
        for (size_t i = 1, rows_number = obj.get_rows_number(); i <= rows_number; ++i) {
            result *= LUP_res.first.at(i, i);
        }

        return result * static_cast<T>(std::pow(-1.0, LUP_res.second));
    } catch (std::invalid_argument& e) {
        return static_cast<T>(0.0L);
    }
}

template <typename T>
T triangular_determinant(const Matrix<T>& obj) {
    if (staff_functions::is_triangular(obj) == 0) {
        throw std::invalid_argument("matrix must be triangular");
    }
```

```
    size_t dim = obj.get_columns_number();
    T det = obj.at(1, 1);
    for (size_t i = 2; i <= dim; ++i) {
        det *= obj.at(i, i);
    }

    return det;
}

// TODO: move it to namespace
template <typename T>
T norm(const Matrix<T>& A) {
    auto rows_number = A.get_rows_number();
    auto columns_number = A.get_columns_number();
    std::vector<T> sums_in_rows(rows_number, static_cast<T>(0.0L));

    for (size_t i = 1; i <= rows_number; ++i) {
        for (size_t j = 1; j <= columns_number; ++j) {
            sums_in_rows[i - 1] += std::abs(A.at(i, j));
        }
    }

    return *std::max_element(sums_in_rows.begin(), sums_in_rows.end());
}

template <typename T>
T condition_number(const Matrix<T>& A) {
    return norm(A) * norm(inverse(A));
}

// TODO: move it to namespace
template <typename T, typename UnaryPredicate, typename Condition>
std::pair<size_t, size_t> find_max_element(const Matrix<T>& A, UnaryPredicate p,
                                           Condition c) {
    size_t rows_number = A.get_rows_number();
    size_t columns_number = A.get_columns_number();

    decltype(auto) max = static_cast<T>(0.0L);
    std::pair<size_t, size_t> result = std::make_pair(0, 0);
    for (size_t j = 1; j <= columns_number; ++j) {
        if (c(j)) {
            max = p(A.at(1, j));
            result = std::make_pair(1, j);
        }
    }
    if (result.first == 0) {
        throw std::logic_error("either matrix is singular or process is finished");
    }

    for (size_t i = 1; i <= rows_number; ++i) {
        for (size_t j = 1; j <= columns_number; ++j) {
```

```
            if (c(j)) {
                decltype(auto) tmp = p(A.at(i, j));
                if (tmp > max) {
                    max = tmp;
                    result = std::make_pair(i, j);
                }
            }
        }
    }

    return result;
}

// Implemented for square matrices
template <typename T>
Matrix<T> gaussian_elimination(Matrix<T> A, Matrix<T> f) {
    size_t dim = A.get_columns_number();
    if (A.get_rows_number() != dim)
        throw std::invalid_argument("matrix A must be square");
    if (f.get_columns_number() != 1)
        throw std::invalid_argument("f must be a single column");
    if (f.get_rows_number() != dim)
        throw std::invalid_argument("f and A dimensions are different");

    std::vector<size_t> leader_indexes(dim, 0);
    std::set<size_t> indexes;
    for (size_t i = 1; i <= dim; ++i) {
        decltype(auto) leader_pos = find_max_element(A, [](const T& x){
            return std::abs(x);
        }, [indexes](size_t j){
            if (indexes.find(j) != indexes.end())
                return false;
            return true;
        });
        indexes.insert(leader_pos.second);
        leader_indexes.at(leader_pos.first - 1) = leader_pos.second;
        decltype(auto) leader = A.at(leader_pos.first, leader_pos.second);

        for (size_t j = 1; j <= dim; ++j) {
            if (j != leader_pos.first) {
                decltype(auto) coef = A.at(j, leader_pos.second) / leader;

                f.at(j, 1) -= f.at(leader_pos.first, 1) * coef;
                for (size_t k = 1; k <= dim; ++k) {
                    if (k == leader_pos.second) {
                        A.at(j, k) = static_cast<T>(0.0L);
                    } else {
                        A.at(j, k) -= A.at(leader_pos.first, k) * coef;
                    }
                }
            }
```

```cpp
        }
    }

    decltype(auto) tmp_leader_indexes = leader_indexes;
    std::sort(tmp_leader_indexes.begin(), tmp_leader_indexes.end());
    std::vector<size_t> sequential_numbers(dim);
    std::iota(sequential_numbers.begin(), sequential_numbers.end(), 1);
    if (tmp_leader_indexes != sequential_numbers) {
        throw std::logic_error("either matrix is singular or smth went wrong with the algorithm");
    }

    Matrix<T> result(dim, 1);
    for (size_t i = 1; i <= dim; ++i) {
        result.at(leader_indexes.at(i - 1), 1) = f.at(i, 1) / A.at(i, leader_indexes.at(i - 1));
    }

    return result;
}

// Implemented for square matrices
template <typename T>
Matrix<T> gauss_seidel_method(const Matrix<T>& A, const Matrix<T>& f) {
    size_t dim = A.get_columns_number();
    if (A.get_rows_number() != dim)
        throw std::invalid_argument("matrix A must be square");
    if (f.get_columns_number() != 1)
        throw std::invalid_argument("f must be a single column");
    if (f.get_rows_number() != dim)
        throw std::invalid_argument("f and A dimensions are different");

    Matrix<T> D(dim, dim);
    for (size_t i = 1; i <= dim; ++i) {
        D.at(i, i) = A.at(i, i);
    }
    Matrix<T> L(dim, dim);
    for (size_t i = 1; i <= dim; ++i) {
        for (size_t j = 1; j < i; ++j) {
            L.at(i, j) = A.at(i, j);
        }
    }
    Matrix<T> U(dim, dim);
    for (size_t i = 1; i <= dim; ++i) {
        for (size_t j = dim; j > i; --j) {
            U.at(i, j) = A.at(i, j);
        }
    }

    decltype(auto) tmp = lower_triangular_inverse(L + D);
    decltype(auto) B = static_cast<T>(-1.0L) * tmp * U;
    decltype(auto) F = tmp * f;
```

```cpp
    Matrix<T> iteration_result(dim, 1, static_cast<T>(1.0L));
    tmp = B * iteration_result + F;

    while (norm(iteration_result - tmp) > std::numeric_limits<T>::epsilon()) {
        iteration_result = tmp;
        tmp = B * iteration_result + F;
    }

    return iteration_result;
}

template <typename T>
std::vector<T> jacobi_eigenvalue_algorithm(Matrix<T> A) {
    if (transpose(A) != A) {
        throw std::invalid_argument("matrix must be symmetric");
    }

    size_t dim = A.get_rows_number();
    while (true) {
        T local_max = std::abs(A.at(1, 2));
        std::pair<size_t, size_t> indexes = std::make_pair(1, 2);
        for (size_t i = 1; i < dim; ++i) {
            for (size_t j = i + 1; j <= dim; ++j) {
                T tmp = std::abs(A.at(i, j));
                if (tmp > local_max) {
                    local_max = tmp;
                    indexes = std::make_pair(i, j);
                }
            }
        }

        if (local_max < std::numeric_limits<T>::epsilon()) {
            break;
        }

        long double fi = 0.25L * pi;
        if (A.at(indexes.first, indexes.first) != A.at(indexes.second, indexes.second)) {
            fi = 0.5L * std::atan(2.0L * A.at(indexes.first, indexes.second) / (A.at(indexes.first,
indexes.first)
                                                                            - A.at(indexes.second, indexes.second)));
        }
        Matrix<T> H(dim, dim);
        for (size_t i = 1; i <= dim; ++i) {
            if (i != indexes.first && i != indexes.second) {
                H.at(i, i) = static_cast<T>(1.0L);
            }
        }
        H.at(indexes.first, indexes.first) = static_cast<T>(std::cos(fi));
        H.at(indexes.second, indexes.second) = static_cast<T>(std::cos(fi));
        H.at(indexes.second, indexes.first) = static_cast<T>(std::sin(fi));
        H.at(indexes.first, indexes.second) = static_cast<T>(-1.0L * std::sin(fi));
```

```cpp
        A = transpose(H) * A * H;
    }

    std::vector<T> result(dim, static_cast<T>(0.0L));
    for (size_t i = 1; i <= dim; ++i) {
        result.at(i - 1) = A.at(i, i);
    }

    return result;
}

template <typename T>
T operator+ (const T& lhs, const Matrix<T>& rhs) {
    if (rhs.get_columns_number() != 1 || rhs.get_rows_number() != 1)
        throw std::logic_error("can't operate");

    return lhs + rhs.at(1, 1);
}

template <typename T>
T operator- (const T& lhs, const Matrix<T>& rhs) {
    if (rhs.get_columns_number() != 1 || rhs.get_rows_number() != 1)
        throw std::logic_error("can't operate");

    return lhs - rhs.at(1, 1);
}

template <typename T>
T operator- (const Matrix<T>& lhs, const T& rhs) {
    if (lhs.get_columns_number() != 1 || lhs.get_rows_number() != 1)
        throw std::logic_error("can't operate");

    return lhs.at(1, 1) - rhs;

}
```

```python
# eigenvalues.py
import numpy as np

dim = 100
L_plus_D = np.zeros((dim, dim))
for i in range(0, dim):
    L_plus_D[i][i] = 20.0
for i in range(0, 4):
    for j in range(0, i):
        L_plus_D[i][j] = 1.0
for i in range(4, dim):
    for j in range(i - 4, i):
```

```
        L_plus_D[i][j] = 1.0

U = np.zeros((dim, dim))
for i in range(0, dim - 4):
    for j in range(i + 1, i + 5):
        U[i][j] = 1.0
for i in range(dim - 4, dim):
    for j in range(i + 1, dim):
        U[i][j] = 1.0

try:
    tmp = np.linalg.inv(L_plus_D)
except np.linalg.LinAlgError:
    print("Something went wrong with matrix initialization")
    exit(1)
else:
    B = np.multiply(np.matmul(tmp, U), -1.0)
    w, v = np.linalg.eig(B)
    np.sort(w)
    print("The condition number is {}".format(np.linalg.norm(L_plus_D + U, np.inf)
                                          * np.linalg.norm(np.linalg.inv(L_plus_D + U), np.inf)))
    print("The minimum eigenvalue is {}".format(w[0]))
    print("The maximum eigenvalue is {}".format(w[len(w) - 1]))
```