Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе**

**«Метод Гаусса»**

**Выполнил**:

студент группы 3824Б1ПМ1-1

Безносов А. С.

**Проверил**:

преподаватель каф. ВВСП,

Волокитин В.Д.

Нижний Новгород

2025

**Содержание**

[Постановка задачи 3](#_Toc26962562)

[Метод решения 4](#_Toc26962563)

[Руководство пользователя 5](#_Toc26962564)

[Описание программной реализации 7](#_Toc26962565)

[Подтверждение корректности 8](#_Toc26962566)

[Результаты экспериментов 9](#_Toc26962567)

[Заключение 11](#_Toc26962568)

[Приложение 12](#_Toc26962569)

# Постановка задачи

Нужно было реализовать метод Гаусса для решения систем линейных уравнений. Также необходимо написать два шаблонных класса: Vector и Matrix. Matrix является наследником Vector<Vector>. Для решений системы написать проверку корректности. В ходе написания программы реализовать механизм исключений (exception). Для проведения эксперимента необходимо реализовать генерацию случайной матрицы с подсчетом времени работы метода для данной матрицы и сравнением полученных данных с теоретической алгоритмической сложностью.

# Метод решения

В ходе решения были реализованы шаблонные классы Vector<T>, Matrix<T> и вспомогательный класс Solution<T>. Для Vector были написаны конструкторы по умолчанию, преобразования и копирования. Также были перегружены арифметические операции и операция “квадратные скобки” для обращения по индексу. Реализованы геттер для обращения к размеру индекса и функция Swap для смены значений между векторами. В Matrix все аналогично, отличие лишь в том, что Matrix – наследник Vector<Vector<T>>. Класс Solution хранит результаты решения СЛАУ.

# Руководство пользователя

Для работы с программой сначала необходимо определиться с типом данных, с которым производится работа, то есть выбирается либо одинарная (float), либо двойная (double) точность вычислений (рис. 1).

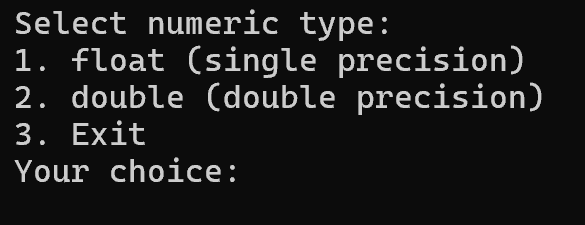


Рис. 1 Выбор типа данных

Далее пользователь выбирает: посмотреть время работы метода Гаусса через генерацию случайной матрицы или ввести собственную расширенную матрицу коэффициентов системы (рис. 2).

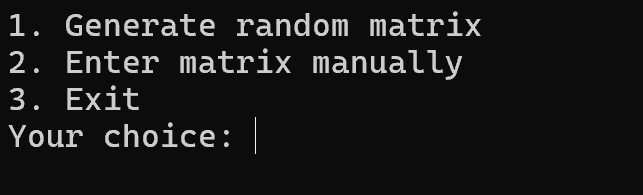


Рис.2 Выбор способа задания

**Сценарий 1 (генерация случайной матрицы)**

Пользователь вводит размер матрицы (рис. 3). После ввода размера на экран выводится сообщение об успешной (или неуспешной) работе алгоритма с выводом времени работы в секундах и информация о невязке системы (рис. 4).

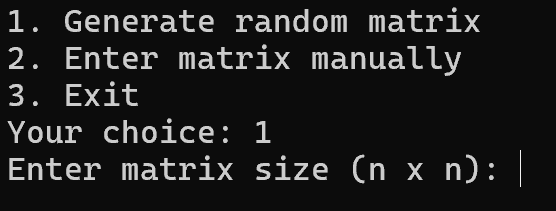


Рис. 3 Ввод размера

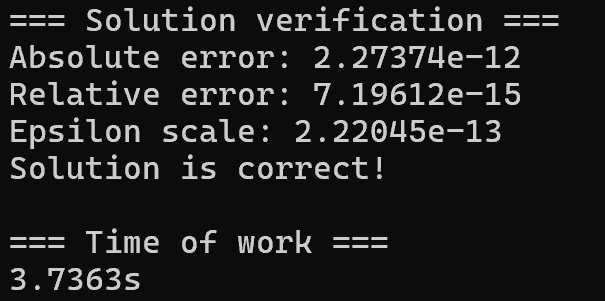


Рис. 4 Вывод результатов

**Сценарий 2 (ввод собственной матрицы)**

Пользователь вводит число уравнений и число неизвестных (рис. 5). Затем вводится расширенная матрица коэффициентов СЛАУ (рис. 6), где правые крайние элементы отвечают за столбец правой части системы. После введения всех данных выводится сообщение о результате работы алгоритма, невязка системы и её решение (рис. 7). Время работы не выводится.

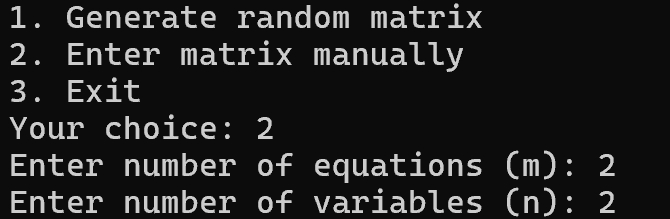


Рис. 5 Ввод числа уравнений и числа неизвестных

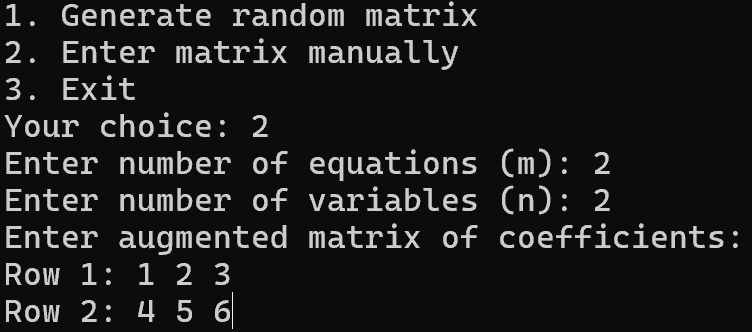


Рис. 6 Ввод матрицы коэффициентов СЛАУ

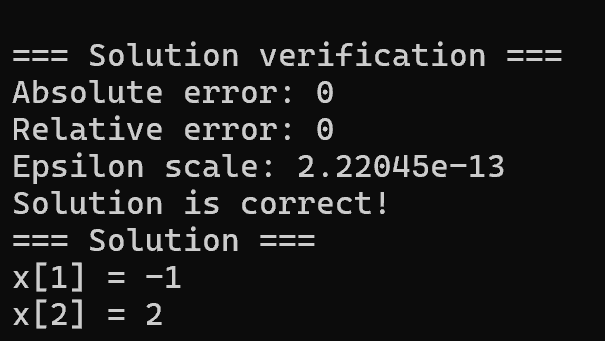


Рис. 7 вывод результатов

# Описание программной реализации

Программа содержит два файла. В файле HeaderGaussMethod.h реализована вся логика работы программы с подключением нужных библиотек. Файл mainGaussMethod.cpp содержит функцию main для запуска программы.

Функция GaussMethod представляет собой реализацию метода Гаусса. Сначала идет прямой ход: приведение к ступенчатому виду, частичный поворот для лучшей устойчивости и нормировка элементов. Далее происходит проверка на совместность системы и построение базиса в случае, если у системы нет одного единственного решения.

Функция CheckSolution проверяет корректность решения системы и выводит информацию о невязке системы. Функция PrintSolution выводит решение системы на экран.

Функция ProcessRandomMatrix генерирует на основе алгоритма “Вихрь Мерсена” случайную матрицу (диагонально – доминантную) того размера, которое введет пользователь, а ProcessManualInput дает возможность ручного ввода матрицы коэффициентов системы.

Функция Run отвечает за выбор пользователем между функционалом ProcessRandomMatrix и ProcessManualInput.

Функция FinalRun отвечает за выбор типа данных.

Прототипы функций, реализованных в программе:

Solution<T> GaussMethod(const Matrix<T>& A, const Vector<T>& b);

T MaxAbs(const Vector<T>& v);

void CheckSolution(const Matrix<T>& A, const Vector<T>& b, const Solution<T>& solution);

void PrintSolution(const Solution<T>& sol);

void ProcessRandomMatrix();

void ProcessManualInput();

void Run();

void FinalRun();

# Подтверждение корректности

Для подтверждения корректности была написана функция CheckSolution, которая проверяла, что решение системы соответствует действительности. Она суммировала произведение коэффициента при неизвестных на саму неизвестную и сравнивала полученную сумму с коэффициентом из столбца правой части. Если разница между суммой и коэффициентом была меньше некоторого фиксированного eps, то решение верное и функция выдавала сообщение об успешном решении системы.

# Результаты экспериментов

Из теоретических соображений известно, что метод Гаусса имеет сложность . Убедимся в этом, протестировав программу на различных входных данных.

Было замерено время работы алгоритма на входных данных в виде матриц с размерами от 100 до 1000. Замер времени был проведен 10 раз и из всех итераций было взято среднее значение.

На рис. 1 представлен график зависимости времени от входных данных. На рис. 2 представлен график функции , который аппроксимирует число операций в методе Гаусса. На рис. 3 представлен график отношения числа операций к времени работы алгоритма. Графики построены по таблице 1, 2 и 3 соответственно. Для большей наглядности графики (кроме рис. 3) представлены с логарифмической шкалой для оси Ox.

Таблица 1 Время работы

|  |  |
| --- | --- |
| Размер матрицы (n x n) | Среднее время (с) |
| 100 | 0,0052676 |
| 150 | 0,0162507 |
| 200 | 0,0415797 |
| 250 | 0,071154 |
| 300 | 0,120015 |
| 350 | 0,186026 |
| 400 | 0,263661 |
| 450 | 0,486491 |
| 500 | 0,642287 |
| 550 | 0,85103 |
| 600 | 1,14836 |
| 650 | 1,30958 |
| 700 | 1,57021 |
| 750 | 2,04381 |
| 800 | 2,0576 |
| 850 | 2,546 |
| 900 | 2,94032 |
| 950 | 3,44872 |
| 1000 | 4,03342 |

Рис. 2 Время работы

Таблица 2 Число операций

Рис. 2 Число операций

|  |  |
| --- | --- |
| Размер матрицы (n x n) | Число операций (n^3) |
| 100 | 1000000 |
| 150 | 3375000 |
| 200 | 8000000 |
| 250 | 15625000 |
| 300 | 27000000 |
| 350 | 42875000 |
| 400 | 64000000 |
| 450 | 91125000 |
| 500 | 125000000 |
| 550 | 166375000 |
| 600 | 216000000 |
| 650 | 274625000 |
| 700 | 343000000 |
| 750 | 421875000 |
| 800 | 512000000 |
| 850 | 614125000 |
| 900 | 729000000 |
| 950 | 857375000 |
| 1000 | 1000000000 |

Таблица 3 Отношение

|  |  |
| --- | --- |
| Размер матрицы (n x n) | O(n)/T(n) |
| 100 | 189839775,2 |
| 150 | 207683361,3 |
| 200 | 192401580,6 |
| 250 | 219594119,8 |
| 300 | 224971878,5 |
| 350 | 230478535,3 |
| 400 | 242735937,4 |
| 450 | 187310762,2 |
| 500 | 194617048,1 |
| 550 | 195498396,1 |
| 600 | 188094325,8 |
| 650 | 209704638,1 |
| 700 | 218442119,2 |
| 750 | 206415958,4 |
| 800 | 248833592,5 |
| 850 | 241211704,6 |
| 900 | 247932197,9 |
| 950 | 248606729,5 |
| 1000 | 247928556,9 |

Рис. 3 Отношение

Из рис. 3 видно, что, начиная с некоторого момента график уходит на плато, то есть прогнозы насчет времени работы оправдали себя: время работы действительно растет как . Для большей точности необходимо провести больше прогонов и взять матрицы больших размеров.

# Заключение

Был реализован метод Гаусса для решения СЛАУ. Написаны шаблонные классы Vector и Matrix – наследник Vector<Vector>. Написана проверка корректности и реализован механизм исключений. Проведен эксперимент с подсчетом времени работы алгоритма и его сравнение с теоретической алгоритмической сложностью.

# Приложение

#pragma once

#include <iostream>

#include <cmath>

#include <stdexcept>

#include <chrono>

#include <random>

#include <iomanip>

#include <limits>

template <typename T>

class Vector {

protected:

    T\* data{ nullptr };

    size\_t size{ 0 };

public:

    Vector() = default;

    Vector(size\_t n) {

        try {

            data = new T[n]();

            size = n;

        }

        catch (...) {

            throw std::runtime\_error("Failed to allocate memory.");

        }

    }

    ~Vector() { delete[] data; }

    Vector(const Vector& other): data(new T[other.size]), size(other.size) {

        try {

            for (size\_t i = 0; i < size; ++i) {

                data[i] = other.data[i];

            }

        }

        catch (...) {

            delete[] data;

            throw;

        }

    }

    size\_t GetSize() const noexcept { return size; }

    void Swap(Vector& other) noexcept;

    Vector& operator=(const Vector& other);

    T& operator[](size\_t i);

    const T& operator[](size\_t i) const;

    Vector& operator/=(T scalar);

    Vector& operator-=(const Vector& rhs);

    Vector& operator+=(const Vector& rhs);

    Vector operator\*(T scalar) const;

private:

    void CheckSize(const Vector& other) const;

};

template <typename T>

void Vector<T>::CheckSize(const Vector& other) const {

    if (size != other.size)

        throw std::invalid\_argument("Vector sizes don't match");

}

template <typename T>

Vector<T>& Vector<T>::operator=(const Vector<T>& other) {

    if (this == &other) {

        return \*this;

    }

    Vector temp(other);

    this->Swap(temp);

    return \*this;

}

template <typename T>

T& Vector<T>::operator[](size\_t i) {

    if (i >= size) {

        throw std::out\_of\_range("Vector index out of range");

    }

    return data[i];

}

template <typename T>

const T& Vector<T>::operator[](size\_t i) const {

    if (i >= size) {

        throw std::out\_of\_range("Vector index out of range");

    }

    return data[i];

}

template <typename T>

Vector<T>& Vector<T>::operator/=(T scalar) {

    if (scalar == T(0)) {

        throw std::invalid\_argument("Division by zero");

    }

    for (size\_t i = 0; i < size; ++i) {

        data[i] /= scalar;

    }

    return \*this;

}

template <typename T>

Vector<T>& Vector<T>::operator-=(const Vector& rhs) {

    CheckSize(rhs);

    for (size\_t i = 0; i < size; ++i) {

        data[i] -= rhs[i];

    }

    return \*this;

}

template <typename T>

Vector<T>& Vector<T>::operator+=(const Vector& rhs) {

    CheckSize(rhs);

    for (size\_t i = 0; i < size; ++i) {

        data[i] += rhs[i];

    }

    return \*this;

}

template <typename T>

Vector<T> Vector<T>::operator\*(T scalar) const {

    Vector res(size);

    for (size\_t i = 0; i < size; ++i) {

        res[i] = data[i] \* scalar;

    }

    return res;

}

template <typename T>

void Vector<T>::Swap(Vector& other) noexcept {

    std::swap(data, other.data);

    std::swap(size, other.size);

}

template <typename T>

class Matrix : public Vector<Vector<T>> {

    size\_t rows{ 0 };

    size\_t cols{ 0 };

public:

    Matrix() = default;

    Matrix(size\_t r, size\_t c) : Vector<Vector<T>>(r), rows(r), cols(c) {

        for (size\_t i = 0; i < rows; ++i) {

            (\*this)[i] = Vector<T>(cols);

        }

    }

    ~Matrix() = default;

    Matrix(const Matrix& other) : Vector<Vector<T>>(other), rows(other.rows), cols(other.cols) {}

    size\_t GetRows() const noexcept { return rows; }

    size\_t GetCols() const noexcept { return cols; }

    Matrix& operator=(const Matrix& other);

    Vector<T>& operator[](size\_t row);

    const Vector<T>& operator[](size\_t row) const;

    void Swap(Matrix& other) noexcept;

    void SwapRows(size\_t i, size\_t j);

};

template <typename T>

Matrix<T>& Matrix<T>::operator=(const Matrix& other) {

    if (this == &other) {

        return \*this;

    }

    Matrix temp(other);

    this->Swap(temp);

    return \*this;

}

template <typename T>

Vector<T>& Matrix<T>::operator[](size\_t row) {

    if (row >= rows) {

        throw std::out\_of\_range("Matrix row index out of range");

    }

    return Vector<Vector<T>>::operator[](row);

}

template <typename T>

const Vector<T>& Matrix<T>::operator[](size\_t row) const {

    if (row >= rows) {

        throw std::out\_of\_range("Matrix row index out of range");

    }

    return Vector<Vector<T>>::operator[](row);

}

template <typename T>

void Matrix<T>::Swap(Matrix& other) noexcept {

    Vector<Vector<T>>::Swap(other);

    std::swap(rows, other.rows);

    std::swap(cols, other.cols);

}

template <typename T>

void Matrix<T>::SwapRows(size\_t i, size\_t j) {

    if (i >= rows || j >= rows) {

        throw std::out\_of\_range("Row indices out of range");

    }

    if (i != j) {

        (\*this)[i].Swap((\*this)[j]);

    }

}

template <typename T>

class Solution {

public:

    bool is\_consistent{ false };

    bool has\_unique{ false };

    Vector<T> particular;

    Vector<T>\* basis{ nullptr };

    size\_t basis\_size{ 0 };

    size\_t\* free\_vars{ nullptr };

    size\_t rank{ 0 };

    Solution() = default;

    ~Solution() {

        delete[] basis;

        delete[] free\_vars;

    }

};

template <typename T>

Solution<T> GaussMethod(const Matrix<T>& \_A, const Vector<T>& \_b) {

    Matrix<T> A = \_A;

    Vector<T> b = \_b;

    const size\_t m = A.GetRows();

    const size\_t n = A.GetCols();

    const T eps = std::numeric\_limits<T>::epsilon() \* 1000;

    Solution<T> sol;

    sol.is\_consistent = true;

    size\_t rank = 0;

    size\_t\* pivot\_cols = new size\_t[m];

    for (size\_t i = 0; i < m; ++i) {

        pivot\_cols[i] = n;

    }

    for (size\_t col = 0; col < n && rank < m; ++col) {

        size\_t max\_row = rank;

        for (size\_t i = rank; i < m; ++i) {

            if (std::abs(A[i][col]) > std::abs(A[max\_row][col])) {

                max\_row = i;

            }

        }

        if (std::abs(A[max\_row][col]) < eps) continue;

        pivot\_cols[rank] = col;

        A.SwapRows(rank, max\_row);

        std::swap(b[rank], b[max\_row]);

        T pivot = A[rank][col];

        A[rank] /= pivot;

        b[rank] /= pivot;

        for (size\_t i = rank + 1; i < m; ++i) {

            T factor = A[i][col];

            if (std::abs(A[i][col]) > eps) {

                A[i] -= A[rank] \* factor;

                b[i] -= b[rank] \* factor;

            }

        }

        ++rank;

    }

    for (size\_t i = rank; i < m; ++i) {

        if (std::abs(b[i]) > eps) {

            sol.is\_consistent = false;

            delete[] pivot\_cols;

            return sol;

        }

    }

    sol.rank = rank;

    sol.has\_unique = (rank == n);

    sol.particular = Vector<T>(n);

    bool\* is\_pivot = new bool[n]();

    for (size\_t i = 0; i < rank; ++i) {

        if (pivot\_cols[i] < n) {

            is\_pivot[pivot\_cols[i]] = true;

        }

    }

    for (ptrdiff\_t i = rank - 1; i >= 0; --i) {

        size\_t col = pivot\_cols[i];

        if (col >= n) continue;

        sol.particular[col] = b[i];

        for (size\_t j = col + 1; j < n; ++j) {

            sol.particular[col] -= A[i][j] \* sol.particular[j];

        }

    }

    sol.basis\_size = 0;

    for (size\_t j = 0; j < n; ++j) {

        if (!is\_pivot[j]) ++sol.basis\_size;

    }

    if (sol.basis\_size > 0) {

        sol.basis = new Vector<T>[sol.basis\_size];

        sol.free\_vars = new size\_t[sol.basis\_size];

        size\_t basis\_idx = 0;

        for (size\_t free\_col = 0; free\_col < n; ++free\_col) {

            if (!is\_pivot[free\_col]) {

                sol.free\_vars[basis\_idx] = free\_col;

                sol.basis[basis\_idx] = Vector<T>(n);

                sol.basis[basis\_idx][free\_col] = 1;

                for (ptrdiff\_t i = rank - 1; i >= 0; --i) {

                    size\_t pivot\_col = pivot\_cols[i];

                    if (pivot\_col >= n) continue;

                    sol.basis[basis\_idx][pivot\_col] = 0;

                    for (size\_t j = pivot\_col; j < n; ++j) {

                        sol.basis[basis\_idx][pivot\_col] -= A[i][j] \* sol.basis[basis\_idx][j];

                    }

                }

                ++basis\_idx;

            }

        }

    }

    delete[] is\_pivot;

    delete[] pivot\_cols;

    return sol;

}

template <typename T>

T MaxAbs(const Vector<T>& v) {

if (v.GetSize() == 0) {

throw std::runtime\_error("Cannot find max of empty vector!");

}

T max = std::abs(v[0]);

for (size\_t i = 1; i < v.GetSize(); ++i) {

T curr = std::abs(v[i]);

if (curr > max) {

max = curr;

}

}

if (max <= 0) {

return 0;

}

return max;

}

template <typename T>

void CheckSolution(const Matrix<T>& A, const Vector<T>& b, const Solution<T>& sol) {

const T eps = std::numeric\_limits<T>::epsilon() \* 1000;

Vector<T> res(b.GetSize());

if (!sol.is\_consistent) {

std::cout << "Solution is correct!\n";

return;

}

for (size\_t i = 0; i < A.GetRows(); ++i) {

T sum = 0;

for (size\_t j = 0; j < A.GetCols(); ++j) {

sum += A[i][j] \* sol.particular[j];

}

res[i] = std::abs(sum - b[i]);

}

T max\_error = MaxAbs(res);

T max\_b = MaxAbs(b);

std::cout << "\n=== Solution verification ===\n";

std::cout << "Absolute error: " << max\_error << "\n";

std::cout << "Relative error: " << max\_error / max\_b << "\n";

std::cout << "Epsilon scale: " << eps << "\n";

if (max\_error > eps \* std::max(T(1), max\_b)) {

throw std::runtime\_error("Solution is numerically unstable!");

}

std::cout << "Solution is correct!\n";

}

template <typename T>

void PrintSolution(const Solution<T>& sol) {

    const T eps = std::numeric\_limits<T>::epsilon() \* 100;

    if (!sol.is\_consistent) {

        std::cout << "System is inconsistent (no solutions)\n";

        return;

    }

    std::cout << "Solution:\n";

    if (sol.has\_unique) {

        for (size\_t i = 0; i < sol.particular.GetSize(); ++i) {

            std::cout << "x[" << i + 1 << "] = " << sol.particular[i] << "\n";

        }

    }

    else {

        for (size\_t i = 0; i < sol.particular.GetSize(); ++i) {

            std::cout << "x[" << i + 1 << "] = " << sol.particular[i];

            for (size\_t k = 0; k < sol.basis\_size; ++k) {

                if (std::abs(sol.basis[k][i]) > eps) {

                    std::cout << " + " << sol.basis[k][i] << "\*t" << k + 1;

                }

            }

            std::cout << "\n";

        }

        for (size\_t k = 0; k < sol.basis\_size; ++k) {

            std::cout << "t" << k + 1 << " is free (x[" << sol.free\_vars[k] + 1 << "])\n";

        }

    }

}

template <typename T>

void ProcessRandomMatrix() {

    size\_t n;

    std::cout << "Enter matrix size (n x n): ";

    std::cin >> n;

    if (n == 0) {

        throw std::invalid\_argument("Matrix size must be positive");

    }

    std::random\_device rd;

    std::mt19937 gen(rd());

    T max\_val = static\_cast<T>(10000.0 / std::sqrt(n));

    std::uniform\_real\_distribution<T> dist(-max\_val, max\_val);

    Matrix<T> A(n, n);

    Vector<T> b(n);

    for (size\_t i = 0; i < n; ++i) {

        T row\_sum = 0;

        for (size\_t j = 0; j < n; ++j) {

            if (i != j) {

                A[i][j] = dist(gen);

                row\_sum += std::abs(A[i][j]);

            }

        }

        A[i][i] = row\_sum;

        b[i] = dist(gen);

    }

    const auto start{ std::chrono::steady\_clock::now() };

    Solution<T> sol = GaussMethod(A, b);

    const auto finish{ std::chrono::steady\_clock::now() };

    const std::chrono::duration<double> elapsed\_seconds{ finish - start };

    CheckSolution(A, b, sol);

    std::cout << "Time of work: ";

    std::cout << elapsed\_seconds.count() << "s\n";

}

template <typename T>

void ProcessManualInput() {

    size\_t m, n;

    std::cout << "Enter number of equations (m): ";

    std::cin >> m;

    std::cout << "Enter number of variables (n): ";

    std::cin >> n;

    if (m == 0 || n == 0) {

        throw std::invalid\_argument("Matrix size must be positive");

    }

    Matrix<T> A(m, n);

    Vector<T> b(m);

    std::cout << "Enter augmented matrix of coefficients:\n";

    for (size\_t i = 0; i < m; ++i) {

        std::cout << "Row " << i + 1 << ": ";

        for (size\_t j = 0; j < n; ++j) {

            if (!(std::cin >> A[i][j])) {

                throw std::invalid\_argument("Invalid input!");

            }

        }

        if (!(std::cin >> b[i])) {

            throw std::invalid\_argument("Invalid input!");

        }

    }

    Solution<T> sol = GaussMethod(A, b);

    CheckSolution(A, b, sol);

    PrintSolution(sol);

}

enum Choice {

    RandomMatrix = 1,

    ManualInput = 2,

    Exit = 3

};

template <typename T>

void Run() {

    std::cout << "1. Generate random matrix\n";

    std::cout << "2. Enter matrix manually\n";

    std::cout << "3. Exit\n";

    std::cout << "Your choice: ";

    size\_t choice;

    std::cin >> choice;

    switch (choice) {

    case RandomMatrix:

        ProcessRandomMatrix<T>();

        break;

    case ManualInput:

        ProcessManualInput<T>();

        break;

    case Exit:

        std::cout << "Exiting program...\n";

        return;

    default:

        throw std::invalid\_argument("Invalid choice!");

    }

    std::cout << std::endl;

}

enum TypeChoice {

    FloatType = 1,

    DoubleType = 2

};

void FinalRun() {

    std::cout << "Select numeric type:\n";

    std::cout << "1. float (single precision)\n";

    std::cout << "2. double (double precision)\n";

    std::cout << "3. Exit\n";

    std::cout << "Your choice: ";

    size\_t type\_choice;

    std::cin >> type\_choice;

    switch (type\_choice) {

    case FloatType:

        Run<float>();

        break;

    case DoubleType:

        Run<double>();

        break;

    case Exit:

        std::cout << "Exiting program...\n";

        break;

    default:

        throw std::invalid\_argument("Invalid choice!");

    }

}