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

Котельников И. Е.

**Проверил**:

преподаватель каф. ВВСП,

Волокитин В.Д.

Нижний Новгород

2025

**Содержание**

[Постановка задачи 3](#_Toc28931)

[Метод решения 4](#_Toc10712)

[Руководство пользователя 5](#_Toc20898)

[Описание программной реализации 8](#_Toc12948)

[Подтверждение корректности 9](#_Toc22842)

[Результаты экспериментов 10](#_Toc20872)

[Заключение 12](#_Toc2508)

[Приложение 13](#_Toc2983)

# Постановка задачи

Реализовать метод Гаусса для решения систем линейных уравнений. Программа должна включать в себя два шаблона класса: Vector и Matrix, последний должен являться наследником вектора векторов. Сделать свои exception. Написать проверку корректности результата работы программы. Эксперименты проводить с использованием генератора случайных совместных матриц. Посчитать время работы метода для подсчета сложности алгоритма.

# Метод решения

Реализованы шаблонные классы Vector, Matrix и класс Solution, содержащий решение. Написаны конструкторы копирования и преобразования, также есть конструктор по умолчанию для класса Vector. Были перегружены арифметические операции, присваивание, “квадратные скобки” для обращения по индексу. Реализованы методы получения длины вектора и обмена значений векторов. Аналогичные операции выполнены для класса Matrix, с учетом того, что он является наследником вектора векторов.

# Руководство пользователя

На старте программы пользователю предлагается выбрать тип данных, от которого зависит точность работы программы, одинарную (float) или двойную (double). Также есть возможность завершить работу программы, выбрать пункт 3. (рис. 1).

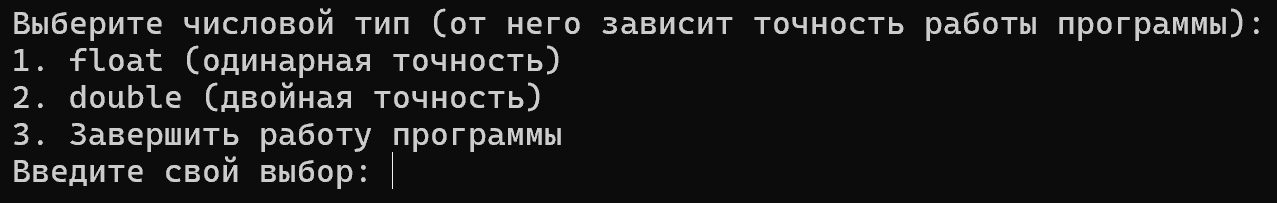
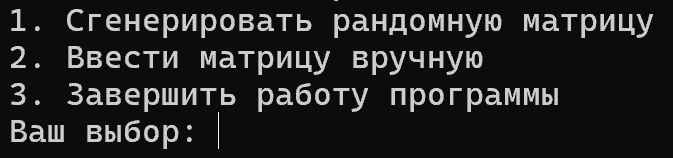


Рис. 1 Программа предлагает выбрать тип данных

Далее программа просит пользователя выбрать, хочет ли он ввести матрицу самостоятельно (расширенную матрицу коэффициентов) или сгенерировать ее автоматически. Также есть возможность завершить работу программы. (рис. 2).



**Рис.2 Программа предлагает выбрать способ расширенной матрицы коэффициентов**

**Вариант 1 (матрицы генерируется случайно)**

Пользователю необходимо ввести одно число, которое будет размером матрицы (рис. 3). После ввода числа программа сообщает пользователю об успешной (или неудачной) работе алгоритма, после чего выводит время его работы в секундах и результат своей работы (рис. 4).

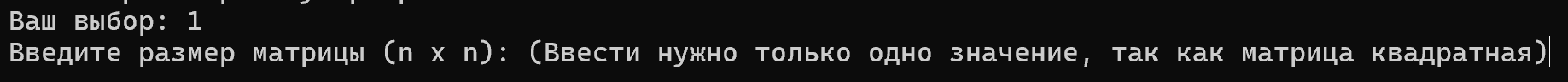


Рис. 3 Программа просит пользователя ввести размер матрицы

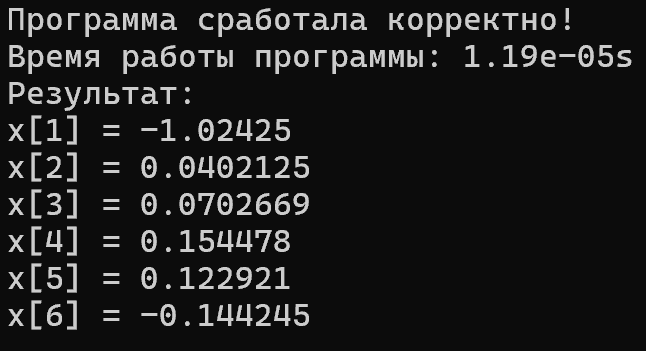


Рис. 4 Программа выводит результат

**Случай 2 (пользователь вводит матрицу самостоятельно)**

Пользователь вводит число уравнений и число неизвестных (рис. 5). После этого он вводит расширенную матрицу коэффициентов СЛАУ (рис. 6). Следует заметить, что крайний правый элемент в строке (их на 1 больше, чем переменных) отвечает за соответствующий номеру строки элемент вектора b. После введения данных последней строки выводится результат работы программы (рис. 7). В этом случае программа время работы не выводит.

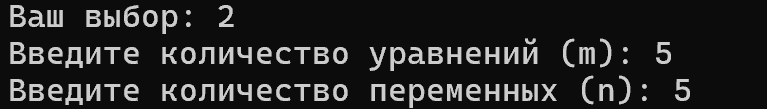


Рис. 5 Программа просит ввести число уравнений и число переменных

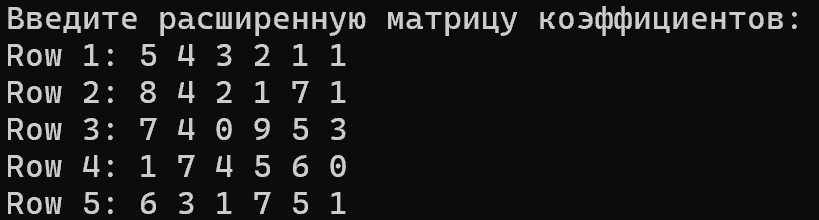


Рис. 6 Непосредственно ввод пользователем матрицы коэффициентов

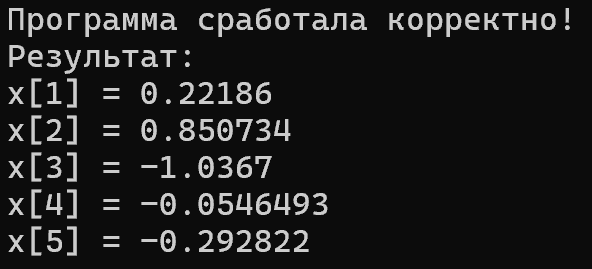


Рис. 7 Выводятся результаты

# Описание программной реализации

В функции GaussMethod реализован сам метод Гаусса. Матрица приводится к треугольному виду, нормируются элементы матрицы, система проверяется на совместность (если несовместна, то функция завершает свою работу). Если система не имеет единственного решения, то собирается базис.

Функции CheckSolution и PrintSolution отвечают соответственно за проверку корректности работы программы и вывод результата ее работы.

Функция ProcessRandomMatrix отвечает за генерацию случайной матрицы, размера, введенного пользователем, а ProcessManualInput за ручной ввод матрицы коэффициентов.

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

Функция main отвечает за запуск программы.

Прототипы функций, реализованных в программе:

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

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();

# Подтверждение корректности

Как было написано выше, функция CheckSolution отвечает за проверку корректности результата работы программы. Она подставляет значения переменных и умножает их на стоящие перед ними коэффициенты, после чего сравнивает полученное слева значение с вектором справа. В случае, когда различие между двумя векторами оказывается меньше определенного близкого к нулю значения eps, программа сообщает, что результат ее работы корректен.

# Результаты экспериментов

Известно, что теоретическая вычислительная сложность у алгоритма Метода Гаусса равна . Замерим скорость работы программы в зависимости от размера матрицы (от 100 до 500)

На рис. 1 представлен график зависимости времени от размера введенной матрицы. На рис. 2 представлен график функции , отображающий число операций в методе Гаусса. Графики построены по таблице 1 и 2 соответственно.

Таблица 1. Время выполнения

|  |  |
| --- | --- |
| **Размер матрицы (n x n)** | **Среднее время (с)** |
| 100 | 0.008538 |
| 120 | 0.0135268 |
| 140 | 0.0191628 |
| 160 | 0.0264403 |
| 180 | 0.0365994 |
| 200 | 0.0453376 |
| 220 | 0.0582491 |
| 240 | 0.0709907 |
| 260 | 0.090512 |
| 280 | 0.104143 |
| 300 | 0.132192 |
| 320 | 0.149228 |
| 340 | 0.182024 |
| 360 | 0.21604 |
| 380 | 0.248236 |
| 400 | 0.287634 |
| 420 | 0.329496 |
| 440 | 0.379531 |
| 460 | 0.422472 |
| 480 | 0.465153 |
| 500 | 0.527026 |

Рис.1 Зависимость времени выполнения программы от размера введенной матрицы

Таблица 2. Количество операций

|  |  |
| --- | --- |
| **Размер матрицы (n x n)** | **Количество операций** |
| 100 | 0.008538 |
| 120 | 0.0135268 |
| 140 | 0.0191628 |
| 160 | 0.0264403 |
| 180 | 0.0365994 |
| 200 | 0.0453376 |
| 220 | 0.0582491 |
| 240 | 0.0709907 |
| 260 | 0.090512 |
| 280 | 0.104143 |
| 300 | 0.132192 |
| 320 | 0.149228 |
| 340 | 0.182024 |
| 360 | 0.21604 |
| 380 | 0.248236 |
| 400 | 0.287634 |
| 420 | 0.329496 |
| 440 | 0.379531 |
| 460 | 0.422472 |
| 480 | 0.465153 |
| 500 | 0.527026 |

Рис. 2. Зависимость количества операций от размера матрицы

Исходя из наглядных данным можно сделать вывод, что графики зависимостей внешне практически идентичны, что подтверждает наше предположение о том, что сложность алгоритма равна .

# Заключение

Реализовали метод Гаусса для решения СЛАУ. Написали два шаблонных класса Vector и Matrix, сделав последний наследником вектора векторов. Организовали свой механизм исключений, добавили функци.ю проверки корректности работы программы.. Провели эксперимент, в процессе которого провели сравнение теоретической алгоритмической сложности с зависимостью времени работы программы от размера входной матрицы, получив удовлетворительный результат.

# Приложение

#include <iostream>

#include <cmath>

#include <stdexcept>

#include <chrono>

#include <cstdlib>

#include <random>

#include <iomanip>

#include <limits>

template <typename T>

class Vector {

protected:

T\* data{ nullptr };

size\_t size{ 0 };

public:

Vector() = default; // конструктор по умолчанию

Vector(size\_t n) { // Выделяем память на элемент

try {

data = new T[n](); // создаем массив указателей типа T

size = n; // длина вектора

}

catch (...) {

throw std::runtime\_error("Failed to allocate memory."); // выбрасывает ошибку, если не хватило памяти

}

}

~Vector() { delete[] data; } // деструктор. необходим, потому что память выделяется засчет new

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

try {

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

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

}

}

catch (...) { // поймает ошибку, если два вектора не совпадут по длине

delete[] data;

throw;

}

}

size\_t GetSize() const noexcept { return size; } // возвращает длину вектора

void Swap(Vector& other) noexcept {

std::swap(data, other.data);

std::swap(size, other.size);

} // меняет два вектора местами

Vector& operator=(const Vector& other) {

if (this == &other) {

return \*this;

}

Vector temp(other);

this->Swap(temp);

return \*this;

} // операция присваивания вектора другому вектору

T& operator[](size\_t i) {

if (i >= size) {

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

}

return data[i];

} // оператор получения элемента вектора по индексу

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

if (i >= size) {

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

}

return data[i];

} // константный оператор получения вектора по индексу, чтобы работать с константами

Vector& operator/=(T scalar) {

if (scalar == 0.0) {

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

}

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

data[i] /= scalar;

return \*this;

}

// оператор деления вектора на скаляр

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

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

data[i] -= rhs[i];

return \*this;

} // оператор вычитания с присваиванием (вектор и вектор)

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

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

data[i] += rhs[i];

return \*this;

} // оператор суммирования с присваиванием (вектор и вектор)

Vector operator\*(T scalar) const {

Vector res(size);

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

res[i] = data[i] \* scalar;

return res;

} // оператор умножения вектора на скаляр

};

template <typename T>

class Matrix : public Vector<Vector<T>> { // матрица наследует шаблон от Vector

size\_t rows{ 0 };

size\_t columns{ 0 };

public:

Matrix() = default; // Конструктор по умолчанию

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

for (size\_t i = 0; i < rows; ++i) { // Создание матрицы из векторов

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

}

}

~Matrix() = default; // Деструктор по умолчанию

Matrix(const Matrix& other) : Vector<Vector<T>>(other), rows(other.rows), columns(other.columns) {} // Конструктор копирования

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

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

Matrix& operator=(const Matrix& other) {

if (this == &other) {

return \*this;

}

Matrix temp(other);

this->Swap(temp);

return \*this;

}

Vector<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);

}

const Vector<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);

}

Matrix operator\*(T scalar) const {

Matrix res(\*this);

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

res[i] \*= scalar;

}

return res;

}

void Swap(Matrix& other) noexcept {

Vector<Vector<T>>::Swap(other);

std::swap(rows, other.rows);

std::swap(columns, other.columns);

}

void 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 }; // совместна

bool has\_unique{ false }; // единственное решение

Vector<T> particular; // частные решения

Vector<T>\* basis{ nullptr }; // указатель на массив с базисными векторами

size\_t basis\_size{ 0 }; // размер базиса

size\_t\* free\_vars{ nullptr }; // указатель на свободные переменные

size\_t rank{ 0 }; // ранг матрицы

Solution() = default; // конструктор по умолчанию

~Solution() { // деструктор

delete[] basis;

delete[] free\_vars;

}

};

template <typename T>

Solution<T> GaussMethod(Matrix<T> A, Vector<T> b) {

const size\_t m = A.GetRows(); // Количество строк = элементов в столбце

const size\_t n = A.GetCols(); // Количество столбцов = элементов в строке

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

Solution<T> sol; // Объект для решения

sol.is\_consistent = true; // Предполагаю, что совместна

size\_t rank = 0; // ранг равен 0 (пока что)

size\_t\* pivot\_columns = new size\_t[m]; // указатель на массив с ведущими элементами столбцов

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

pivot\_columns[i] = n; // изначально ставим ведущим последний элемент столбца

}

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

size\_t max\_row = rank; // значение максимального элемента в столбце

for (size\_t i = rank; i < m; ++i) { // пробегаемся по столбцу, выбираем максимальный элемент и записываем его индекс

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

max\_row = i;

}

}

if (std::abs(A[max\_row][column]) < eps) continue; // если весь столбец крайне незначительно отличается от 0, то его пропускаем

pivot\_columns[rank] = column; // выбираем в качестве ведущего элемента элемент с индексом порядкового номера столбца (потом сделаем верхнетруегольную матрицу)

A.SwapRows(rank, max\_row); // меняем местами строку с позицией ведущего элемента и самим ведущим элементом

std::swap(b[rank], b[max\_row]); // после смены местами двух строк меняем два элемента в векторе b

T pivot = A[rank][column];

A[rank] /= pivot; // делим строку на ведущий элемент, чтобы он был равен 1

b[rank] /= pivot;

for (size\_t i = rank + 1 ; i < m; ++i) { // обнуляем столбец с ведущим элементом

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

T factor = A[i][column]; // множитель, на который нужно помножить ведущий столбец, чтобы обнулить ячейку

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

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

}

}

++rank; // переход на следующую строку

}

for (size\_t i = rank; i < m; ++i) { // проходим под главной диагональю матрицы по вектору b. Проверяем на 0\*х != 0

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

sol.is\_consistent = false; // если система несовместна, то высвобождается память из под массива и возвращается объект решения

delete[] pivot\_columns;

return sol;

}

}

sol.rank = rank;

sol.has\_unique = (rank == n);

sol.particular = Vector<T>(n);

bool\* is\_pivot = new bool[n](); // проверяем, есть ли в столбце ведущий элемент

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

if (pivot\_columns[i] < n) {

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

}

}

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

size\_t column = pivot\_columns[i];

if (column >= n) continue;

sol.particular[column] = b[i]; // собираем частное решение из ведущих элементов. Пропускаем не ведущие столбцы

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

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

}

}

sol.basis\_size = 0; // считаем, сколько свободных переменных

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]; // массив базисных векторов

sol.free\_vars = new size\_t[sol.basis\_size]; // массив свободных переменных

size\_t basis\_idx = 0;

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

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

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

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

sol.basis[basis\_idx][free\_column] = 1; // множитель перед свободной переменной в векторе будет равен 1

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

size\_t pivot\_column = pivot\_columns[i];

if (pivot\_column >= n) continue;

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

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

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

}

}

++basis\_idx;

}

}

}

delete[] is\_pivot;

delete[] pivot\_columns;

return sol;

}

template <typename T>

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

const T eps = std::numeric\_limits<T>::epsilon() \* static\_cast<T>(A.GetRows()) \* 1000;

if (!solution.is\_consistent) {

std::cout << "Программа сработала корректно!\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] \* solution.particular[j];

}

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

T max\_val = std::max(T(1), std::max(std::abs(sum), std::abs(b[i])));

if (error > eps \* max\_val) {

throw std::runtime\_error("Программа сработала некорректно.\n");

}

}

std::cout << "Программа сработала корректно!\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 << "Система уравнений несовместна, у нее отсутствуют решения\n"; // система несовместна

return;

}

std::cout << "Результат:\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 << "Введите размер матрицы (n x n): (Ввести нужно только одно значение, так как матрица квадратная)";

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(static\_cast<double>(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 \* (1 + dist(gen) / static\_cast<T>(1000.0));

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 << "Время работы программы: ";

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

PrintSolution(sol);

}

template <typename T>

void ProcessManualInput() {

size\_t m, n;

std::cout << "Введите количество уравнений (m): "; // количество строк

std::cin >> m;

std::cout << "Введите количество переменных (n): "; // количество столбцов - неизвестных

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 << "Введите расширенную матрицу коэффициентов:\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. Сгенерировать рандомную матрицу\n";

std::cout << "2. Ввести матрицу вручную\n";

std::cout << "3. Завершить работу программы\n";

std::cout << "Ваш выбор: ";

size\_t choice;

std::cin >> choice;

switch (choice) {

case RandomMatrix:

ProcessRandomMatrix<T>();

break;

case ManualInput:

ProcessManualInput<T>();

break;

case Exit:

std::cout << "Завершение работы программы...\n";

return;

default:

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

}

std::cout << std::endl;

}

enum TypeChoice {

FloatType = 1,

DoubleType = 2

};

int main() {

setlocale(LC\_ALL, "Russian");

try {

std::cout << "Выберите числовой тип (от него зависит точность работы программы):\n";

std::cout << "1. float (одинарная точность)\n";

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

std::cout << "3. Завершить работу программы\n";

std::cout << "Введите свой выбор: ";

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";

return 0;

default:

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

}

}

catch (const std::exception& e) {

std::cerr << "Error: " << e.what() << std::endl;

}

return 0;

}