Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ	«Информатика и системы управления»
Кафедра	ИУ-6 «Компьютерные системы и сети»
Группа	«ИУ6-64Б »

Отчет по домашнему заданию №1:

Прямые методы решения СЛАУ

Вариант 13ВЫЧИСЛИТЕЛЬНАЯ МАТЕМАТИКА

Отчет по домашнему заданию №1:

Прямые методы решения СЛАУ

Вариант 13

Студент: _____ Кощенков Д. О.

(дата, подпись) (ФИО)

Преподаватель: _____ Орлова А. С.

(дата, подпись) (ФИО)

Москва, 2024

Содержание

1 Введение

Цель домашней работы: изучение методов Гаусса, LU-разложения для численного решения квадратной СЛАУ с невырожденной матрицей, оценка числа обусловленности матрицы и исследование его влияния на погрешность приближенного решения. Изучение метода прогонки для решения СЛАУ с трехдиагональной матрицей.

Для достижения цели необходимо решить следующие задачи:

1. Реализовать на языке C++ указанные методы решения СЛАУ.
2. Провести решение двух заданных квадратных СЛАУ указанными методами, вычислить нормы невязок полученных приближенных решений, их предельные абсолютные и относительные погрешности (использовать 1-норму и ∞-норму).
3. С использованием реализованных методов найти обратные матрицы для матриц квадратных систем, проверить выполнение равенств $A \cdot A^{-1} = E$.
4. Для каждой системы оценить порядок числа обусловленности матрицы системы и сделать вывод о его влиянии на точность полученного приближенного решения и отвечающую ему невязку.
5. Провести решение СЛАУ с трехдиагональной матрицей реализованным методом прогонки и найти норму его невязки (использовать 1-норму и ∞-норму).

Отчет должен содержать:

1. Постановку задачи и исходные данные.
2. Краткое описание реализуемых методов.
3. Текст программы.
4. Результаты расчетов (оформленные в виде таблиц).
5. Анализ полученных результатов (влияние числа обусловленности матрицы системы на точность полученного приближенного решения и отвечающую ему невязку).
6. Заключение.

2 Постановка задачи и исходные данные

2.1 Решение СЛАУ (Хорошо и Плохо обусловленные матрицы)

Задача: Даны две квадратные СЛАУ $Ax = b$ размерности 4×4: одна с хорошо обусловленной матрицей $(A_{\text{good}}, b_{\text{good}})$, другая - с плохо обусловленной $(A_{\text{bad}}, b_{\text{bad}})$. Для каждой из этих систем необходимо:

1. Найти вектор решения x^* с использованием метода Гаусса (с выбором главного элемента) и метода LU-разложения.
2. Вычислить обратную матрицу A^{-1}.
3. Проверить выполнение равенства $A \cdot A^{-1} = E$ (где E - единичная матрица) с учетом вычислительной погрешности.
4. Оценить число обусловленности матрицы $cond(A) = ||A|| \cdot ||A^{-1}||$, используя 1-норму и ∞-норму.
5. Для каждого полученного решения x^* вычислить:
   - Вектор невязки $r = Ax^* - b$.
   - Нормы невязки: $||r||_1$ и $||r||_\infty$.
   - Вектор абсолютной погрешности $\Delta x = x^* - x_{exact}$, где x_{exact} - заданное точное решение.
   - Нормы абсолютной погрешности: $||\Delta x||_1$ и $||\Delta x||_\infty$.
   - Относительные погрешности: $\frac{||\Delta x||_1}{||x_{exact}||_1}$ и $\frac{||\Delta x||_\infty}{||x_{exact}||_\infty}$.
6. Проанализировать полученные результаты, сравнив точность решений и величину невязок для хорошо и плохо обусловленных систем, и связать это с числом обусловленности.

Исходные данные (Вариант 13):

- Хорошо обусловленная система:

$$A_{\text{good}} = \begin{pmatrix} -38.40 & 4.03 & 8.38 & 3.53 \\ -8.32 & -81.20 & -8.09 & -3.67 \\ 4.33 & 7.21 & -110.80 & -2.63 \\ 4.22 & 4.22 & 6.15 & 73.80 \end{pmatrix}, \quad b_{\text{good}} = \begin{pmatrix} 292.59 \\ -504.32 \\ -185.66 \\ -430.50 \end{pmatrix}, \quad x_{\text{exact, good}} = \begin{pmatrix} -7 \\ 7 \\ 2 \\ -6 \end{pmatrix}$$

- Плохо обусловленная система:

$$A_{\text{bad}} = \begin{pmatrix} -190.327 & 189.760 & -18.016 & 72.064 \\ -194.520 & 193.953 & -18.432 & 73.728 \\ -919.480 & 919.480 & -99.247 & 398.680 \\ -219.390 & 219.390 & -23.772 & 95.511 \end{pmatrix}, \quad b_{\text{bad}} = \begin{pmatrix} 1414.281 \\ 1446.762 \\ 7705.500 \\ 1845.192 \end{pmatrix}, \quad x_{\text{exact, bad}} = \begin{pmatrix} 1 \\ 2 \\ 20 \\ 22 \end{pmatrix}$$

2.2 Решение СЛАУ с трехдиагональной матрицей

Задача: Дана СЛАУ $Tx = d$ с трехдиагональной матрицей T размерности 6×6. Необходимо:

1. Найти решение x^* системы с использованием метода прогонки (алгоритм Томаса).
2. Вычислить вектор невязки $r = Tx^* - d$.
3. Найти нормы невязки: $||r||_1$ и $||r||_\infty$.

Исходные данные (Вариант 13):

- Матрица T задана тремя векторами:
  - Поддиагональ (ниже главной, $a_i = T_{i,i-1}$ для $i = 1..5$): $a = (1.0, -1.0, 0.0, 1.0, 1.0)$
  - Главная диагональ ($b_i = T_{i,i}$ для $i = 0..5$): $b = (65.0, 91.0, 147.0, 78.0, 99.0, 132.0)$
  - Наддиагональ (выше главной, $c_i = T_{i,i+1}$ для $i = 0..4$): $c = (1.0, -1.0, 1.0, 0.0, 1.0)$

  Полная матрица T имеет вид:

$$T = \begin{pmatrix} 65.0 & 1.0 & 0 & 0 & 0 & 0 \\ 1.0 & 91.0 & -1.0 & 0 & 0 & 0 \\ 0 & -1.0 & 147.0 & 1.0 & 0 & 0 \\ 0 & 0 & 0.0 & 78.0 & 0.0 & 0 \\ 0 & 0 & 0 & 1.0 & 99.0 & 1.0 \\ 0 & 0 & 0 & 0 & 1.0 & 132.0 \end{pmatrix}$$

- Вектор правых частей d: $d = (6.0, 10.0, 14.0, 8.0, 10.0, 13.0)$.

3 Краткое описание реализуемых методов

- Метод Гаусса (с выбором главного элемента по столбцу): Прямой метод решения СЛАУ $Ax = b$. Состоит из двух этапов:

  1. Прямой ход: Расширенная матрица $[A|b]$ приводится к верхнетреугольному виду $[U|y]$ путем элементарных преобразований строк. На каждом i-м шаге выбирается максимальный по модулю элемент a_{ki} в i-м столбце ($k \geq i$). Строки i и k меняются местами (пивотирование). Затем из всех нижележащих строк вычитается i-я строка, умноженная на соответствующий коэффициент, чтобы обнулить элементы под a_{ii}. Те же преобразования применяются к вектору b.
  2. Обратный ход: Решается система $Ux = y$ с верхнетреугольной матрицей U путем обратной подстановки, начиная с x_{n-1} и двигаясь к x_0.

  Выбор главного элемента повышает вычислительную устойчивость метода.

- LU-разложение (метод Дулиттла): Метод факторизации, представляющий квадратную матрицу A как произведение $A = LU$, где L - нижняя треугольная матрица с единицами на диагонали, U - верхняя треугольная матрица. Решение $Ax = b$ сводится к решению двух систем: $Ly = b$ (прямая подстановка) и $Ux = y$ (обратная подстановка). Элементы L и U вычисляются по формулам, например:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}, \quad l_{ji} = \frac{1}{u_{ii}}\left(a_{ji} - \sum_{k=1}^{i-1} l_{jk}u_{ki}\right)$$

  Реализованная версия не использует пивотирование, что может быть проблематично для плохо обусловленных матриц.

- Нахождение обратной матрицы (методом Гаусса-Жордана): Модификация метода Гаусса. Формируется расширенная матрица $[A|E]$. С помощью элементарных преобразований строк (с выбором главного элемента) матрица A приводится к единичной матрице E. В результате этих же преобразований правая часть E превращается в A^{-1}, получая итоговую матрицу $[E|A^{-1}]$.

- Число обусловленности: $cond(A) = ||A|| \cdot ||A^{-1}||$. Характеризует чувствительность решения СЛАУ к возмущениям входных данных. Большое значение $cond(A)$ указывает на плохую обусловленность. Вычисляется с использованием 1-нормы и ∞-нормы.

- Нормы векторов и матриц: Используются для оценки величины погрешностей и невязок.
  - $||\mathbf{v}||_1 = \sum_i |v_i|$ (1-норма вектора).
  - $||\mathbf{v}||_\infty = \max_i |v_i|$ (∞-норма вектора).
  - $||A||_1 = \max_j \sum_i |a_{ij}|$ (1-норма матрицы, максимальная сумма модулей по столбцам).

- $||A||_\infty = \max_i \sum_j |a_{ij}|$ (∞-норма матрицы, максимальная сумма модулей по строкам).

- Невязка: Вектор $r = Ax^* - b$. Показывает, насколько точно найденное решение x^* удовлетворяет исходной системе.

- Абсолютная погрешность: Вектор $\Delta x = x^* - x_{exact}$. Разница между вычисленным и точным решением.

- Относительная погрешность: Величина $||\Delta x||/||x_{exact}||$. Характеризует точность решения относительно величины самого решения (в соответствующей норме).

- Метод прогонки (Алгоритм Томаса): Эффективный прямой метод для решения СЛАУ с трехдиагональной матрицей $Tx = d$.

  – Прямой ход: Вычисляются прогоночные коэффициенты P_i и Q_i:

$$P_0 = -\frac{c_0}{b_0}, \quad Q_0 = \frac{d_0}{b_0}$$

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

  (где a_i - элементы поддиагонали $T_{i,i-1}$, b_i - диагонали $T_{i,i}$, c_i - наддиагонали $T_{i,i+1}$,: Индексация векторов a, b, c в формулах соответствует индексам матрицы, но в коде они могут быть сдвинуты).

  – Обратный ход: Решение x находится по формулам:

$$x_{n-1} = Q_{n-1}$$

$$x_i = P_i x_{i+1} + Q_i, \quad i = n-2, \ldots, 0$$

4 Текст программы

```
//
typedef std::vector<std::vector<double>> Matrix;

//

const double EPSILON = 1e-9;

// ---                                                                    ---

void print_vector(const std::vector<double>& v, const std::string& name =
    "", int precision = 6) {
     if (!name.empty()) {
         std::cout << name << ": ";
     }
     std::cout << std::fixed << std::setprecision(precision);
     std::cout << "[";
     for (size_t i = 0; i < v.size(); ++i) {
         std::cout << v[i] << (i == v.size() - 1 ? "" : ", ");
     }
     std::cout << "]" << std::endl;
}

void print_matrix(const Matrix& matrix, const std::string& name = "", int
    precision = 6) {
     if (!name.empty()) {
         std::cout << name << ":" << std::endl;
     }
     std::cout << std::fixed << std::setprecision(precision);
     for (size_t i = 0; i < matrix.size(); ++i) {
         std::cout << "  "; //

         print_vector(matrix[i], "", precision);
     }
}

// ---                                                                    ---

//                          1-                              : sum(|v_i|)
double compute_vector_norm_1(const std::vector<double>& v) {
     double norm = 0.0;
```

```
    for (double val : v) {
        norm += std::abs(val);
    }
    return norm;
}

//                        infinity-                          : max(|v_i|)
double compute_vector_norm_inf(const std::vector<double>& v) {
    double norm = 0.0;
    for (double val : v) {
        norm = std::max(norm, std::abs(val));
    }
    return norm;
}

//                                                          : A - B
std::vector<double> diff_vector(const std::vector<double>& A, const
   std::vector<double>& B) {
    if (A.size() != B.size()) {
        throw std::invalid_argument("
                                                                    .");
    }
    const size_t n = A.size();
    std::vector<double> result(n);
    for (size_t i = 0; i < n; ++i) {
        result[i] = A[i] - B[i];
    }
    return result;
}

// ---                                                              ---

//                        1-                          : max
                    (sum(|a_ij|))
double compute_matrix_norm_1(const Matrix& matrix) {
    if (matrix.empty() || matrix[0].empty()) return 0.0;
    const size_t N = matrix.size();
    const size_t M = matrix[0].size();
    double max_col_sum = 0.0;

    for (size_t j = 0; j < M; ++j) {
        double current_col_sum = 0.0;
```

```
        for (size_t i = 0; i < N; ++i) {
            current_col_sum += std::abs(matrix[i][j]);
        }
        max_col_sum = std::max(max_col_sum, current_col_sum);
    }
    return max_col_sum;
}

//                        infinity -                          : max
                  (sum(|a_ij|))
double compute_matrix_norm_inf(const Matrix& matrix) {
    if (matrix.empty()) return 0.0;
    const size_t N = matrix.size();
    const size_t M = matrix[0].size(); //                          ,

    double max_row_sum = 0.0;

    for (size_t i = 0; i < N; ++i) {
        double current_row_sum = 0.0;
        for (size_t j = 0; j < M; ++j) {
            //
                                                              (
                                                          )
            if (j < matrix[i].size()) {
                current_row_sum += std::abs(matrix[i][j]);
            }
        }
        max_row_sum = std::max(max_row_sum, current_row_sum);
    }
    return max_row_sum;
}


//                                                    : y = A * x
std::vector<double> multiply_matrix_vector(const Matrix& A, const
   std::vector<double>& x) {
    const size_t rowsA = A.size();
    if (rowsA == 0) return {};
    const size_t colsA = A[0].size();
    const size_t sizeX = x.size();

    if (colsA != sizeX) {
```

```
        throw std::invalid_argument("

                                                    .");
    }

    std::vector<double> y(rowsA, 0.0);
    for (size_t i = 0; i < rowsA; ++i) {
        for (size_t j = 0; j < colsA; ++j) {
            y[i] += A[i][j] * x[j];
        }
    }
    return y;
}

//                                  : C = A * B
Matrix multiply(const Matrix& A, const Matrix& B) {
    const size_t rowsA = A.size();
    if (rowsA == 0) return {};
    const size_t colsA = A[0].size();
    const size_t rowsB = B.size();
    if (rowsB == 0) return {};
    const size_t colsB = B[0].size();

    if (colsA != rowsB) {
        throw std::invalid_argument("

                                                                .");
    }

    Matrix result(rowsA, std::vector<double>(colsB, 0.0));
    for (size_t i = 0; i < rowsA; ++i) {
        for (size_t j = 0; j < colsB; ++j) {
            for (size_t k = 0; k < colsA; ++k) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return result;
}

//
Matrix create_identity_matrix(size_t n) {
    Matrix I(n, std::vector<double>(n, 0.0));
```

```
    for (size_t i = 0; i < n; ++i) {
        I[i][i] = 1.0;
    }
    return I;
}

//                                                              epsilon
bool are_matrices_close(const Matrix& A, const Matrix& B, double tolerance
    = EPSILON) {
    if (A.size() != B.size() || (A.empty() && !B.empty()) || (!A.empty() &&
        B.empty())) {
        return false;
    }
    if (A.empty()) return true; //
    if (A[0].size() != B[0].size()) {
        return false;
    }

    const size_t rows = A.size();
    const size_t cols = A[0].size();

    for (size_t i = 0; i < rows; ++i) {
        for (size_t j = 0; j < cols; ++j) {
            std::cout << std::abs(A[i][j] - B[i][j]) << ' ';

            if (std::abs(A[i][j] - B[i][j]) > tolerance) {
                return false;
            }
        }

        std::cout << std::endl;
    }
    return true;
}


// ---                                          ---

//

std::vector<double> gauss(Matrix A, std::vector<double> b) { //
                                    ,    .   .
```

```
    const size_t n = A.size();
    if (n == 0 || A[0].size() != n || b.size() != n) {
        throw std::invalid_argument("

                                    .");
    }

    //
    for (size_t i = 0; i < n; ++i) {
        //
                            (                    i-                )
        size_t maxRow = i;
        for (size_t k = i + 1; k < n; ++k) {
            if (std::abs(A[k][i]) > std::abs(A[maxRow][i])) {
                maxRow = k;
            }
        }

        //                                                  (
                                                    )
        if (std::abs(A[maxRow][i]) < EPSILON) {
            //
                                                                ,

            // (

                                                )
            bool found_pivot = false;
            for(size_t check_col = i + 1; check_col < n; ++check_col) {
                if (std::abs(A[maxRow][check_col]) >= EPSILON) {
                    //

                                                                    ,
                    //
                                                ( )
                                                                .
                    //
                                                .
                    found_pivot = true; //
                                                                ,

                    break;
```

```
            }
        }
        if (!found_pivot) {
            throw std::runtime_error("

                                        (                          ).");
        }
        //
           (
                                                    ):
        // continue; //                                                ,

        //                                                          :
        throw std::runtime_error("

                                (                          ).");
}


//                                        (              A
                    b)
std::swap(A[i], A[maxRow]);
std::swap(b[i], b[maxRow]);

//                              i-              (
                              ,                                  )
// double pivot = A[i][i];
// for (size_t j = i; j < n; ++j) {
//     A[i][j] /= pivot;
// }
// b[i] /= pivot;

//

double pivot = A[i][i]; //                    pivot

for (size_t k = i + 1; k < n; ++k) {
    const double factor = A[k][i] / pivot;
    if (std::abs(factor) < EPSILON) continue; //
                              ,

    for (size_t j = i; j < n; ++j) { //                        j=i,
```

```
                . . A[k][i]
            A[k][j] -= factor * A[i][j];
        }
        b[k] -= factor * b[i];
        //                                                    (
                                                              0)
        A[k][i] = 0.0;
    }
}

//
std::vector<double> x(n);
for (int i = static_cast<int>(n) - 1; i >= 0; --i) {
    if (std::abs(A[i][i]) < EPSILON) {
        //
                                                ,

         throw std::runtime_error("

            (                          ).");
    }
    double sum = 0.0;
    for (size_t j = i + 1; j < n; ++j) {
        sum += A[i][j] * x[j];
    }
    x[i] = (b[i] - sum) / A[i][i];
}

return x;
}


// LU-                    (Doolittle's method: L          1
                    )
//                                {L, U}
std::pair<Matrix, Matrix> LU_decomposition(const Matrix& matrix) {
    const size_t n = matrix.size();
    if (n == 0 || matrix[0].size() != n) {
        throw std::invalid_argument("
                                        LU-                    .");
    }
```

```
    Matrix L(n, std::vector<double>(n, 0.0));
    Matrix U(n, std::vector<double>(n, 0.0));

    for (size_t i = 0; i < n; ++i) {
        //              U
        for (size_t j = i; j < n; ++j) {
            double sum = 0.0;
            for (size_t k = 0; k < i; ++k) {
                sum += L[i][k] * U[k][j];
            }
            U[i][j] = matrix[i][j] - sum;
        }

        //                                                    U
        if (std::abs(U[i][i]) < EPSILON) {
            // LU-

            throw std::runtime_error("
                                                        U
               LU-                                                  .");
        }

        //              L
        L[i][i] = 1.0; //                    L          1        Doolittle
        for (size_t j = i + 1; j < n; ++j) {
            double sum = 0.0;
            for (size_t k = 0; k < i; ++k) {
                sum += L[j][k] * U[k][i];
            }
            L[j][i] = (matrix[j][i] - sum) / U[i][i];
        }
    }

    return {L, U};
}

//                                                    Ly = b (L -
                                    )
std::vector<double> solve_forward_substitution(const Matrix& L, const
    std::vector<double>& b) {
    const size_t n = L.size();
    if (n == 0 || L[0].size() != n || b.size() != n) {
```

```
            throw std::invalid_argument("
                                                                    .");
        }
        std::vector<double> y(n);

        for (size_t i = 0; i < n; ++i) {
            double sum = 0.0;
            for (size_t j = 0; j < i; ++j) {
                sum += L[i][j] * y[j];
            }
            //                                                      L
               (              Doolittle L[i][i]=1)
            if (std::abs(L[i][i]) < EPSILON) {
                throw std::runtime_error("
                                                                L
                                                              .");
            }
            y[i] = (b[i] - sum) / L[i][i];
        }
        return y;
    }

    //                                                          Ux = y (U
       -                                    )
    std::vector<double> solve_backward_substitution(const Matrix& U, const
       std::vector<double>& y) {
        const size_t n = U.size();
         if (n == 0 || U[0].size() != n || y.size() != n) {
            throw std::invalid_argument("

                                        .");
        }
        std::vector<double> x(n);

        for (int i = static_cast<int>(n) - 1; i >= 0; --i) {
            double sum = 0.0;
            for (size_t j = i + 1; j < n; ++j) {
                sum += U[i][j] * x[j];
            }
             if (std::abs(U[i][i]) < EPSILON) {
                throw std::runtime_error("
                                                                U
```

```
                                                        .");
        }
        x[i] = (y[i] - sum) / U[i][i];
    }
    return x;
}

//                          Ax=b
   LU-
std::vector<double> solve_lu(const Matrix& A, const std::vector<double>& b)
   {
    try {
        auto [L, U] = LU_decomposition(A);
        std::vector<double> y = solve_forward_substitution(L, b);
        std::vector<double> x = solve_backward_substitution(U, y);
        return x;
    } catch (const std::runtime_error& e) {
        //

        throw std::runtime_error(std::string("
                                            LU: ") + e.what());
    } catch (const std::invalid_argument& e) {
         throw std::invalid_argument(std::string("
                                            LU: ") + e.what());
    }
}

//
               -
Matrix inverse_matrix(Matrix matrix) { //
    const size_t n = matrix.size();
     if (n == 0 || matrix[0].size() != n) {
        throw std::invalid_argument("

                            .");
    }

    //                                                      [A | E]
    Matrix augmented(n, std::vector<double>(2 * n));
    for (size_t i = 0; i < n; ++i) {
        for (size_t j = 0; j < n; ++j) {
            augmented[i][j] = matrix[i][j];
```

```
        }
        augmented[i][i + n] = 1.0; //

    }

    //                          (
                                                )
    for (size_t i = 0; i < n; ++i) {
        //
        size_t maxRow = i;
        for (size_t k = i + 1; k < n; ++k) {
            if (std::abs(augmented[k][i]) > std::abs(augmented[maxRow][i]))
               {
                maxRow = k;
            }
        }

        if (std::abs(augmented[maxRow][i]) < EPSILON) {
            throw std::runtime_error("                                    ,
                                                        .");
        }

        //
        std::swap(augmented[i], augmented[maxRow]);

        //                              i-                  (
                                            )
        double pivot = augmented[i][i];
        for (size_t j = i; j < 2 * n; ++j) { //

            augmented[i][j] /= pivot;
        }
         augmented[i][i] = 1.0; //                    ,
                                                                1


        //

        for (size_t k = 0; k < n; ++k) {
            if (k != i) { //                              ,

                double factor = augmented[k][i];
                if (std::abs(factor) < EPSILON) continue; //
```

```
                                        ,
                for (size_t j = i; j < 2 * n; ++j) { //
                    j=i
                     augmented[k][j] -= factor * augmented[i][j];
                }
                augmented[k][i] = 0.0; //

            }
        }
    }
    //
                                                    (
                -           )

    //                                                  (
                                                    )
    Matrix inv(n, std::vector<double>(n));
    for (size_t i = 0; i < n; ++i) {
        for (size_t j = 0; j < n; ++j) {
            inv[i][j] = augmented[i][j + n];
        }
    }

    return inv;
}

//                                                  r = Ax - b
std::vector<double> compute_residual(const Matrix& A, const
   std::vector<double>& x, const std::vector<double>& b) {
    std::vector<double> Ax = multiply_matrix_vector(A, x);
    return diff_vector(Ax, b);
}


// ---
                ---

//

// a -                          (a_1 .. a_{n-1}),              n-1
// b -                                (b_0 .. b_{n-1}),              n
// c -                          (c_0 .. c_{n-2}),              n-1
```

```
// d -                          (d_0 .. d_{n-1}),              n
std::vector<double> solve_tridiagonal(
    const std::vector<double>& a, //        -
       (                1..n-1                  )
    const std::vector<double>& b, //
       (                0..n-1)
    const std::vector<double>& c, //        -
       (                0..n-2)
    const std::vector<double>& d)
{
    const size_t n = b.size();
    if (a.size() != n - 1 || c.size() != n - 1 || d.size() != n || n == 0) {
        throw std::invalid_argument("

                                .");
    }

    std::vector<double> P(n); //
                                P_i
    std::vector<double> Q(n); //
                                Q_i
    std::vector<double> x(n); //

    //                        (                    P    Q)
    if (std::abs(b[0]) < EPSILON) {
         throw std::runtime_error("                                b[0]
                                          .");
    }
    P[0] = -c[0] / b[0];
    Q[0] = d[0] / b[0];

    for (size_t i = 1; i < n; ++i) {
        double denominator;
        //                        a    c: a[i-1]
           a_i                     , c[i-1]
           c_{i-1}
        if (i < n - 1) { //                ,

            denominator = b[i] + a[i-1] * P[i-1];
            if (std::abs(denominator) < EPSILON) {
                throw std::runtime_error("
```

```
                                                                    .");
            }
            P[i] = -c[i] / denominator; // c[i]
                c_i
            Q[i] = (d[i] - a[i-1] * Q[i-1]) / denominator;
        } else { //                                  (i = n - 1)
            denominator = b[i] + a[i-1] * P[i-1];
            if (std::abs(denominator) < EPSILON) {
                throw std::runtime_error("


                                                Q.");
            }
            // P[n-1]
                (                                          0)
            P[n-1] = 0.0; //
                                        c[n-1]
            Q[i] = (d[i] - a[i-1] * Q[i-1]) / denominator;
        }
    }

    //                          (                  x)
    x[n - 1] = Q[n - 1];
    for (int i = static_cast<int>(n) - 2; i >= 0; --i) {
        x[i] = P[i] * x[i + 1] + Q[i];
    }

    return x;
}


//

Matrix build_tridiagonal_matrix(
    const std::vector<double>& a, //       -                  (a_1 ..
       a_{n-1}),                  n-1
    const std::vector<double>& b, //                                  (b_0
       .. b_{n-1}),                 n
    const std::vector<double>& c) //       -                  (c_0 ..
       c_{n-2}),                  n-1
{
    const size_t n = b.size();
```

```
    if (a.size() != n - 1 || c.size() != n - 1 || n == 0) {
        throw std::invalid_argument("

                                                                .");
    }
    Matrix T(n, std::vector<double>(n, 0.0));
    for (size_t i = 0; i < n; ++i) {
        T[i][i] = b[i];
        if (i > 0) {
            T[i][i - 1] = a[i - 1]; // a[k]
                                    T[k+1][k]
        }
        if (i < n - 1) {
            T[i][i + 1] = c[i]; // c[k]
                                T[k][k+1]
        }
    }
    return T;
}
```

5 Результаты расчетов

Результаты получены путем запуска программы, код которой представлен в Листинге 1. Для вывода чисел используется формат с плавающей точкой с 15 знаками после запятой для решений и погрешностей, и 7 знаками для норм матриц. Для чисел обусловленности и очень малых/больших норм используется научная нотация.

5.1 Хорошо обусловленная система

Таблица 1. Результаты для хорошо обусловленной системы

Величина	Метод Гаусса	Метод LU
x_0^*	−7,000 000 000 000 000	−7,000 000 000 000 000
x_1^*	7,000 000 000 000 000	7,000 000 000 000 000
x_2^*	2,000 000 000 000 000	2,000 000 000 000 000
x_3^*	−6,000 000 000 000 000	−6,000 000 000 000 000
Невязка $r = Ax^* - b$		
$\|\|r\|\|_1$	1,136 868 377 216 160	2,273 736 754 432 321
$\|\|r\|\|_\infty$	5,684 341 886 080 802	1,136 868 377 216 160
Абсолютная погрешность $\Delta x = x^* - x_{\text{exact}}$		
$\|\|\Delta x\|\|_1$	0,000 000 000 000 000	0,000 000 000 000 000
$\|\|\Delta x\|\|_\infty$	0,000 000 000 000 000	0,000 000 000 000 000
Относительная погрешность		
$\|\|\Delta x\|\|_1 / \|\|x_{\text{exact}}\|\|_1$	0,000 000 000 000 000	0,000 000 000 000 000
$\|\|\Delta x\|\|_\infty / \|\|x_{\text{exact}}\|\|_\infty$	0,000 000 000 000 000	0,000 000 000 000 000

Примечание: $x_{\text{exact}} = (-7, 7, 2, -6)^T$. Нормы $||x_{\text{exact}}||_1 = 22.0$, $||x_{\text{exact}}||_\infty = 7.0$. Невязки и погрешности близки к машинной эпсилон ($\approx 2.2 \times 10^{-16}$ для double).

Обратная матрица A_{good}^{-1}: Вычисленная обратная матрица A_{good}^{-1} успешно прошла проверку $A_{\text{good}} \cdot A_{\text{good}}^{-1} \approx E$ с допуском 10^{-9}.

Таблица 2. Нормы и число обусловленности для A_{good}

Величина	Значение
$\|\|A_{\text{good}}\|\|_1$	152,397 000 0
$\|\|A_{\text{good}}\|\|_\infty$	130,450 000 0
$\|\|A_{\text{inv\_good}}\|\|_1$	0,129 388 5
$\|\|A_{\text{inv\_good}}\|\|_\infty$	0,150 626 0
$cond_1(A_{\text{good}})$	19,717 840 0
$cond_\infty(A_{\text{good}})$	19,649 733 0

5.2 Плохо обусловленная система

Таблица 3. Результаты для плохо обусловленной системы

Величина	Метод Гаусса	Метод LU*
x_0^*	0,999 999 999 999 801	0,999 999 999 999 801
x_1^*	1,999 999 999 999 804	1,999 999 999 999 804
x_2^*	19,999 999 999 998 579	19,999 999 999 998 579
x_3^*	21,999 999 999 998 398	21,999 999 999 998 398
Невязка $r = Ax^* - b$		
$\|\|r\|\|_1$	1,307 313 069 701 195	1,307 313 069 701 195
$\|\|r\|\|_\infty$	3,410 605 131 648 481	3,410 605 131 648 481
Абсолютная погрешность $\Delta x = x^* - x_{\text{exact}}$		
$\|\|\Delta x\|\|_1$	5,018 604 113 254 696	5,018 604 113 254 696
$\|\|\Delta x\|\|_\infty$	1,601 779 695 294 681	1,601 779 695 294 681
Относительная погрешность		
$\|\|\Delta x\|\|_1 / \|\|x_{\text{exact}}\|\|_1$	1,115 245 358 499 933	1,115 245 358 499 933
$\|\|\Delta x\|\|_\infty / \|\|x_{\text{exact}}\|\|_\infty$	7,280 816 796 794 005	7,280 816 796 794 005

Примечание: $x_{\text{exact}} = (1, 2, 20, 22)^T$. Нормы $||x_{\text{exact}}||_1 = 45.0$, $||x_{\text{exact}}||_\infty = 22.0$. *Метод LU без перестановок дал те же результаты, что и метод Гаусса. В общем случае это не гарантировано для плохо обусловленных матриц.

Обратная матрица A_{bad}^{-1}: Вычисленная обратная матрица A_{bad}^{-1} прошла проверку $A_{\text{bad}} \cdot A_{\text{bad}}^{-1} \approx E$ с увеличенным допуском 10^{-5}.

Таблица 4. Нормы и число обусловленности для A_{bad}

Величина	A_{bad}	$A_{\text{inv\_bad}}$
1-норма	$1{,}523\,717 \times 10^3$	$1{,}372\,911 \times 10^6$
∞-норма	$2{,}436\,887 \times 10^3$	$8{,}722\,867 \times 10^5$
Число обусловленности	Значение	
$cond_1(A_{\text{bad}})$	$2{,}091\,931 \times 10^9$	
$cond_\infty(A_{\text{bad}})$	$2{,}125\,903 \times 10^9$	

5.3 Трехдиагональная система

Таблица 5. Результаты для трехдиагональной системы (Метод прогонки)

Величина	Значение
x_0^*	0,090 929 244 331 559
x_1^*	0,100 000 000 000 000
x_2^*	0,095 918 367 346 939
x_3^*	0,102 564 102 564 103
x_4^*	0,089 873 089 873 090
x_5^*	0,091 681 188 840 281
Невязка $r = Tx^* - d$	
$\|\|r\|\|_1$	4,440 892 098 500 626
$\|\|r\|\|_\infty$	2,220 446 049 250 313

6 Анализ полученных результатов

В ходе выполнения работы были решены три системы линейных алгебраических уравнений: одна хорошо обусловленная, одна плохо обусловленная и одна с трехдиагональной матрицей.

1. Хорошо обусловленная система:

- Оба метода (Гаусса с выбором главного элемента и LU-разложение без перестановок) дали решение, совпадающее с точным в пределах точности представления чисел ‘double‘ (абсолютная и относительная погрешности близки к машинной эпсилон, см. Таблицу 1).
- Нормы невязки ($||r||_1 \approx 1.1 \times 10^{-13}$, $||r||_\infty \approx 5.7 \times 10^{-14}$) также очень малы, что соответствует высокой точности решения.
- Вычисленные числа обусловленности ($cond_1 \approx 19.7$, $cond_\infty \approx 19.6$, см. Таблицу 2) являются малыми. Это подтверждает хорошую обусловленность матрицы и объясняет высокую точность полученных решений и малость невязок.
- Проверка $A \cdot A^{-1} = E$ прошла успешно, что свидетельствует о корректности вычисления обратной матрицы.

2. Плохо обусловленная система:

- Методы Гаусса и LU (в данном случае, без пивотирования) также дали решения, очень близкие к точному. Абсолютные погрешности ($||\Delta x||_1 \approx 5.0 \times 10^{-12}$, $||\Delta x||_\infty \approx 1.6 \times 10^{-12}$) и относительные погрешности ($||\Delta x||_1/||x||_1 \approx 1.1 \times 10^{-13}$, $||\Delta x||_\infty/||x||_\infty \approx 7.3 \times 10^{-14}$) оказались весьма малыми (см. Таблицу 3).
- Нормы невязки ($||r||_1 \approx 1.3 \times 10^{-9}$, $||r||_\infty \approx 3.4 \times 10^{-10}$) также малы, хотя и на несколько порядков больше, чем у хорошо обусловленной системы.
- Ключевое наблюдение: Несмотря на малые невязки и погрешности в данном конкретном случае, вычисленные числа обусловленности ($cond_1 \approx 2.1 \times 10^9$, $cond_\infty \approx 2.1 \times 10^9$, см. Таблицу 4) огромны. Это указывает на то, что система крайне чувствительна к малейшим изменениям входных данных или ошибкам округления. Получение точного решения здесь скорее удача, связанная с конкретными числами и использованием ‘double‘ точности. Небольшое изменение коэффициентов матрицы A или вектора b могло бы привести к катастрофической потере точности решения, даже при сохранении малой невязки. Малая невязка для плохо обусловленной системы не гарантирует близость вычисленного решения к истинному.
- Проверка $A \cdot A^{-1} = E$ потребовала увеличения допуска до 10^{-5}, что также косвенно указывает на проблемы с точностью вычислений из-за плохой обусловленности.

- Стоит отметить, что метод LU без перестановок строк (пивотирования) может быть неустойчивым для плохо обусловленных матриц и мог бы дать совершенно неверный результат или вовсе не сработать (из-за деления на малое число). В данном случае он сработал так же, как метод Гаусса с пивотированием, но это не общее правило.

3. Трехдиагональная система:

- Метод прогонки успешно нашел решение системы (см. Таблицу 5).
- Нормы невязки ($||r||_1 \approx 4.4 \times 10^{-16}$, $||r||_\infty \approx 2.2 \times 10^{-16}$) близки к машинной эпсилон, что свидетельствует о высокой точности решения, полученного этим специализированным и эффективным методом. Поскольку точное решение не было дано, анализ погрешности не проводился, но малая невязка для метода прогонки обычно указывает на высокую точность (при условии его применимости).

Вывод о влиянии числа обусловленности: Сравнение результатов для хорошо и плохо обусловленных систем наглядно демонстрирует важность числа обусловленности. Для хорошо обусловленной системы малая невязка соответствует малой погрешности. Для плохо обусловленной системы, даже если невязка мала, погрешность решения может быть значительной (хотя в данном примере погрешность оказалась мала, это не типично). Число обусловленности $cond(A)$ служит индикатором потенциальных проблем с точностью и устойчивостью решения СЛАУ. Чем больше $cond(A)$, тем сильнее могут искажаться результаты из-за ошибок округления или неточностей во входных данных.

7 Заключение

В ходе выполнения данной работы были реализованы и исследованы прямые методы решения систем линейных алгебраических уравнений: метод Гаусса с выбором главного элемента, метод LU-разложения (в варианте Дулиттла без пивотирования) и специализированный метод прогонки для трехдиагональных систем.

Были решены две СЛАУ размерности 4×4 (одна хорошо обусловленная, другая - плохо обусловленная) и одна трехдиагональная СЛАУ размерности 6×6. Для общих СЛАУ были вычислены векторы решений, векторы невязок, их 1-нормы и ∞-нормы, а также абсолютные и относительные погрешности относительно заданных точных решений. Были найдены обратные матрицы и оценены числа обусловленности исходных матриц в 1-й и ∞-й нормах. Для трехдиагональной системы было найдено решение и вычислены нормы невязки.

Основные выводы работы:

1. Реализованные методы позволяют численно решать СЛАУ. Метод Гаусса с выбором главного элемента продемонстрировал устойчивость для обеих систем (хорошо и плохо обусловленной). Метод LU без пивотирования, хотя и сработал корректно в данном конкретном примере для обеих систем, в общем случае менее надежен для плохо обусловленных матриц и может привести к большим ошибкам или сбою. Метод прогонки показал себя как эффективный и точный алгоритм для решения систем с трехдиагональной структурой матрицы.

2. Анализ результатов подтвердил критическое влияние числа обусловленности матрицы на точность и устойчивость численного решения СЛАУ. Для хорошо обусловленной системы ($cond(A) \approx 20$) оба метода дали результаты, практически совпадающие с точным решением, и малые невязки. Для плохо обусловленной системы ($cond(A) \approx 2 \times 10^9$), несмотря на получение решения с малыми погрешностями и невязками в данном примере (что может быть связано со спецификой входных данных и использованием double precision), огромное число обусловленности указывает на потенциальную неустойчивость. Малые возмущения входных данных или ошибки округления могли бы привести к значительному отклонению вычисленного решения от истинного.

3. Установлено, что малая величина невязки не всегда является достаточным критерием точности полученного решения, особенно в случае плохо обусловленных систем. Число обусловленности $cond(A)$ предоставляет более надежную информацию о чувствительности задачи к погрешностям.

Цели работы, поставленные во введении, были успешно достигнуты: изучены и программно реализованы основные прямые методы решения СЛАУ, проведено их сравнение, вычислены характеристики точности и невязки, исследовано влияние числа обусловленности на качество получаемого решения.