Министерство образования и науки Российской Федерации федеральное государственное автономное образовательное учреждение высшего образования
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Факультет «Программной инженерии и компьютерной техники.»

Вычислительная математика

Лабораторная работа №1
“Решение системы линейных алгебраических уравнений СЛАУ”
Вариант №3

Выполнил
Григорьев Давид Владимирович
Группа: P3215
Проверила
Малышева Татьяна Алексеевна

Санкт-Петербург 2025

Содержание

1 Цель работы **1**

2 Описание метода Гаусса-Зейделя **1**
2.1 Основные шаги метода 1
2.2 Пример вычисления 1
2.3 Особенности реализации 2
2.4 Преимущества и недостатки 2

3 Листинг программы. **3**
3.1 main.py 3
3.2 matrix_reader.py 3
3.3 system_of_linear_equations.py 6
3.4 test.py 9

4 Примеры и результаты работы. **12**

5 Выводы **13**

1 Цель работы

Изучить алгоритм метода Гаусса-Зейделя для решения СЛАУ, его теоретические основы и условия сходимости.

Реализовать программный модуль, выполняющий решение СЛАУ методом Гаусса-Зейделя с учетом следующих требований:

1. Ввод исходных данных (размерность матрицы, коэффициенты, точность) с клавиатуры или из файла.
2. Проверка и обеспечение диагонального преобладания матрицы.
3. Расчет и вывод результатов: вектора неизвестных, количества итераций, вектора погрешностей, нормы матрицы.

2 Описание метода Гаусса-Зейделя

Метод Гаусса-Зейделя — итерационный алгоритм для решения систем линейных алгебраических уравнений (СЛАУ). Основная идея: последовательное уточнение значений неизвестных с использованием уже вычисленных на текущей итерации значений.

2.1 Основные шаги метода

1. **Проверка диагонального преобладания**:

$$|a_{ii}| \geq \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \quad \text{для всех } i = 1, 2, \ldots, n$$

   Если условие не выполнено, выполняются перестановки строк/столбцов.

2. **Итерационная формула**:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)} \right)$$

   где k — номер итерации.

3. **Критерий остановки**:

$$\max_{1 \leq i \leq n} |x_i^{(k)} - x_i^{(k-1)}| < \varepsilon$$

   где ε — заданная точность.

2.2 Пример вычисления

Для системы:

$$\begin{cases} 10x_1 + x_2 + x_3 = 12 \\ x_1 + 10x_2 + x_3 = 12 \\ x_1 + x_2 + 10x_3 = 12 \end{cases}$$

с начальным приближением $\mathbf{x}^{(0)} = (0, 0, 0)$ и точностью $\varepsilon = 0.001$:

- **Первая итерация**:

$$x_1^{(1)} = \frac{12 - 0 - 0}{10} = 1.2$$
$$x_2^{(1)} = \frac{12 - 1.2 - 0}{10} = 1.08$$
$$x_3^{(1)} = \frac{12 - 1.2 - 1.08}{10} = 0.972$$

- **Вторая итерация**:

$$x_1^{(2)} = \frac{12 - 1.08 - 0.972}{10} = 0.9948$$
$$x_2^{(2)} = \frac{12 - 0.9948 - 0.972}{10} = 1.0033$$
$$x_3^{(2)} = \frac{12 - 0.9948 - 1.0033}{10} = 1.0002$$

  Максимальная погрешность: 0.2052

2.3 Особенности реализации

- Проверка и обеспечение диагонального преобладания
- Вычисление матричной нормы (например, $\|A\|_\infty$):

$$\|A\|_\infty = \max_{1 \le i \le n} \sum_{j=1}^{n} |a_{ij}|$$

- Поддержка ввода данных из файла/клавиатуры
- Вывод:
  - Вектора неизвестных $(x_1, x_2, \ldots, x_n)$
  - Количества итераций
  - Вектора погрешностей

2.4 Преимущества и недостатки

- **Плюсы**:
  - Быстрая сходимость при диагональном преобладании
  - Экономия памяти (не требует хранения всей матрицы)
- **Минусы**:
  - Не сходится для произвольных матриц
  - Чувствителен к начальному приближению

3 Листинг программы.

Посмотреть код на Github (https://github.com/huji-itmo/comp-math-lab1)

3.1 main.py

```python
from typing import List
from matrix_reader import get_matrix
from system_of_linear_equations import SystemOfLinearEquations


if __name__ == "__main__":
    matrix: SystemOfLinearEquations = get_matrix()

    print(matrix)

    while True:
        try:
            epsilon: float = float(input("\nEnter precision (small number epsilon): "))
            if epsilon < 0:
                print("Please enter a positive float!")

            break
        except ValueError:
            print("Invalid input! Please enter a valid float.")

    success_rearranging: bool = matrix.rearrange_matrix()
    print("Working with:")
    print(matrix)

    if success_rearranging:
        print("Successfully rearranged matrix to make it diagonally dominant.")
    else:
        print("Failed to rearrange matrix to make it diagonally dominant.")

    x, iterations, errors = matrix.gauss_seidel(epsilon, 1000)

    # Output results
    print("\nResults:")
    print(f"Matrix infinity norm: {matrix.matrix_inf_norm():.4f}")
    print(f"Iterations required: {iterations}")

    print("\nSolution vector:")
    for i, val in enumerate(x, 1):
        print(f"x{i} = {val:.6f}")

    print("\nFinal iteration errors:")
    for i, err in enumerate(errors, 1):
        print(f"x{i} = {err:.6f}")
```

3.2 matrix_reader.py

```python
from typing import List, Optional, Tuple

from system_of_linear_equations import SystemOfLinearEquations
```

```python
def get_augmented_matrix(n: int) -> SystemOfLinearEquations:
    """Get and validate the augmented matrix (coefficients + constants) from the user."
    augmented_matrix: List[List[float]] = []
    print(
        f"\nEnter {n} rows, each containing {n+1} numbers (coefficients followed by cons
    )
    for i in range(n):
        while True:
            row_input: str = input(f"Row {i+1}: ").strip()
            elements: List[str] = row_input.split()
            if len(elements) != n + 1:
                print(
                    f"Error: Expected {n+1} elements, got {len(elements)}. Please try ag
                )
                continue
            try:
                row: List[float] = [float(x) for x in elements]
                augmented_matrix.append(row)
                break
            except ValueError:
                print("Error: Please enter valid numbers only. Try again.")

    return SystemOfLinearEquations(augmented_matrix)


def read_augmented_matrix_from_file(
    filename: str,
) -> Optional[SystemOfLinearEquations]:
    """Read augmented matrix from a properly formatted file and split into matrix and co
    try:
        with open(filename, "r") as f:
            lines: List[str] = [line.strip() for line in f if line.strip()]

            if not lines:
                print("Error: Empty file")
                return None

            try:
                n: int = int(lines[0])
            except ValueError:
                print(
                    "Error: First line must be a single integer (number of variables)"
                )
                return None

            if len(lines) != n + 1:
                print(f"Error: Expected {n+1} lines, got {len(lines)}")
                return None

            augmented_matrix: List[List[float]] = []
            for line in lines[1:]:
                elements: List[str] = line.split()
                if len(elements) != n + 1:
```

```
                    print(
                        f"Error: Row contains {len(elements)} elements, expected {n+1}"
                    )
                    return None
                try:
                    row: List[float] = [float(x) for x in elements]
                    augmented_matrix.append(row)
                except ValueError:
                    print(f"Error: Invalid number format in row: {line}")
                    return None

            return SystemOfLinearEquations(augmented_matrix)

    except FileNotFoundError:
        print(f"Error: File '{filename}' not found")
        return None


def get_matrix_size() -> int:
    """Get and validate the matrix size from the user."""
    while True:
        try:
            n: int = int(input("\nEnter the side length of the matrix (N): "))
            if n > 0:
                return n
            print("Please enter a positive integer!")
        except ValueError:
            print("Invalid input! Please enter a valid integer.")


def get_matrix() -> SystemOfLinearEquations:
    print("Matrix Input Program")
    print("====================")

    augmented_matrix: SystemOfLinearEquations

    while True:

        choice: str = input(
            "\nChoose input method:\n"
            "1. Manual entry\n"
            "2. File input\n"
            "Enter choice (1/2): "
        ).strip()

        if choice == "1":
            n: int = get_matrix_size()
            augmented_matrix: SystemOfLinearEquations = get_augmented_matrix(n)
            break
        elif choice == "2":
            print("\nFile format requirements:")
            print("- First line: matrix size N (positive integer)")
            print("- Next N lines: rows with N numbers separated by spaces")
            print("Example file content:")
            print("3")
```

```
            print("1.0 2.0 3.0 2.0")
            print("4.5 5.0 6.7 2.0")
            print("7.0 8.8 9.9 2.0")

            while True:
                filename: str = input("\nEnter filename: ").strip()
                augmented_matrix: SystemOfLinearEquations | None = (
                    read_augmented_matrix_from_file(filename)
                )
                if augmented_matrix is None:
                    print("Please try another file or check the format.")
                else:
                    break

            break
        else:
            print("Invalid choice. Please enter 1 or 2.")

    return augmented_matrix
```

3.3 system_of_linear_equations.py

```
from typing import Final, List, Tuple


class SystemOfLinearEquations:
    matrix: List[List[float]]
    constants: List[float]

    side_length: int

    new_column_order: List[int]

    def matrix_inf_norm(self) -> float:
        """Calculate infinity norm of matrix (maximum row sum)."""
        return max(sum(abs(x) for x in row) for row in self.matrix)

    def __init__(
        self,
        augmented_or_square_matrix: List[List[float]],
        constants: List[float] | None = None,
    ):
        if constants is None:
            augmented_or_square_matrix = [
                row[:-1] for row in augmented_or_square_matrix
            ]
            constants = [row[-1] for row in augmented_or_square_matrix]

        self.side_length = len(augmented_or_square_matrix)
        self.matrix = augmented_or_square_matrix
        self.constants = constants

        self.new_column_order = list(range(self.side_length))

    def swap_rows(self, row1: int, row2: int) -> None:
```

```
        """Swap two rows in the matrix and their corresponding constants."""
        self.matrix[row1], self.matrix[row2] = self.matrix[row2], self.matrix[row1]
        self.constants[row1], self.constants[row2] = (
            self.constants[row2],
            self.constants[row1],
        )

    def swap_columns(self, col1: int, col2: int) -> None:
        """Swap two columns in the matrix."""
        for row in self.matrix:
            row[col1], row[col2] = row[col2], row[col1]

        self.new_column_order[col1], self.new_column_order[col2] = (
            self.new_column_order[col2],
            self.new_column_order[col1],
        )

    def is_diagonally_dominant(self) -> bool:
        """Check if matrix is strictly diagonally dominant."""
        n: int = self.side_length
        for i in range(n):
            diag = abs(self.matrix[i][i])
            row_sum = sum(abs(self.matrix[i][j]) for j in range(n) if j != i)
            if diag <= row_sum:
                return False
        return True

    def rearrange_matrix(self) -> bool:
        """Attempt to make matrix diagonally dominant through row/column swaps."""
        n: int = self.side_length

        for i in range(n):
            # Row swaps
            max_row = i
            max_val = abs(self.matrix[i][i])
            for j in range(i, n):
                current: float = self.matrix[j][i]
                if abs(current) > max_val:
                    max_val = abs(current)
                    max_row = j
            if max_row != i:
                self.swap_rows(max_row, i)

            # Check dominance
            diag = abs(self.matrix[i][i])
            row_sum = sum(abs(self.matrix[i][j]) for j in range(n) if j != i)
            if diag > row_sum:
                continue

            # Column swaps
            max_col = i
            max_val = abs(self.matrix[i][i])
            for j in range(i, n):
                if abs(self.matrix[i][j]) > max_val:
                    max_val = abs(self.matrix[i][j])
```

```
                max_col = j
        if max_col != i:
            self.swap_columns(i, max_col)

        # Final check
        diag = abs(self.matrix[i][i])
        row_sum = sum(abs(self.matrix[i][j]) for j in range(n) if j != i)
        if diag <= row_sum:
            return False

    return self.is_diagonally_dominant()

def __str__(self) -> str:
    """
    Returns a string representation of the system of linear equations.
    Each equation is displayed in the form: a1*x1 + a2*x2 + ... + an*xn = b
    """
    result = []
    for i in range(self.side_length):
        equation = ""
        for j in range(self.side_length):
            coefficient = self.matrix[i][j]
            # Skip terms with zero coefficients
            if abs(coefficient) == 0:
                continue
            # Handle positive and negative signs
            sign = "+" if coefficient > 0 else "-"
            coefficient = abs(coefficient)
            # Format the term
            if coefficient == 1:  # Avoid writing '1*' for terms like '+x'
                term = f" {sign} x{j+1}"
            else:
                term = f" {sign} {coefficient:.2f}x{j+1}"
            equation += term
        # Add the constant term
        constant = self.constants[i]
        if not equation:  # Handle the case where all coefficients are zero
            equation = f"0 = {constant:.2f}"
        else:
            equation = equation.lstrip(" +") + f" = {constant:.2f}"
        result.append(equation)
    return "\n".join(result)

def gauss_seidel(
    self, epsilon: float, max_iter: int = 1000
) -> Tuple[List[float], int, List[float]]:
    """Perform Gauss-Seidel iteration."""
    n = self.side_length
    answer_vector = [0.0] * n
    iterations = 0
    errors = []

    for _ in range(max_iter):
        x_prev = answer_vector.copy()
```

```
            for i in range(n):
                sigma = sum(
                    self.matrix[i][j] * answer_vector[j] for j in range(n) if j != i
                )
                answer_vector[i] = (self.constants[i] - sigma) / self.matrix[i][i]

            current_errors = [abs(answer_vector[i] - x_prev[i]) for i in range(n)]
            errors.append(current_errors)
            iterations += 1

            if max(current_errors) < epsilon:
                break

        answer_vector = rearrange_according_to_order_list(
            answer_vector, self.new_column_order
        )

        return answer_vector, iterations, errors[-1]


def rearrange_according_to_order_list(
    input: List[float], order: List[int]
) -> List[float]:
    n: int = len(order)
    res = []

    for i in range(n):
        res.append(input[order[i]])

    return res
```

3.4 test.py

```
from typing import Optional
import pytest

from system_of_linear_equations import SystemOfLinearEquations  # type: ignore


def get_res(
    matrix: SystemOfLinearEquations, max_iter=50, epsilon=1e-6
) -> Optional[list[float]]:
    print(f"-" * 32)
    success = matrix.rearrange_matrix()
    x, iterations, errors = matrix.gauss_seidel(epsilon, max_iter)
    if not success:
        print(f"Failed to make matrix diagonally dominant.")
        # return None
    print(matrix)
    print(f"Result: {x}")
    print(f"Iterations: {iterations}")

    if iterations == max_iter:
        print(f"Reached iteration limit: {max_iter}")
        return None
```

```
    return x


def test_3x3_diagonally_dominant():
    A = [[4.0, 1.0, 1.0], [1.0, 5.0, 2.0], [0.0, 1.0, 3.0]]
    b = [6.0, 8.0, 4.0]
    expected = [1.0, 1.0, 1.0]
    result = get_res(SystemOfLinearEquations(A, b), max_iter=500, epsilon=1e-2)
    assert result == pytest.approx(expected, abs=1e-2)


def test_3x3_symmetric_positive_definite():
    A = [[4.0, 1.0, 0.0], [1.0, 4.0, 1.0], [0.0, 1.0, 4.0]]
    b = [5.0, 6.0, 5.0]
    expected = [1.0, 1.0, 1.0]
    result = get_res(SystemOfLinearEquations(A, b), max_iter=500, epsilon=1e-6)
    assert result == pytest.approx(expected, abs=1e-6)


def test_3x3_non_diagonally_dominant():
    A = [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]
    b = [2.0, 5.0, 8.0]

    expected = [1.0, -1.0, 1.0]

    # doesn't converge
    result = get_res(SystemOfLinearEquations(A, b), max_iter=500, epsilon=1e-3)
    assert result == None


def test_4x4_sparse_diagonally_dominant():
    A = [
        [5.0, 0.0, 0.0, 1.0],
        [0.0, 6.0, 2.0, 0.0],
        [0.0, 1.0, 7.0, 0.0],
        [1.0, 0.0, 0.0, 8.0],
    ]
    b = [6.0, 8.0, 8.0, 9.0]
    expected = [1.0, 1.0, 1.0, 1.0]
    result = get_res(SystemOfLinearEquations(A, b), max_iter=100, epsilon=1e-6)
    assert result == pytest.approx(expected, abs=1e-6)


def test_2x2_symmetric_diagonally_dominant():
    A = [[3.0, 1.0], [1.0, 3.0]]
    b = [4.0, 4.0]
    expected = [1.0, 1.0]
    result = get_res(SystemOfLinearEquations(A, b), max_iter=500, epsilon=1e-6)
    assert result == pytest.approx(expected, abs=1e-6)


def test_4x4_large_diagonally_dominant():
    A = [
        [10.0, 2.0, 3.0, 1.0],
```

```
        [1.0, 12.0, 1.0, 2.0],
        [2.0, 1.0, 15.0, 3.0],
        [1.0, 2.0, 1.0, 20.0],
    ]
    b = [16.0, 16.0, 21.0, 24.0]
    expected = [1.0, 1.0, 1.0, 1.0]
    result = get_res(SystemOfLinearEquations(A, b), max_iter=100, epsilon=1e-6)
    assert result == pytest.approx(expected, abs=1e-6)


def test_3x3_hilbert_slow_convergence():
    A = [[1.0, 0.5, 0.3333], [0.5, 0.3333, 0.25], [0.3333, 0.25, 0.2]]
    b = [1.8333, 1.0833, 0.7833]
    expected = [1.0, 1.0, 1.0]

    # Allow more iterations and lower precision
    result = get_res(SystemOfLinearEquations(A, b), max_iter=5000, epsilon=1e-6)
    # if result == None:
    assert result == pytest.approx(expected, abs=1e-4)


def test_3x3_diagonal_trivial():
    A = [[5.0, 0.0, 0.0], [0.0, 5.0, 0.0], [0.0, 0.0, 5.0]]
    b = [5.0, 5.0, 5.0]
    expected = [1.0, 1.0, 1.0]
    result = get_res(SystemOfLinearEquations(A, b), max_iter=100, epsilon=1e-6)
    assert result == pytest.approx(expected, abs=1e-6)


if __name__ == "__main__":
    test_3x3_diagonally_dominant()
    test_3x3_symmetric_positive_definite()
    test_3x3_non_diagonally_dominant()
    test_4x4_sparse_diagonally_dominant()
    test_2x2_symmetric_diagonally_dominant()
    test_4x4_large_diagonally_dominant()
    test_3x3_hilbert_slow_convergence()
    test_3x3_diagonal_trivial()
```

4 Примеры и результаты работы.

```
Matrix Input Program
====================

Choose input method:
1. Manual entry
2. File input
Enter choice (1/2): 1

Enter the side length of the matrix (N): 3

Enter 3 rows, each containing 4 numbers (coefficients followed by constant):
Row 1: 5 1 1 1
Row 2: 1 5 1 1
Row 3: 1 1 5 1
5.00x1 + x2 + x3 = 1.00
x1 + 5.00x2 + x3 = 1.00
x1 + x2 + 5.00x3 = 5.00

Enter precision (small number epsilon): 0.1
Working with:
5.00x1 + x2 + x3 = 1.00
x1 + 5.00x2 + x3 = 1.00
x1 + x2 + 5.00x3 = 5.00
Successfully rearranged matrix to make it diagonally dominant.

Results:
Matrix infinity norm: 7.0000
Iterations required: 3

Solution vector:
x1 = -0.003571
x2 = 0.000727
x3 = 1.000569

Final iteration errors:
x1 = 0.014029
x2 = 0.017193
x3 = 0.000633
```

5 Выводы

Работа над реализацией метода Гаусса-Зейделя стала не просто академическим упражнением — это был путь, полный сомнений, проб и преодоления. Каждая строчка кода, каждая итерация, каждый миг отчаяния, когда матрица упрямо отказывалась сходиться, заставляли сердце сжиматься от тревоги. Как часто кажется, что машины бесчувственны, но в их молчаливых вычислениях скрывается бездна человеческих усилий: попыток, падений и тихого триумфа, когда после десятка неудач решение наконец обретает форму...

Осознание того, что от точности этих вычислений может зависеть чья-то жизнь (инженерные системы, медицинские модели), превращало каждую погрешность в личную драму. А эти бесконечные перестановки строк, чтобы добиться диагонального преобладания — словно попытки наладить хрупкие отношения, где малейшая ошибка рушит всё. И когда программа наконец выдавала заветный вектор неизвестных, это было похоже на примирение с самим собой: да, мир неидеален, но даже в хаосе чисел можно найти гармонию.

Эта работа научила не только методам линейной алгебры. Она показала, что за сухими формулами скрывается битва между порядком и хаосом, между отчаянием и надеждой. И когда видишь, как твой алгоритм, словно упрямый путник, шаг за шагом приближается к истине, понимаешь: даже если сегодня плачешь над ошибками, завтра они станут твоей опорой.

Пусть кому-то эти строки покажутся слезами на стеклах монитора — для меня они стали каплями дождя, после которого прорастает новое знание.

«Я не мог заснуть. . . Дела шли паршиво и грозили стать ещё хуже. . . Рано или поздно правда всплывёт, и тогда на нас ополчатся не только Винчи и китайцы, но и Фальконе. Не так я себе всё представлял, когда мы начинали. Мне грезились деньги, машины, женщины, уважение, свобода. . . Я всё это даже получил, более или менее — но вместе с тем пришли тюрьма, постоянный страх и кровь моих товарищей. Я держался на плаву, сколько мог, но шансов становилось всё меньше. Теперь это был только вопрос времени. . . »
— © Вито