Национальный исследовательский университет ИТМО
Факультет информационных технологий и программирования
Прикладная математика и информатика

Методы решения СЛАУ

Отчёт по лабораторной работе №4

Работу выполнили:
Гуревич Михаил
Трохан Александр

Преподаватель:
Москаленко Мария Александровна

Санкт-Петербург
2023

Лабораторная работа №4

Выполнили: Гуревич Михаил и Трохан Александр, М32001

✅ Полный код лабораторной работы с комментариями можно найти на **Github**, а отчёт в **Notion**.

Цель работы

Анализ прямых и итерационных методов решения систем линейных уравнений.

Постановка задачи

1. Реализовать различные методы решения систем линейных алгебраических уравнений.
2. Провести исследование методов на матрицах с определённым числом обусловленности.
3. Провести исследование на матрице Гильберта.
4. Сравнить методы по эффективности в зависимости от размеров n матрицы.

Ход работы

Реализация методов

Для начала реализуем метод для определения вырожденности матрицы, так как системы с такими матрицами не будут иметь однозначного решения:

```
def is_singular(matrix: np.ndarray):
    return np.linalg.det(matrix) == 0
```

Метод Гаусса с выбором ведущего элемента

Описание метода: метод похож на обычный метод Гаусса, который применяется для решения систем линейных уравнений, то есть сначала матрица приводится к треугольному виду, а потом обратным ходом находятся решения системы. Однако здесь на каждой итерации выбирается наибольший элемент в столбце, и затем вся строка делится на этот коэффициент. Деление на наибольший возможный коэффициент снижает погрешность и избавляет от ситуаций, когда на диагонали находится 0.

Реализация метода:

```
def solve_Gauss(A: np.ndarray, b: np.ndarray):
    if is_singular(A):
        raise ValueError("Matrix is singular")
    A = np.c_[A, b]  # присоединяем столбец b к матрице A

    n = A.shape[0]
```

```
    for k in range(n):
        pivot = k + np.argmax(abs(A[k:, k]))
        if pivot != k:
            A[k], A[pivot] = A[pivot], np.copy(A[k])
        current_row = A[k]
        A[k] /= current_row[k]
        for lower_row in A[k + 1:]:
            lower_row -= lower_row[k] * current_row

    for k in range(n - 1, 0, -1):
        for upper_row in A[:k]:
            upper_row[-1] -= upper_row[k] * A[k, -1]
            upper_row[k] = 0

    return A[:, -1]
```

Метод LU-разложения

Помимо самого метода, в работе требуется реализовать хранение матриц с использованием разреженно-строчного формата. Класс для такой матрицы был реализован следующим образом:

```
class CSRMatrix:
    def __init__(self, data, indices, indptr, shape):
        self.data = data
        self.indices = indices
        self.indptr = indptr
        self.shape = shape

    @staticmethod
    def from_dense(matrix: np.ndarray):
        data = []
        indices = []
        indptr = [0]
        for row in matrix:
            for i, elem in enumerate(row):
                if elem != 0:
                    data.append(elem)
                    indices.append(i)
            indptr.append(len(data))
        return CSRMatrix(data, indices, indptr, matrix.shape)

    def to_dense(self):
        matrix = np.zeros(self.shape)
        for i in range(self.shape[0]):
            for j in range(self.indptr[i], self.indptr[i + 1]):
                matrix[i, self.indices[j]] = self.data[j]
        return matrix

    def get_arrays(self):
        return self.data, self.indices, self.indptr, self.shape
```

Описание метода: метод заключается в нахождении LU-разложения матрицы A, где L и U — нижняя и верхняя треугольная матрица соответсвенно. Тогда решение системы вида $Ax = b$ сводится к решению системы $Ly = b$, а затем $Ux = y$.

Реализация метода:

```
def LU_decomposition(matrix: CSRMatrix):
    matrix = matrix.to_dense()
    n = matrix.shape[0]
    L = np.eye(n)
```

```
    U = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            if j >= i:
                U[i, j] = matrix[i, j] - sum(L[i, k] * U[k, j] for k in range(i))
            else:
                L[i, j] = (matrix[i, j] - sum(L[i, k] * U[k, j] for k in range(j))) / U[j, j]

    return L, U

def solve_LU(A: np.ndarray, b: np.ndarray):
    if is_singular(A):
        raise ValueError("Matrix is singular")

    L, U = LU_decomposition(CSRMatrix.from_dense(A))
    n = A.shape[0]
    y = np.zeros(n)
    for i in range(n):
        y[i] = b[i] - sum(L[i, j] * y[j] for j in range(i))

    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        x[i] = (y[i] - sum(U[i, j] * x[j] for j in range(i + 1, n))) / U[i, i]
    return x
```

Метод Зейделя

Метод Зейделя гарантированно сходится для симметричных, положительно определенных матриц или для матриц со строгим диагональным преобладанием.

Описание метода: идея метода заключается в том, что на каждой итерации будет вычисляться решение по формуле: $L_* x^{k+1} = b - U x^k$, при этом матрицы L_* и U удовлетворяют условию

$$A = L_* + U = \begin{pmatrix} a_{11} & 0 & ... & 0 \\ a_{22} & a_{22} & ... & 0 \\ ... & & & \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix} + \begin{pmatrix} 0 & a_{12} & ... & a_{1n} \\ 0 & 0 & ... & a_{2n} \\ ... & & & \\ 0 & 0 & ... & 0 \end{pmatrix}$$

где $Ax = b$ — данная для решения система уравнений.

Ввиду этого разложения, сам вектор x^{k+1} можно записать как две суммы элементов матрицы:

$$x^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^k \right), \; i = 0, 1, 2, ..., n$$

Метод выполняется, пока не будет достигнута требуемая точность, то есть до выполнения условия $||x^{k+1} - x^k|| \leq \varepsilon$.

Реализация метода: в данной реализации также добавлено ограничение на количество итераций.

```
def solve_Seidel(A: np.ndarray, b: np.ndarray, epsilon=1e-6, max_iterations=2e6):
    n = A.shape[0]
    x = np.zeros(n)
    x_new = np.zeros(n)
    for _ in range(int(max_iterations)):
        for i in range(n):
            x_new[i] = b[i] - sum(A[i, j] * x_new[j] for j in range(i)) - sum(A[i, j] * x[j] for j in range(i + 1, n))
            x_new[i] /= A[i, i]
```

```
        if np.linalg.norm(x_new - x) < epsilon:
            break
        x = np.copy(x_new)
    else:
        print("Max iterations exceeded")

    return x_new
```

Исследование методов на матрицах с диагональным преобладанием

Для начала реализуем функцию для генерации матриц согласно требуемым условиям:

```
def generate_diag(n, k):
    matrix = np.array([[np.random.randint(-4, 1) for _ in range(n)] for _ in range(n)], dtype=float)

    for i in range(n):
        if i == 0:
            matrix[i, i] = -sum(matrix[i, j] for j in range(n) if j != i) + 10 ** (-k)
        else:
            matrix[i, i] = -sum(matrix[i, j] for j in range(n) if j != i)

    return matrix
```

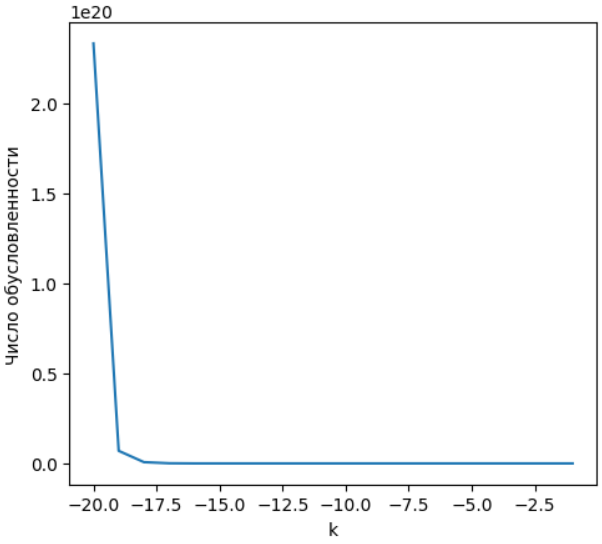
А также функцию для генерации коэффициентов правой части уравнения $Ax = b$, чтобы искомый вектор x имел вид $x = (1, 2, \ldots, n)^T$:

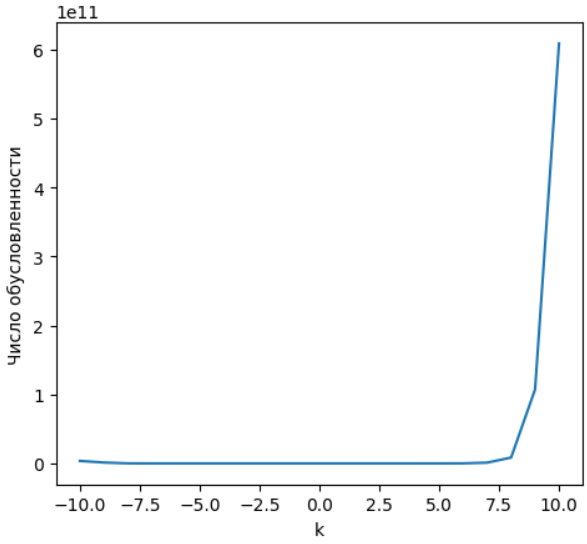
```
def generate_b(matrix: np.ndarray):
    n = matrix.shape[0]
    return matrix @ np.array(np.arange(1, n + 1, 1), dtype=np.float64)
```

Зависимость числа обусловленности от k

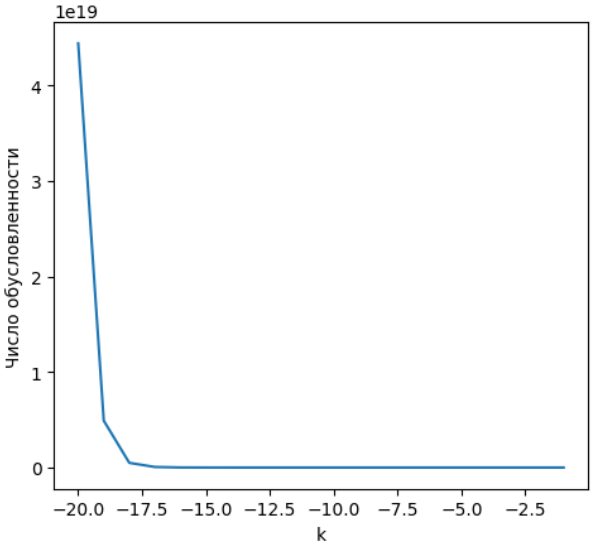
Для начала рассмотрим зависимость числа обусловленности от k при разных конкретных значениях n:

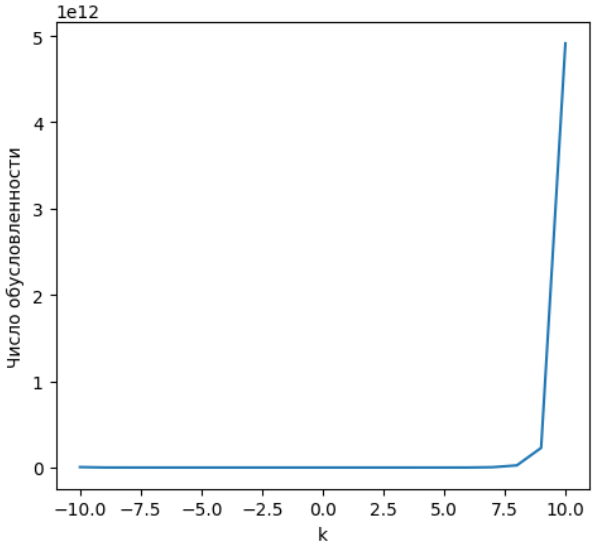
Зависимость числа обусловленности от k при n = 5

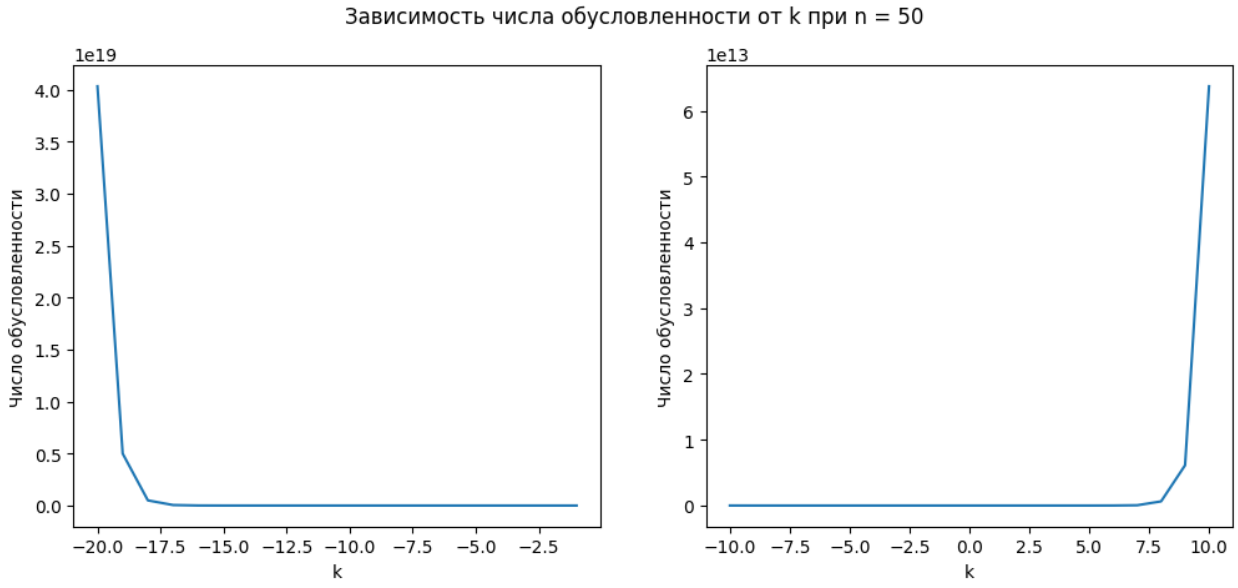




Зависимость числа обусловленности от k при n = 10

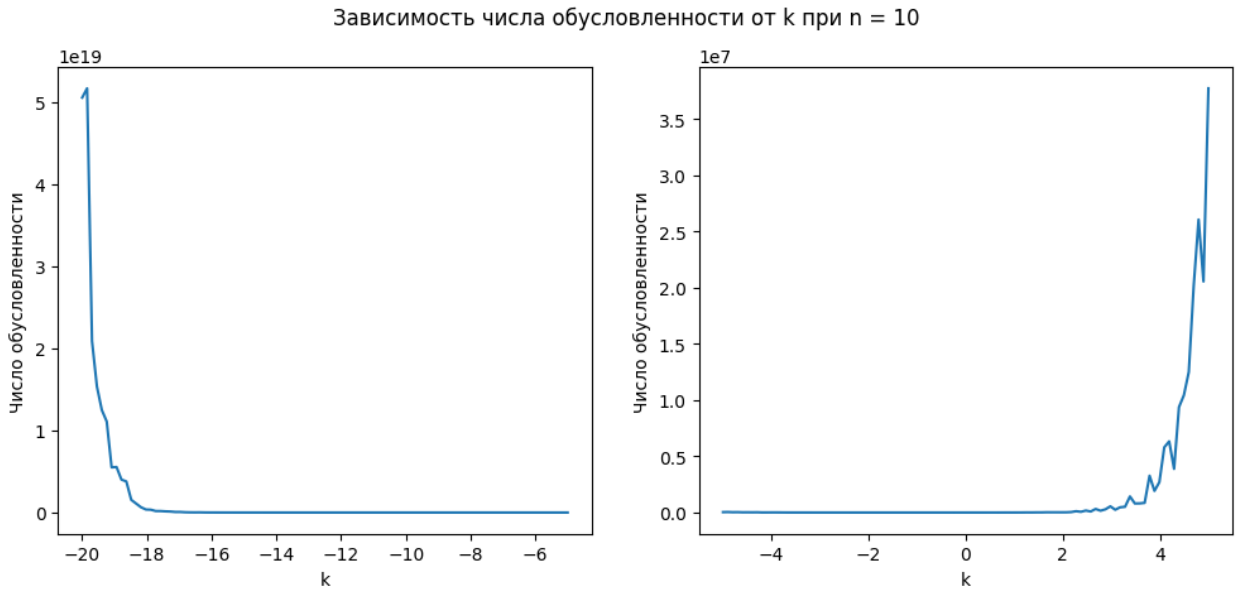






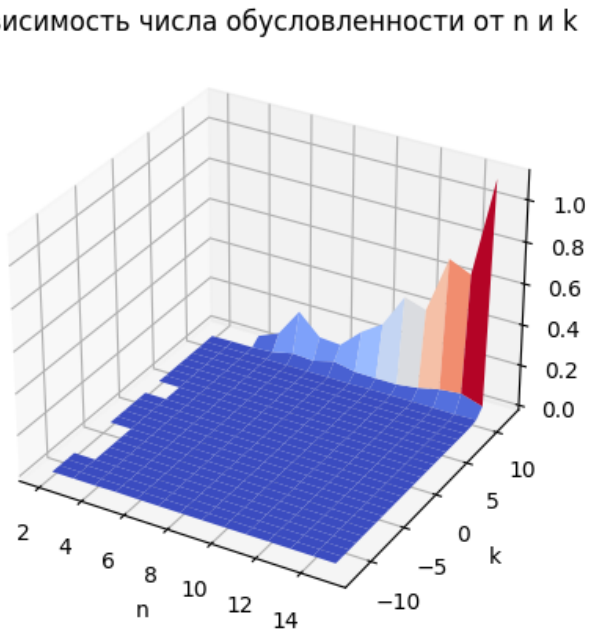
Во всех случаях можно наблюдать, что при увеличении k по модулю число обусловленности очень быстро возрастает.

На графиках выше рассматривались только целые k, посмотрим как ведёт себя график на действительных числах:



В целом, зависимость сохраняется, однако иногда число обусловленности всё таки уменьшается. Это можно объяснить случайным выбором элементов в матрице.

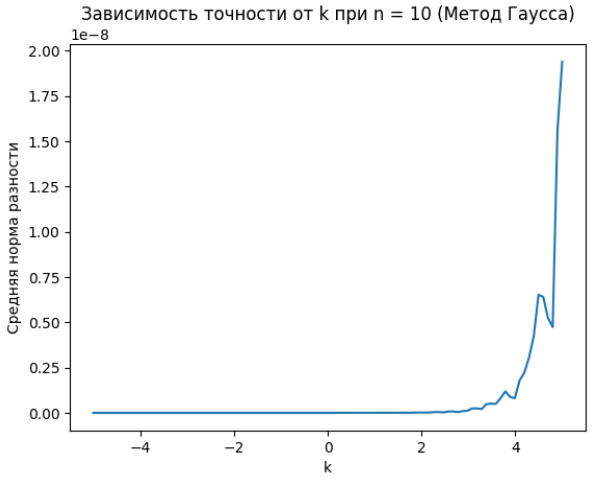
Теперь можно посмотреть на трёхмерный график зависимости числа обусловленности от n и k:

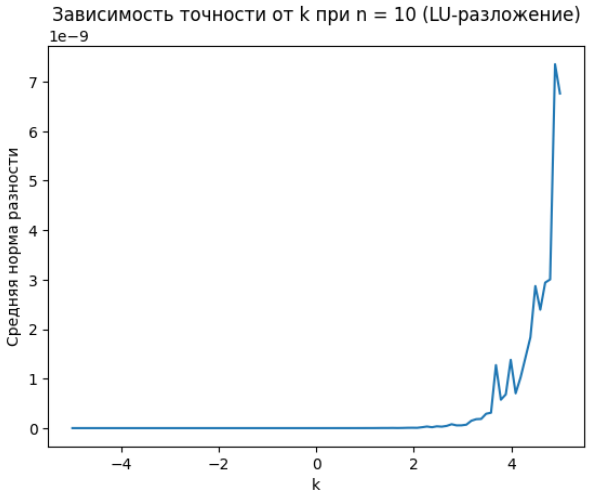


По нему видно, что число обусловленности растёт и при росте n, что логично, так как увеличивается количество элементов в строках и возрастает диагональное преобладание.

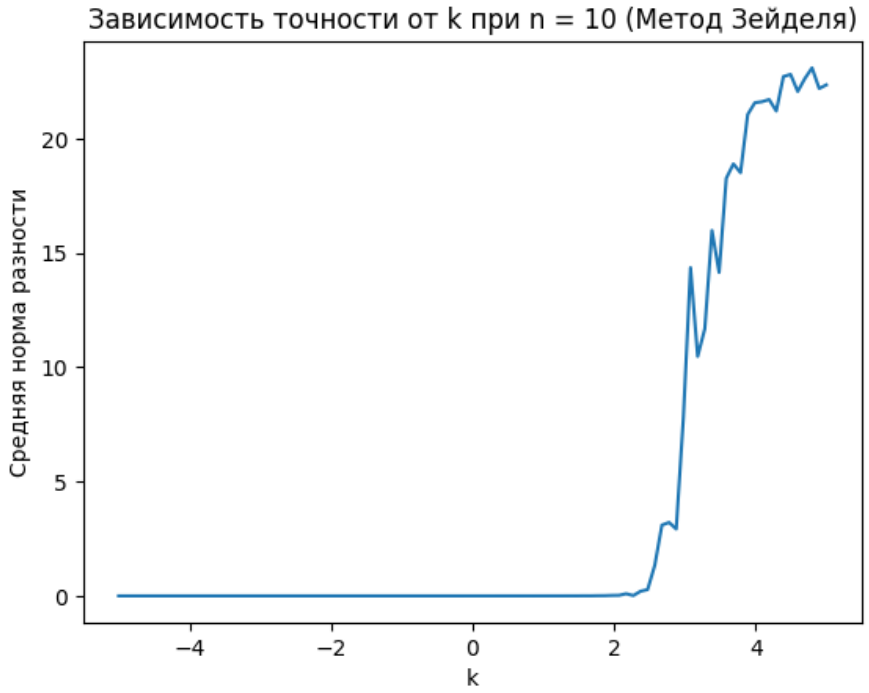
Зависимость точности от k

В случае прямых методов (метода Гаусса и LU-разложения) точность падает с ростом k, однако незначительно:





Однако в случае итерационного метода Зейделя точность значительно падает с ростом k:



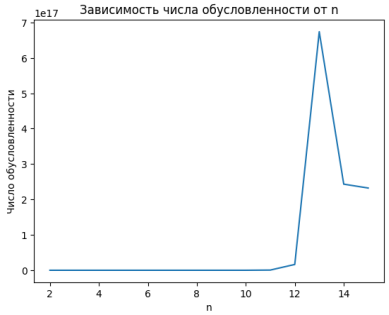
Также стоит отметить, что в некоторых ситуациях (особенно при больших k) метод не сходился вообще.

Исследование матриц Гильберта

Генерация матриц:

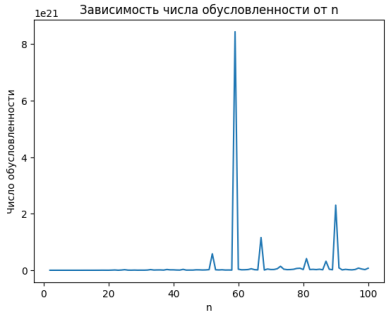
```
def generate_Hilbert(n):
    return np.array([[1 / (i + j + 1) for j in range(n)] for i in range(n)], dtype=float)
```

Зависимость числа обусловленности от n



$n \in [2; 15],\ n \in \mathbb{Z}$

$n \in [15; 30],\ n \in \mathbb{Z}$

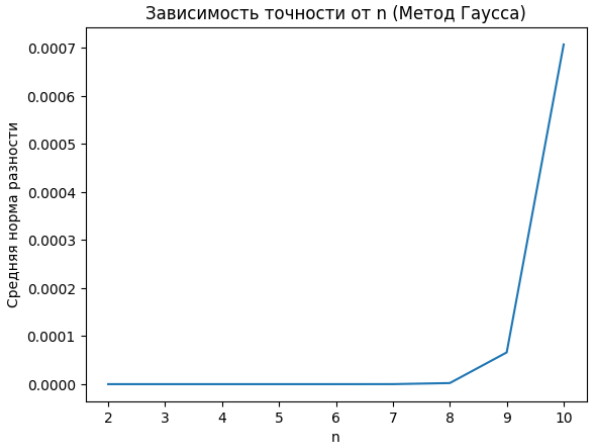


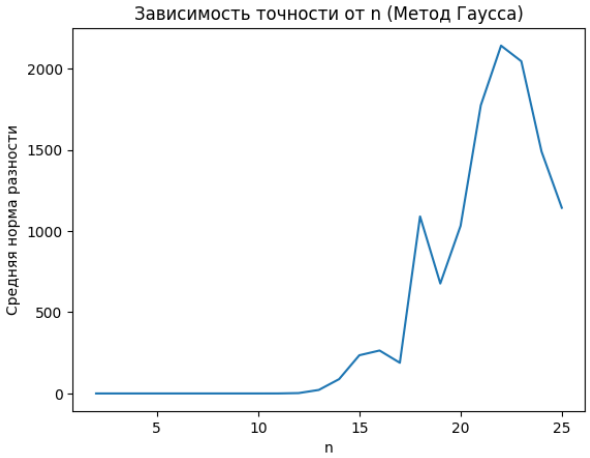
$n \in [2; 100],\ n \in \mathbb{Z}$

Зависимость числа обусловленности матрицы Гильберта от n нелинейна, однако можно сказать, что число обусловленности растёт с увеличением n.

Зависимость точности от n

Как и в случае с матрицами с диагональным преобладанием, которые были рассмотрены выше, точность прямых методов падает с увеличением n:



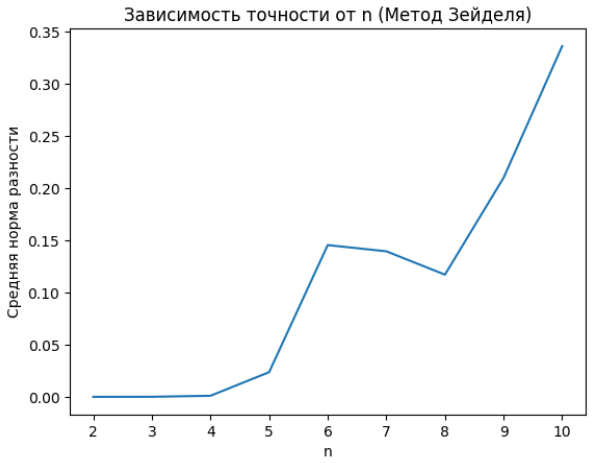


$n \in [2; 10],\ n \in \mathbb{Z}$

$n \in [2; 26],\ n \in \mathbb{Z}$

Но в данном случае точность падает колоссально, в то время как в предыдущем случае она падала незначительно.

Ещё одним отличием является то, что точность итерационного метода теперь выше, чем точность прямых методов при больших n:



$n \in [2; 10],\ n \in \mathbb{Z}$

$n \in [2; 26],\ n \in \mathbb{Z}$

Это можно объяснить тем, что в каждом из прямых методов на некотором этапе происходит деление коэффициентов уравнения, а так как в случае с матрицами Гильберта коэффициентами являются обыкновенные дроби, которые неточно представлены в памяти компьютера, точность сильно падает. В то же время метод Зейделя не использует деление, и точность остаётся приемлемой.

Эффективность методов

Сравним методы по скорости работы на матрицах с диагональным преобладанием размерностью $n \in \{10, 50, 100, 1000, 10000\}$:

- **Метод Гаусса:**

  Первый прогрессбар — прямой ход, второй — обратный ход

```
Processing n = 10 ...
```


```
n = 10, time = 0:00:00.019866
Processing n = 50 ...
```


```
n = 50, time = 0:00:00.021442
Processing n = 100 ...
```


```
n = 100, time = 0:00:00.072784
Processing n = 1000 ...
```


```
n = 1000, time = 0:00:01.100277
Processing n = 10000 ...
```


```
n = 10000, time = 0:05:25.641090
```

Можно заметить, что скорость итераций падает с увеличением n. Также заметно, что основную часть времени занимает именно прямой ход метода Гаусса.

- **Метод LU-разложения:**

  Первый прогрессбар — LU-разложение, второй — решение относительно y, третий — решение относительно x

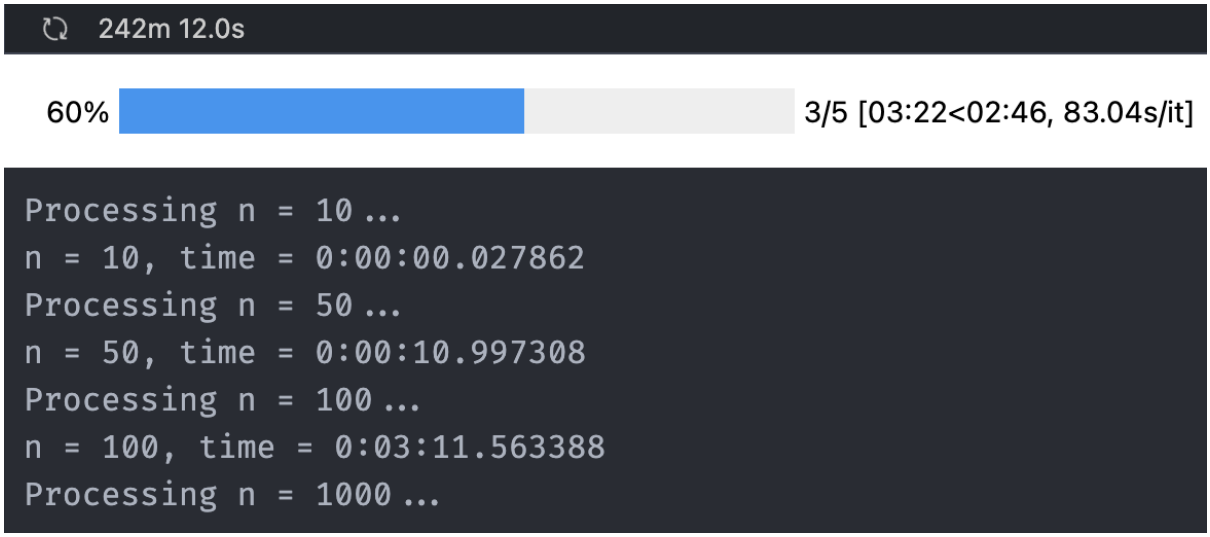
```
Processing n = 10 ...
100% 10/10 [00:00<00:00, 832.42it/s]
100% 10/10 [00:00<00:00, 1040.05it/s]
100% 10/10 [00:00<00:00, 1131.58it/s]
n = 10, time = 0:00:00.033025
Processing n = 50 ...
100% 50/50 [00:00<00:00, 2301.70it/s]
100% 50/50 [00:00<00:00, 5050.58it/s]
100% 50/50 [00:00<00:00, 5070.97it/s]
n = 50, time = 0:00:00.045181
Processing n = 100 ...
100% 100/100 [00:00<00:00, 668.81it/s]
100% 100/100 [00:00<00:00, 3733.71it/s]
100% 100/100 [00:00<00:00, 3289.47it/s]
n = 100, time = 0:00:00.255069
Processing n = 1000 ...
100% 1000/1000 [01:08<00:00, 9.85it/s]
100% 1000/1000 [00:00<00:00, 11237.70it/s]
100% 1000/1000 [00:00<00:00, 11070.53it/s]
n = 1000, time = 0:01:08.702649
```

В случае с $n = 10000$ ожидаемое время решения было более 4 часов, поэтому завершения мы не дождались 😢.

```
Processing n = 10000 ...
10% 984/10000 [19:03<4:45:20, 1.90s/it]
```

Самым затратным процессом в этом методе является LU-разложение.

- **Метод Зейделя:**



К сожалению, матрицы с 1000 и более строками методу Зейделя не поддались даже за 4 часа, поэтому вычисления было решено остановить.

Итоговые результаты можно представить в виде таблицы:

n	Метод Гаусса	LU-разложение	Метод Зейделя
10	19 мс	33 мс	27 мс
50	21 мс	45 мс	11 с
100	73 мс	255 мс	3 мин 11 с
1000	1.1 с	1 мин 8 с	>4 ч
10000	5 мин 25 с	4 ч 45 мин	∞?

Выводы:

- Самым эффективным оказался метод Гаусса. Он способен достаточно быстро справиться даже с большими матрицами.
- Метод LU-разложения проигрывает методу Гаусса на каждом этапе, но всё ещё способен обработать все матрицы за обозримое время. Самой долгой по выполнению частью метода является непосредственно получение L и U матриц, поэтому если ускорить именно эту часть метода (например, с помощью `scipy.linalg.lu`), он может обогнать метод Гаусса по производительности.
- Метод Зейделя оказался эффективнее LU-разложения для $n = 10$, но заметно проиграл другим методам на следующих этапах. Но стоит отметить, что асимптотическая сложность этого метода ниже, чем метода LU-разложения, поэтому на “идеальном” железе метод Зейделя мог бы сработать быстрее.

Выводы

Были реализованы прямые и итерационные методы решения систем линейных алгебраических уравнений. Самым эффективным по времени выполнения оказался метод Гаусса. Самым точным методом для расчётов с матрицами Гильберта оказался метод Зейделя. Методы Гаусса и LU-разложения имеют высокую точность для матриц с диагональным преобладанием, но точность всех методов падает с ростом числа обусловленности матрицы.