Национальный исследовательский университет ИТМО

Факультет информационных технологий и программирования

Прикладная математика и информатика

Отчет по лабораторной работе №3

Работу выполнили:

Гранкин Максим, группа M3237
Панов Иван, группа M3239
Назаров Георгий

Преподаватель: Корсун М.М.

Санкт-Петербург
2021 г.

1. Введение

1.1. Постановка задачи

- Реализовать прямой метод решения СЛАУ на основе LU-разложения, матрица хранится в профильном формате, провести его исследование.
- Реализовать метод Гаусса с выбором ведущего элемента для плотных матриц.
- Сравнение метода Гаусса и метода на основе LU-разложения по точности и по числу ресурсоемких операций.
- Реализовать метод сопряженных градиентов для решения СЛАУ, матрица которых хранится в разреженном строчно-столбцовом формате и является симметричной.
- Провести исследование и сделать выводы о реализованных методах.

2. Теоретическое обоснование

2.1. Форматы хранения матриц

2.1.1. Плотный формат

В этом формате хранятся все элементы матрицы A в двумерном вещественном массиве. Затраты памяти — $\mathcal{O}(n^2)$, где n — размерность квадратной матрицы A.

2.1.2. Профильный формат

Используется, если матрица обладает диагональным преобладанием, то есть, в основном ненулевые элементы сосредоточенны около главной диагонали.

Структура хранения:

- di — массив, хранящий элементы главной диагонали. Размер: n.
- al, au — массивы хранящие внутрипрофильные элементы нижнего и верхнего треугольников соответственно. Каждый размера $\sum_{i=1}^{n} p_i$, где p_i — размер профиля i-ой строки/столбца.
- ia — массив указателей (или индексов) на начальный элемент строки/столбца в массивах al, au. Размер: n.

Итого, затраты памяти: $\mathcal{O}(n+P)$, где P — сумма профилей каждой строки/столбца.

2.2. Системы линейных алгебраических уравнений

Система линейных алгебраических уравнений (СЛАУ) — система уравнений, каждое уравнение в которой является линейным алгебраическим уравнением первой степени.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases}$$

Может быть записано в матричном виде:

$$Ax = b; \quad A = (a_{ij})_{i,j=1}^{nn}; \quad b = (b_1, \ldots, b_n)^T; \quad x = (x_1, \ldots, x_n)^T$$

Решить СЛАУ — найти вектор x, удовлетворяющий заданным A и b. Решение существует, когда существует матрица A^{-1}, то есть матрица A — невырожденна.

2.3. Метод Гаусса

Метод Гаусса — классический метод решения СЛАУ — метод последовательного исключения переменных, когда с помощью элементарных преобразований система уравнений приводится к равносильной системе треугольного вида, из которой последовательно, начиная с последних (по номеру), находятся все переменные системы.

2.3.1. Прямой ход

Здесь и далее k — номер итерации, от 1 до $n-1$.

$$A_{i,j}^{(0)} = A_{i,j}; \quad b_i^{(0)} = b_i$$

На каждой итерации $\forall i, j \in [k+1..n]$:

$$A_{i,j}^{(k)} = A_{i,j}^{(k-1)} - \frac{A_{i,k}^{(k-1)}}{A_{k,k}^{(k-1)}} A_{k,j}^{(k-1)};$$

$$b_i^{(k)} = b_i^{(k-1)} - \frac{A_{i,k}^{(k-1)}}{A_{k,k}^{(k-1)}} b_k^{(k-1)}$$

$A_{k,k}^{(k-1)}$ называется главным элементом.

2.3.2. *Обратный ход*

$$x_n = \frac{b_n^{(n-1)}}{A_{n,n}^{(n-1)}}$$

$$\forall k \in [n-1..1] \quad x_k = \frac{1}{A_{k,k}^{(n-1)}} \left[b_k^{(n-1)} - \sum_{j=k+1}^{n} A_{k,j}^{(n-1)} x_j \right]$$

2.4. *LU*-разложение

***LU*-разложение** - это представление матрицы A в виде $A = L \cdot U$, где L — нижнетреугольная матрица с единицами на главной диагонали, а U — верхнетреугольная матрица. LU-разложение возможно, когда матрица A невырожденна.

2.4.1. *Алгоритм LU-разложения*

$$\forall i \in 1..n \quad L_{i,i} = 1 \quad and:$$

$$\forall j \in i..n \quad U_{i,j} = A_{i,j} - \sum_{k=1}^{i-1} L_{i,k} U_{k,j}$$

$$\forall j \in i+1..n \quad L_{j,i} = \frac{1}{U_{j,j}} \cdot \left[A_{j,i} - \sum_{k=1}^{i-1} L_{j,k} U_{k,i} \right]$$

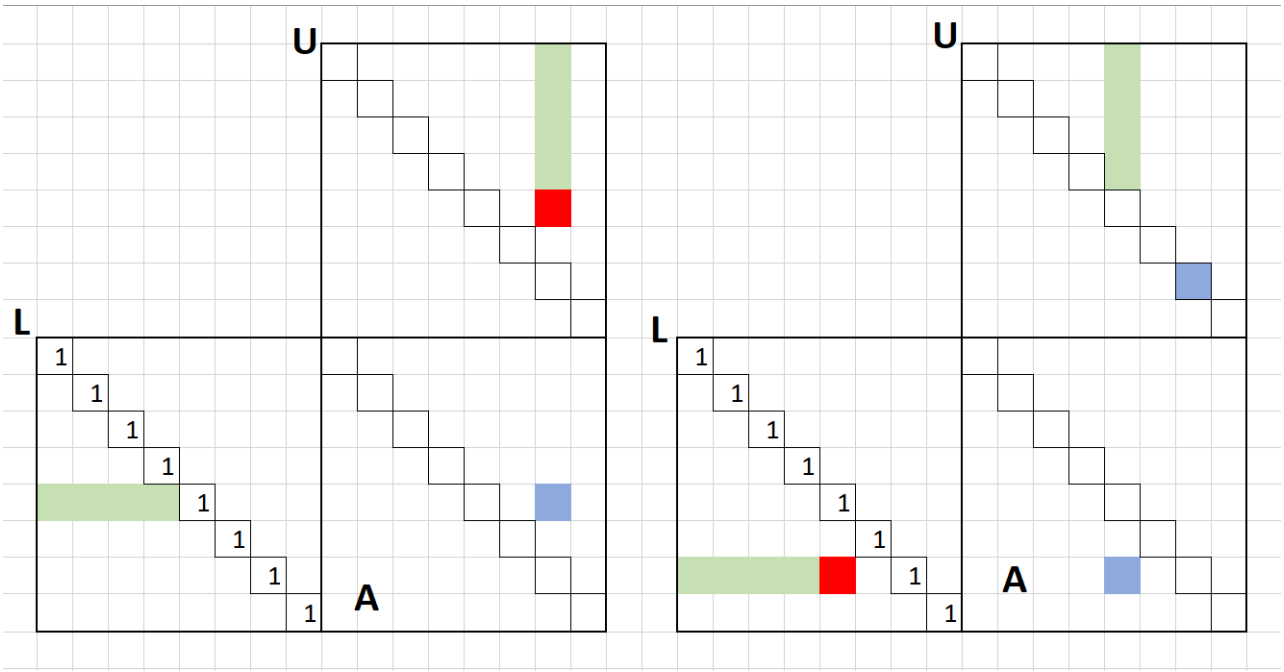


Иллюстрация формул: слева — вычисление значения матрицы U, справа — матрицы L. Красным выделен вычисляемый элемент, зеленым — элементы суммы произведений.

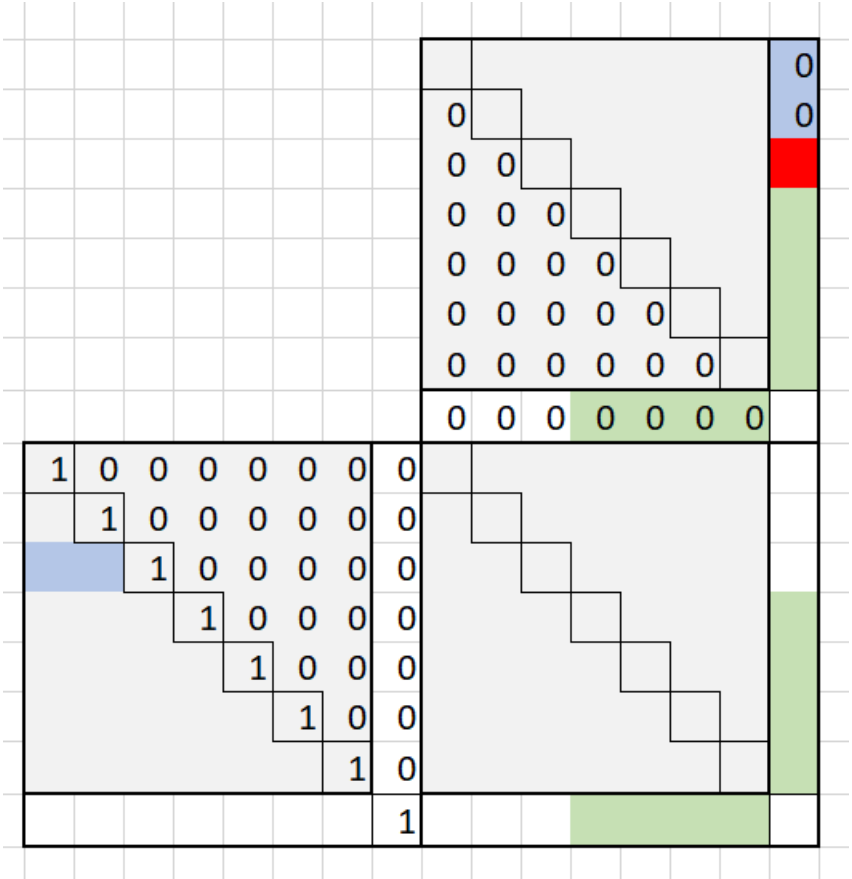
2.4.2. Сохранение профиля при LU-разложении

Утверждение. Пусть матрица A имеет профиль $p_1, \dots p_n$. Тогда, если LU-разложение существует, то матрицы L и U имеют такой же профиль.

Доказательство. Будем строить доказательство индукцией по n — размерности матрицы A. Рассмотрим матрицу $A_1 = \{a_{1,1}\}$. LU-разложение для неё тривиально, $L = \{1\}$, $U = \{a_{1,1}\}$, Профили матриц A, L и U совпадают.

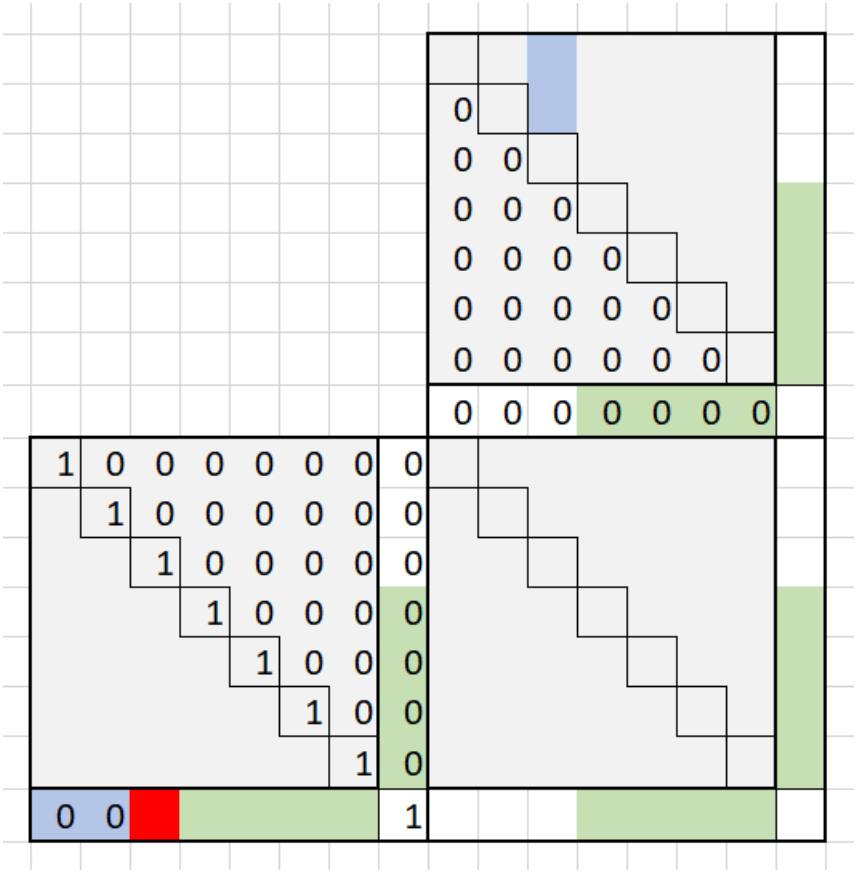
Теперь предположим, что для всех $n' < n$ утверждение верно. Рассмотрим матрицу A размерности $n \times n$. Для подматрицы A', состоящей из первых $n-1$ строк и $n-1$ столбцов матрицы A утверждение верно, так как в формулах LU-разложения (см. предыдущий пункт) всегда выполнено $k < i \leq j$, то есть каждый элемент матриц L и U зависит только от элементов с меньшими или равными обоими индексами.

Рассмотрим последний столбец матрицы U. Если профиль этого столбца $p_n = n$, то утверждение верно, так как нам не важны значения элементов, входящих в профиль. Если $p_n < n$: $U_{1,n} = A_{1,n} = 0$, так как элемент $A_{1,n}$ находится вне профиля. Применим индукцию по i вдоль этой строки. Пусть $\forall i' < i : L_{n,i} = 0$. Тогда, сумма, участвующая в вычислении $L_{n,i}$ равна 0, так как произведения внутри суммы обращаются в 0. Таким образом, все элементы $L_{n,i}$, не входящие в профиль исходной матрицы A равны 0. (См. иллюстрацию: Красным — вычисляемый элемент, зеленым — профиль, синим — элементы, участвующие в сумме произведений).



Проведем аналогичные рассуждения для матрицы L. Рассмотрим послед-

нюю строку матрицы L. Если профиль этой строки $p_n = n$, то утверждение верно, так как нам не важны значения элементов, входящих в профиль. Если $p_n < n$: $L_{n,1} = \frac{A_{n,1}}{U_{j,j}} = 0$, так как этот элемент находится вне профиля. Применим индукцию по j вдоль этого столбца. Пусть $\forall j' < j : U_{j,n} = 0$. Тогда, сумма, участвующая в вычислении $U_{j,n}$ равна 0, так как произведения внутри суммы обращаются в 0. Таким образом, все элементы $U_{j,n}$, не входящие в профиль исходной матрицы A равны 0. (См. иллюстрацию: Красным — вычисляемый элемент, зеленым — профиль, синим — элементы, участвующие в сумме произведений).



Доказанное утверждение, совместно с фактом, что вычисления выполняются в порядке увеличения индексов, дают возможность реализовать LU-разложение «in-place», без выделения дополнительной памяти.

2.5. Решение СЛАУ методом LU-разложения

Рассмотрим СЛАУ $Ax = b$. Если A — невырожденна, то существует решение этого СЛАУ и, одновремменно с этим, существует её LU-разложение. Тогда, положим:

$$A = LU; \quad LUx = b$$

Пусть $y = Ux$, тогда $Ly = b$. Исходя из этого, получаем алгоритм решения СЛАУ:

1. По A найти L и U.
2. Решить $Ly = b$ прямым ходом метода Гаусса. Обратный ход не нужен, потому что L — нижнетреугольная матрица.
3. Решить $Ux = y$ обратным ходом метода Гаусса. Прямой ход не нужен, потому что U — верхнетреугольная матрица.

2.6. Решение СЛАУ методом сопряженных градиентов

Новый пункт

Рассмотрим СЛАУ $Ax = b$ с симметричной положительно определенной матрицей A. Тогда, процесс решения системы можно трактовать как минимизацию следующей квадратичной функции:

$$f(x) = \frac{1}{2}(Ax, x) - (b, x)$$

Будем минимизировать эту функцию методом сопряженных градиентов. Выбираем начальное приближение x_0. Полагаем:

$$r_0 = b - Ax_0; \quad z_0 = r_0$$

Итерационная процедура такова: для всех $k = 1, 2, \ldots$:

$$\alpha_k = \frac{(r_{k-1}, r_{k-1})}{(Az^{k-1}, z^{k-1})}$$

$$x_k = x_{k-1} + \alpha_k z_{k-1}; \quad r_k = r_{k-1} - \alpha_k A z_{k-1}$$

$$\beta_k = \frac{(r_k, r_k)}{(r_{k-1}, r_{k-1})}; \quad z_k = r_k + \beta_k z_{k-1}$$

Здесь r_k — вектор невязки на k-той итерации, z_k — вектор спуска. Окончание процесса происходит при $\frac{\|r_k\|}{\|b\|} < \varepsilon$ или при достижении максимально допустимого количества итераций.

2.7. Подготовка тестовых данных

Матрица хранится в профильном формате, доступ к элементам выполняется за $O(1)$ времени. Такая скорость достигается за счет использования массивов индексов и пересчета индексов расположения элемента в конечной матрице.

Для удобства пользователя и тестирования нами был разработан генератор систем линейных алгебраических уравнений в профильном виде. На вход пользователь подает название желаемой директории и, если она существует, то в нее записывается/перезаписывается файл test.txt. Если директория не

существует, то она создается и повторяется шаг описанный выше. Поддерживаются все символы UTF-8, в том числе различные языки и эмоджи.
Формат генерации файла:

- n = Размер матрицы.
- au = Массив значений с первого ненулевого элемента сверху от главной диагонали.
- al = Массив значений с первого ненулевого элемента снизу от главной диагонали.
- r = Массив значения справа от равно в системе уравнений.
- ia = "Профиль матрицы".
- di = Массив значений на главной диагонали.

3. Исследование прямого метода решения СЛАУ с помощью LU-разложения

3.1. Зависимость от диагонального преобладания

Для проведения исследования была решена последовательность СЛАУ: $A_k \cdot x_k = f_k \; \forall k \in 1 \dots 10$, каждая матрица строилась следующим образом:

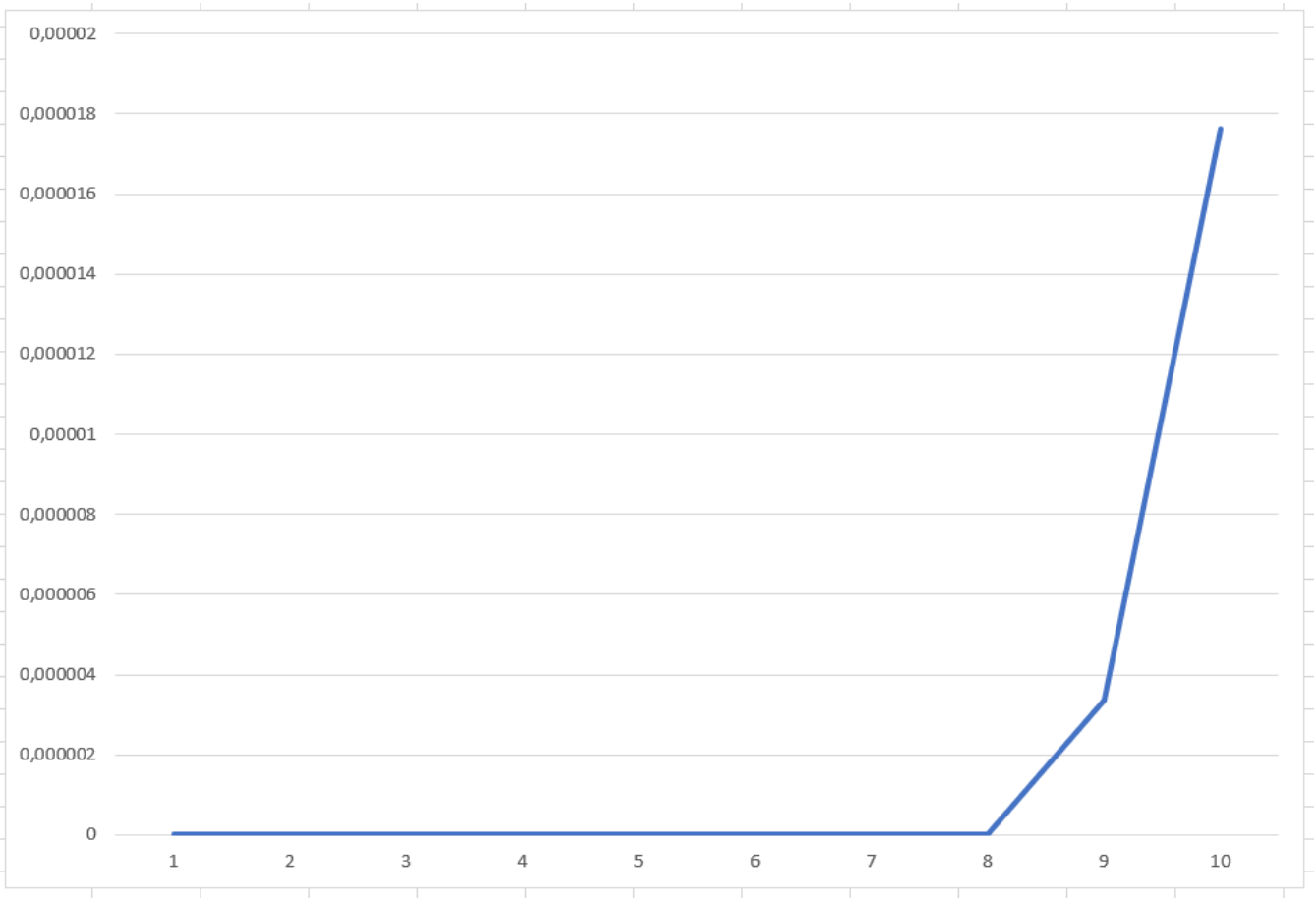
$$a_{ii} = \begin{cases} -\sum_{i \neq j} a_{ij} & i > 1 \\ -\sum_{i \neq j} a_{ij} + 10^{-k} & i = 1 \end{cases}$$

Где $a_{ij} \in \{0, -1, -2, -3, -4\}$ выбирается случайным образом, $f_k = A_k x^*$, где $x^* = (1, 2, 3, \dots, n)^T$

Приведем результаты исследований, погрешности решений из последовательности систем линейных алгебраических уравнений при различных размерностях $n = 10^q$, где $q = 1..3$

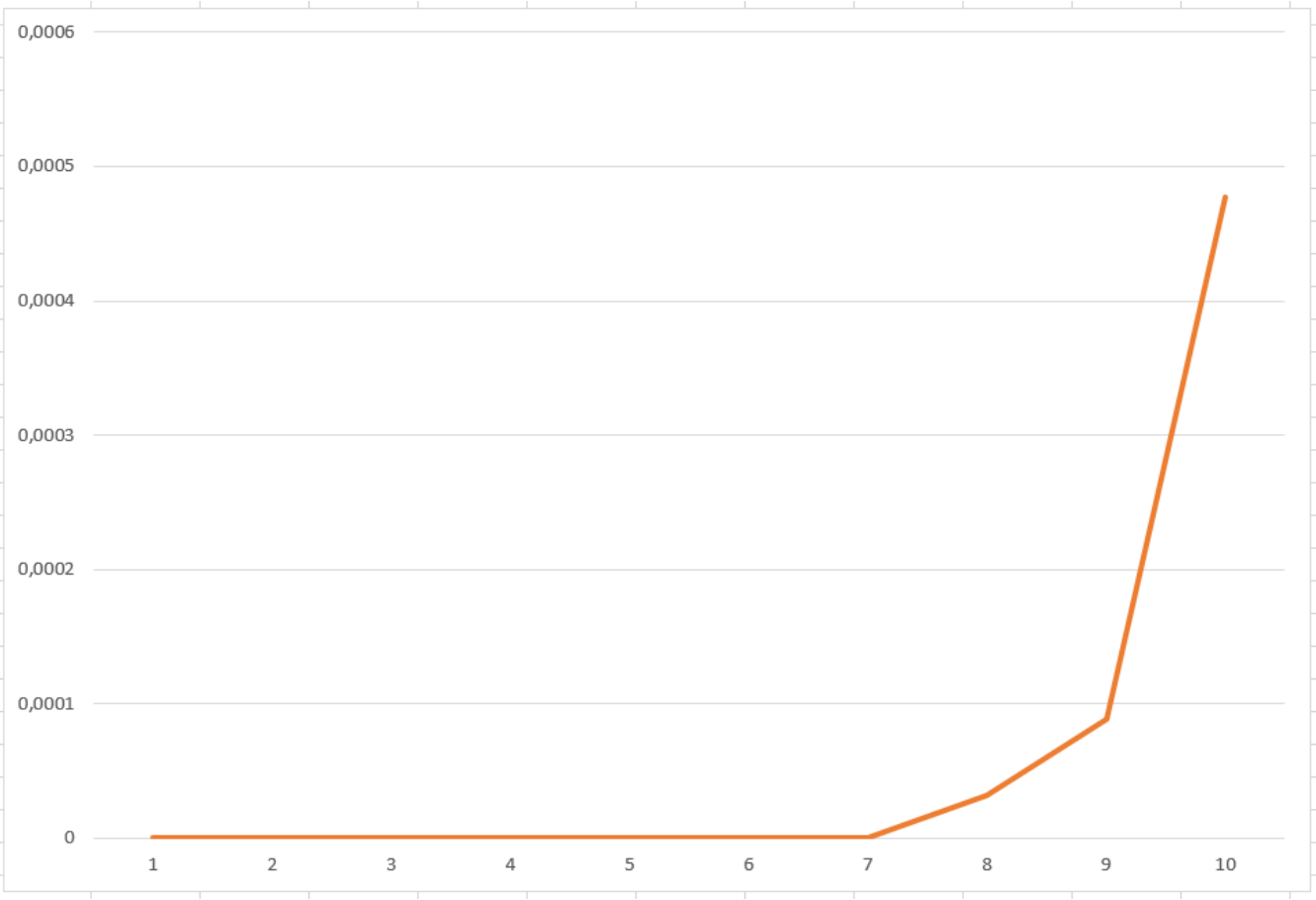
3.1.1. Результаты при $n = 10$

n	k	$\|\|x^* - x_k\|\|$	$\frac{\|\|x^* - x_k\|\|}{\|\|x^*\|\|}$
10	1	0.000000000	0.000000000
10	2	0.000000000	0.000000000
10	3	0.000000000	0.000000000
10	4	0.000000001	0.000000000
10	5	0.000000000	0.000000001
10	6	0.000000008	0.000000002
10	7	0.000000000	0.000000000
10	8	0.000008715	0.000000000
10	9	0.000056458	0.000003346
10	10	0.001066955	0.000017625



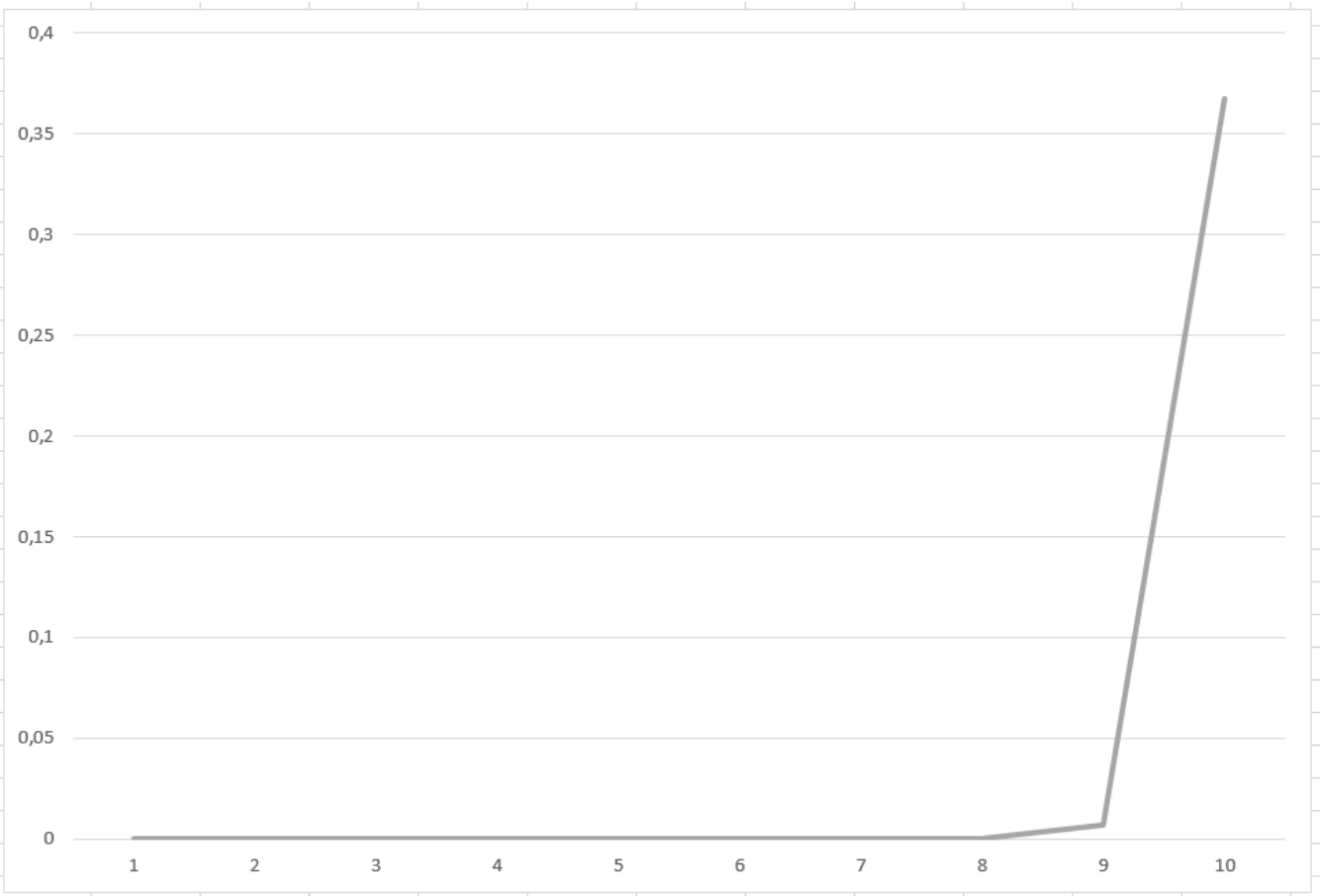
3.1.2. Результаты при $n = 100$

n	k	$\|\|x^* - x_k\|\|$	$\frac{\|\|x^* - x_k\|\|}{\|\|x^*\|\|}$
100	1	0.000000002	0.000000000
100	2	0.000000004	0.000000000
100	3	0.000000140	0.000000000
100	4	0.000000000	0.000000004
100	5	0.000002408	0.000000011
100	6	0.000143428	0.000000533
100	7	0.000357455	0.000000611
100	8	0.007832128	0.000031766
100	9	0.205230885	0.000088503
100	10	0.370370370	0.000477177



3.1.3. Результаты при $n = 1000$

n	k	$\|\|x^* - x_k\|\|$	$\frac{\|\|x^* - x_k\|\|}{\|\|x^*\|\|}$
1000	1	0.000001171	0.000000000
1000	2	0.000002353	0.000000004
1000	3	0.000279974	0.000000025
1000	4	0.009845066	0.000000030
1000	5	0.040407366	0.000001189
1000	6	0.159929111	0.000005248
1000	7	5.610367065	0.000055889
1000	8	27.517444738	0.000003939
1000	9	444.524197059	0.006715169
1000	10	1373.183286835	0.367215448



Во время работы метода ошибка вычислений может накапливаться, что ведет за собой погрешность в результатах, которая растет с ростом n и экспоненциально с ростом k.

Рост k влечет за собой появление операций сложения, в которых участвуют числа, близкие к нулю, что способствует возникновению ошибки из-за недостаточной точности ЭВМ.

3.2. Матрицы Гильберта различной размерности

Было проведено исследование решения СЛАУ через LU-разложение на матрицах Гильберта.

Элементы матрицы Гильберта A размерности n задается следующим образом:

$$a_{ij} = \frac{1}{i+j-1} \quad i \in 1 \dots n; \quad j \in 1 \dots n$$

3.2.1. Результаты исследования при различных n

n	$\|\|x^* - x_k\|\|$	$\frac{\|\|x^* - x_k\|\|}{\|\|x^*\|\|}$
1	0.000000000	0.000000000
3	0.000000000	0.000000000
5	0.000000000	0.000000000
7	0.000000014	0.000000001
9	0.000055981	0.000003316
11	0.306896841	0.013643229
13	356.932343016	12.472225431
15	52.985671718	1.504691965
17	228.509685701	5.408607957
19	1135.446214524	22.846416511
21	636.953303346	11.069499982
23	2159.073167480	32.834054708
25	51816.982071975	697.117482136
27	3413.138662361	41.000326484
29	10318.421077298	111.558647346

Даже на маленьких размерностях наблюдается большая погрешность, которая растет с ростом n. Это связано с тем, что матрицы Гильберта плохо обусловлены и в процессе решения на главной диагонали появляются близкие к нулю числа.

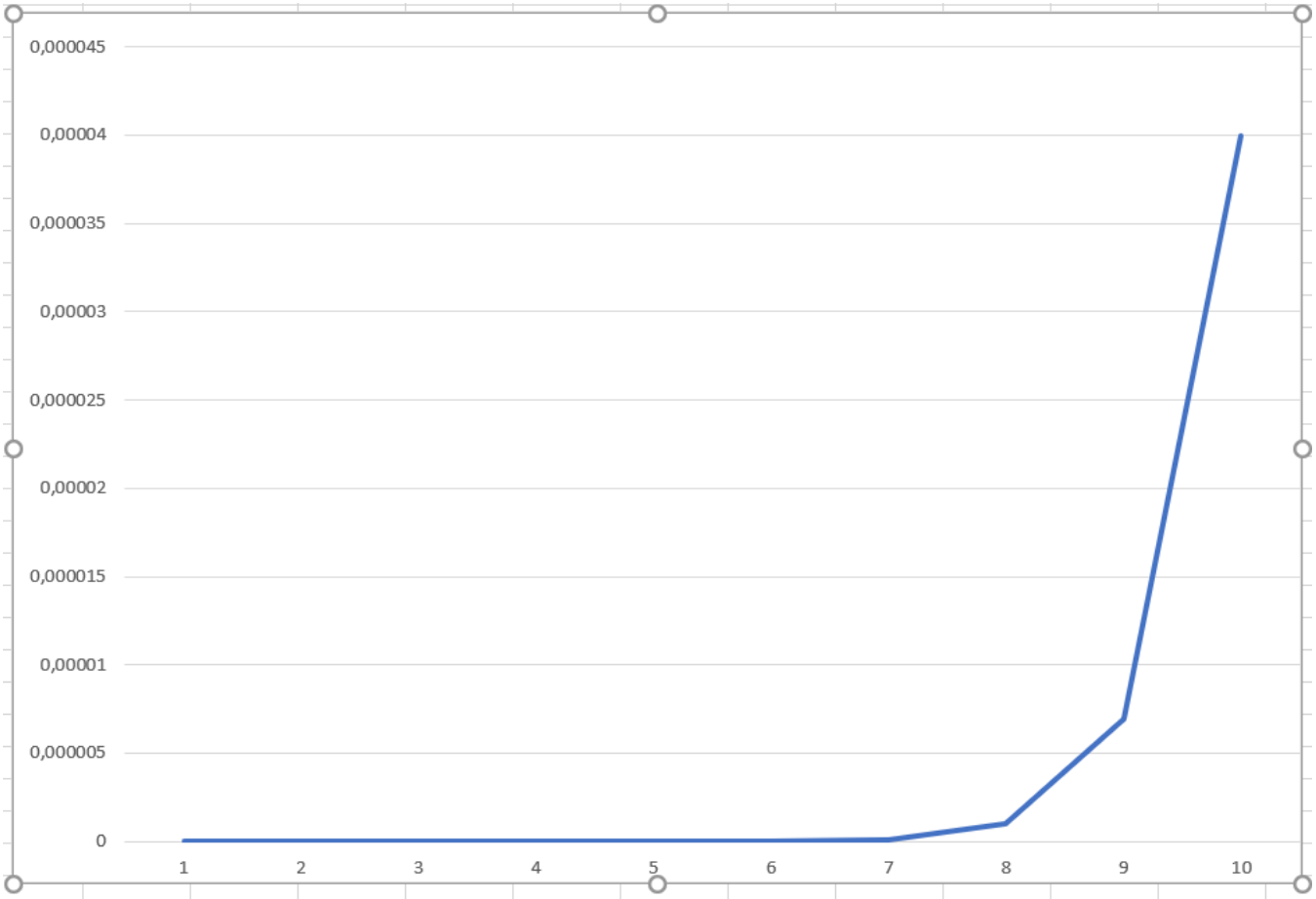
4. Исследование метода Гаусса и сравнение с LU-разложением

Было проведено исследование на матрицах Гильберта и на матрицах A_k из прошлого пункта. Матрицы элементов A_k генерировалась аналогичных способом. Ниже приведены таблицы результатов исследования.

4.0.1. Результаты исследования на матрицах A_k

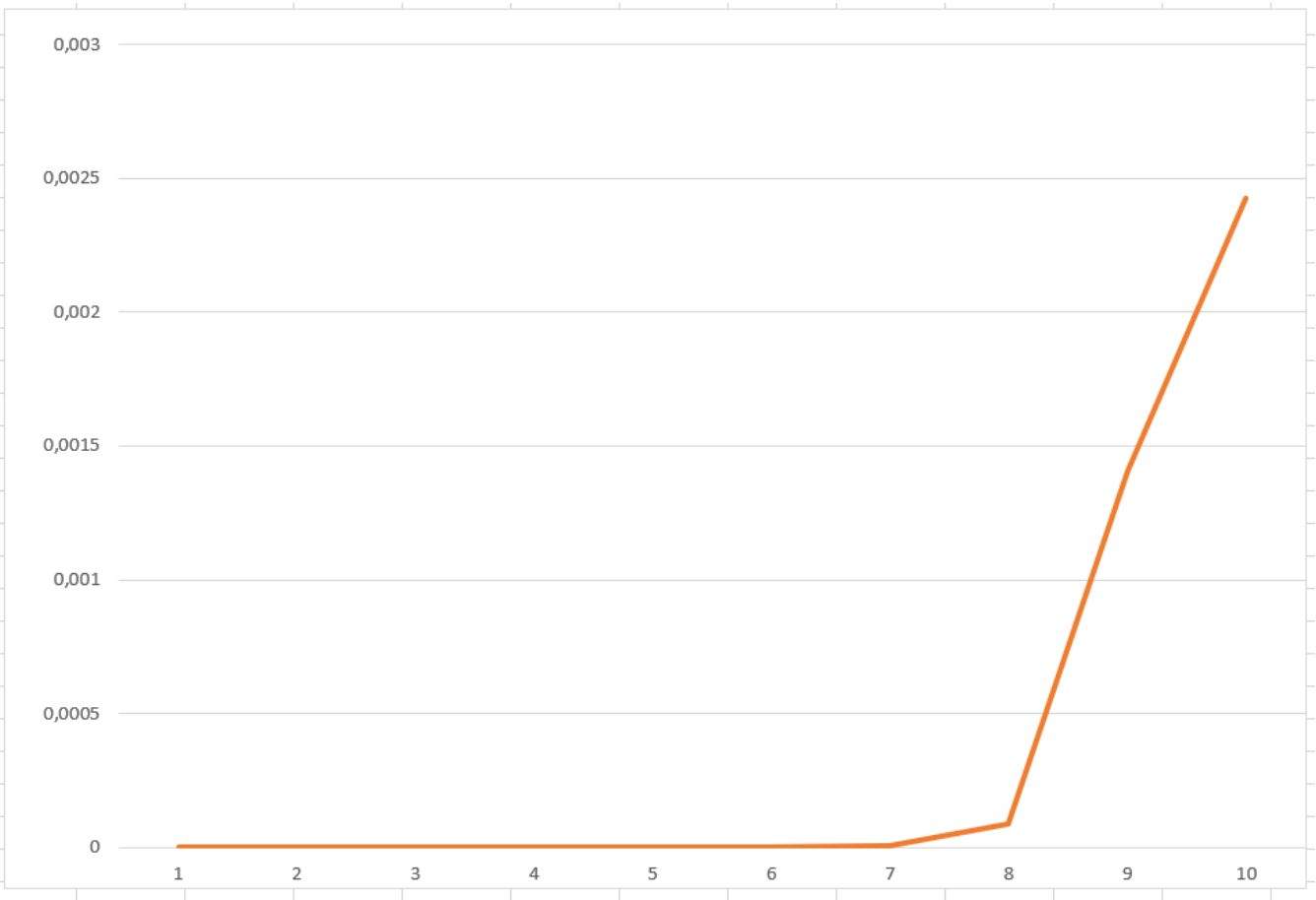
Результаты при $n = 10$

n	k	$\|\|x^* - x_k\|\|$	$\frac{\|\|x^* - x_k\|\|}{\|\|x^*\|\|}$
10	1	0.000000000000218	0.000000000000055
10	2	0.000000000034335	0.000000000001409
10	3	0.000000000004251	0.000000000001808
10	4	0.000000000295553	0.000000000027005
10	5	0.000000005142660	0.000000002501850
10	6	0.000000034767850	0.000000002428999
10	7	0.000000065727307	0.000000062516676
10	8	0.000047893217257	0.000001026270114
10	9	0.000087272080635	0.000006971204379
10	10	0.000322934029360	0.000039981293179



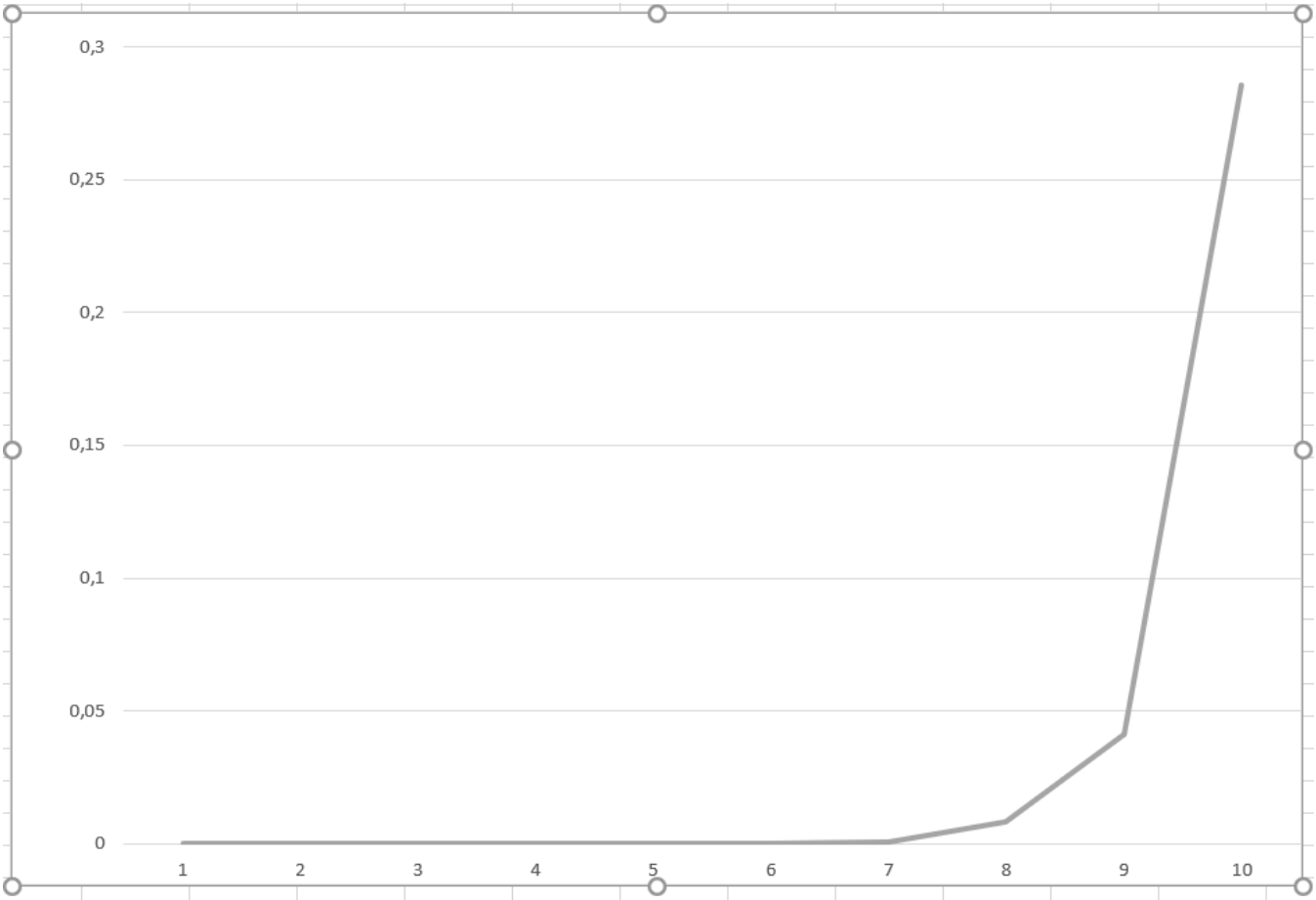
Результаты при $n = 100$

n	k	$\|\|x^* - x_k\|\|$	$\frac{\|\|x^* - x_k\|\|}{\|\|x^*\|\|}$
100	1	0.000000003105681	0.000000000010273
100	2	0.000000000122992	0.000000000197957
100	3	0.000000159235234	0.000000000525145
100	4	0.000000186904712	0.000000009041253
100	5	0.000034024482351	0.000000280029550
100	6	0.000625940994445	0.000000542018571
100	7	0.001131210388208	0.000005275183764
100	8	0.002965495876525	0.000087609160841
100	9	0.358552067249603	0.001405429480429
100	10	0.699956069702699	0.002425911225498



Результаты при $n = 1000$

n	k	$\|\|x^* - x_k\|\|$	$\frac{\|\|x^* - x_k\|\|}{\|\|x^*\|\|}$
1000	1	0.000000655624788	0.000000000518806
1000	2	0.000004276317780	0.000000007636586
1000	3	0.000016637398144	0.000000017996114
1000	4	0.003887528155016	0.000000186698426
1000	5	0.112192121330675	0.000018808585472
1000	6	1.297136406090628	0.000164895958873
1000	7	16.082114049704340	0.000434014764445
1000	8	145.064905856394920	0.008129393750657
1000	9	696.688716865127000	0.041409723819976
1000	10	1196.854364645385700	0.285599159295805



4.0.2. Результаты исследования на матрицах Гильберта

Результаты исследования при различных n

n	$\|\|x^* - x_k\|\|$	$\frac{\|\|x^* - x_k\|\|}{\|\|x^*\|\|}$
1	0.000000000000000	0.000000000000000
3	0.000000000000015	0.000000000000004
5	0.000000000009103	0.000000000001227
7	0.000000005693990	0.000000000481230
9	0.000083310163586	0.000004934868191
11	0.024259130735842	0.001078449905069
13	67.542414520338840	2.360122965908584
15	222.567855487526030	6.320502372356547
17	50.979315975175780	1.206632152956001
19	7610.265749313176000	153.126849026488690
21	239.633078397170400	4.164541329931263
23	358.793235367204550	5.456339737117030
25	10154.595058573514000	136.614396599035870
27	1855.940056559511000	22.294479006290373
29	1961.260877851453200	21.204369252430350

Исследование на матрицах Гильберта показало, что при выборе строки с максимальным по модулю ведущим элементом, начиная с какого-то момента, метод Гаусса не будет завершать свою работу. Исследователи связывают это с присутствием в матрицах Гильберта чисел, близких к нулю.

При всех исследованных размерностях для матриц A_k почти всегда погрешность оказывалась выше. Но, на больших размерностях метод Гаусса показывает себя быстрее чем LU разложение.

Однако, из-за большего числа арифметических операций, метод Гаусса показывает несколько меньшую точность, хоть относительные ошибки примерно одинаковы.

4.1. Алгоритмическая сложность методов

Обозначим как g — стоимость операции чтения по индексу элемента из матрицы, s — стоимость операции записи по индексу элемента в матрицу.

Тогда, вычислительная сложность LU-разложения (получено методом чтения исходного кода):

$$\sum_{i=0}^{n-1}\left[\sum_{j=i}^{n-1}\left[\sum_{k=0}^{i-1}(3g+s)\right]+\sum_{j=i+1}^{n-1}\left[\sum_{k=0}^{i-1}(4g+s)\right]\right]=$$

$$\sum_{i=0}^{n-1}\left[\sum_{j=i+1}^{n-}\left[\sum_{k=0}^{i-1}(3g+s)\right]+\sum_{k=0}^{i-1}(3g+s)+\sum_{j=i+1}^{n-1}\left[\sum_{k=0}^{i-1}(4g+s)\right]\right]=$$

$$\sum_{i=0}^{n-1}\left[\sum_{j=i+1}^{n-1}\left[\sum_{k=0}^{i-1}(7g+2s)\right]+\sum_{k=0}^{i-1}(3g+s)\right]=$$

$$\sum_{i=0}^{n-1}\left[\sum_{j=i+1}^{n-1}\sum_{k=0}^{i-1}7g+\sum_{j=i+1}^{n-1}\sum_{k=0}^{i-1}2s+\sum_{k=0}^{i-1}3g+\sum_{k=0}^{i-1}s\right]=$$

$$\sum_{i=0}^{n-1}\left[\sum_{j=i+1}^{n-1}7gi+\sum_{j=i+1}^{n-1}2si+(3g+s)i\right]=$$

$$\sum_{i=0}^{n-1}\left[(7g+2s)i(n-i-1)+(3g+s)i\right]=$$

$$(7g+2s)\sum_{i=0}^{n-1}i(n-i-1)+\frac{(3g+s)(n-1)n}{2}=$$

$$\frac{(7g+2s)n(n-1)(n-2)}{3}+\frac{(3g+s)(n-1)n}{2}=$$

$$\frac{7gn^3}{3}-\frac{11gn^2}{2}+\frac{19gn}{6}+\frac{2sn^3}{3}-\frac{3sn^2}{2}+\frac{5sn}{6}$$

Полученная асимптотика: $\mathcal{O}((g+s)n^3)$. Если доступ на чтение/запись в матрицу осуществляется за $\mathcal{O}(1)$, то, формулу можно упростить до

$$3n^3-14n^2+4n+\mathcal{O}(1)$$

Алгоритмическая сложность метода Гаусса (получено методом чтения исходного кода):

$$\sum_{i=0}^{n-1}\left[\sum_{j=i}^{n-1}(2g)+g+n-i+\sum_{j=i+1}^{n-1}\left[2g+\sum_{k=i}^{n-1}(s+2g)+s+2g\right]\right]+\sum_{i=0}^{n-1}\left[2g+\sum_{j=i+1}^{n-1}(g)\right]=$$

$$\sum_{i=0}^{n-1}\left[\sum_{j=i}^{n-1}(2g)-i+(n-i-1)(4g+s)\sum_{j=i+1}^{n-1}(s+2g)(n-i)\right]+$$

$$+\sum_{i=0}^{n-1}\big[2g+(n-i-1)g\big]+n^2+ng=$$

$$\sum_{i=0}^{n-1}\Big[2g(n-i)-i\Big]+(4g+s)\sum_{i=0}^{n-1}(n-i-1)+(s+2g)\sum_{i=0}^{n-1}(n-i-1)(n-i)+2gn+$$

$$+2gn+g\sum_{i=0}^{n-1}(n-i-1)+n^2+gn=$$

$$2g\sum_{i=0}^{n-1}(n-i)-\frac{(n-1)n}{2}+(4g+s)\frac{(n-1)n}{2}+(s+2g)\frac{n(n^2-1)}{3}+3gn+g\frac{(n-1)n}{2}+n^2=$$

$$2g\frac{(n+1)n}{2}+(s+2g)\frac{n(n^2-1)}{3}+3gn+(5g+s-1)\frac{(n-1)n}{2}+n^2=$$

$$\frac{2gn^3}{3}+gn^2+\frac{gn}{3}+\frac{sn^3}{3}-\frac{sn}{3}+3$$

Полученная асимптотика: $\mathcal{O}((g+s)n^3)$. Если доступ на чтение/запись в матрицу осуществляется за $\mathcal{O}(1)$, то, формулу можно упростить до

$$n^3+n^2+\mathcal{O}(1)$$

Итого, алгоритмические сложности LU и метода Гаусса совпадают, но метод Гаусса работает быстрее из-за того, что множитель перед n^3 меньше.

5. Метод сопряженных градиентов

Новый пункт

Был реализован метод сопряженных градиентов для решения СЛАУ с симметричной матрицей A. Базовая функциональность была протестирована на следующих примерах:

$$A = \begin{pmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{pmatrix} \qquad x^* = \begin{pmatrix} 1.0 \\ 1.0 \end{pmatrix} \qquad b = \begin{pmatrix} 1.0 \\ 1.0 \end{pmatrix} \qquad x \begin{pmatrix} 1.0 \\ 1.0 \end{pmatrix}$$

$$A = \begin{pmatrix} 5.0 & -2.0 \\ -2.0 & 1.0 \end{pmatrix} \qquad x^* = \begin{pmatrix} 0.0 \\ 3.0 \end{pmatrix} \qquad b = \begin{pmatrix} -6.0 \\ 3.0 \end{pmatrix} \qquad x = \begin{pmatrix} 4.8 \cdot 10^{-15} \\ 3.0 \end{pmatrix}$$

$$A = \begin{pmatrix} 5.0 & -2.0 & 14.0 \\ -2.0 & 1.0 & 7.0 \\ 14.0 & 7.0 & -1.0 \end{pmatrix} \qquad x^* = \begin{pmatrix} 19.0 \\ 84.0 \\ 115.0 \end{pmatrix} \qquad b = \begin{pmatrix} 1537.0 \\ 851.0 \\ 739.0 \end{pmatrix}$$

$$x = \begin{pmatrix} 18.999999999999943 \\ 84.0 \\ 114.99999999999986 \end{pmatrix}$$

$$A = \begin{pmatrix} 18.0 & -20.0 & 148.0 & 1337.0 \\ -20.0 & 1548.0 & 77.0 & 15.0 \\ 148.0 & 77.0 & 8412.0 & -1.0 \\ 1337.0 & 15.0 & -1.0 & 844.0 \end{pmatrix} \qquad x^* = \begin{pmatrix} -87.0 \\ -999.0 \\ 265.0 \\ 148.0 \end{pmatrix} \qquad b = \begin{pmatrix} 255510.0 \\ -1522087.0 \\ 2139233.0 \\ -6657.0 \end{pmatrix}$$

$$x = \begin{pmatrix} -87.00000000000001 \\ -999.0 \\ 264.99999999999744 \\ 147.99999999999994 \end{pmatrix}$$

Вектора b вычислялись умножением A на x^*. Использовалась точность $\varepsilon = 10^{-12}$.

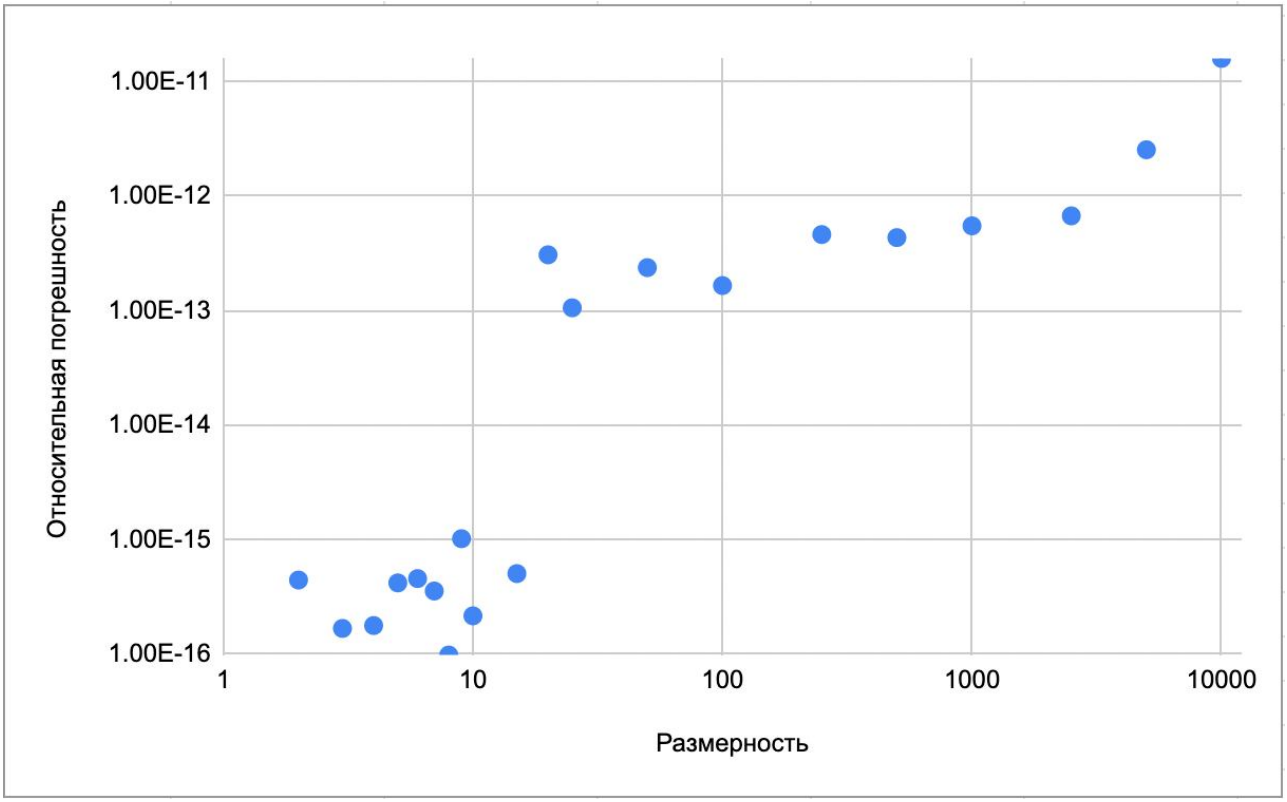
5.1. Матрицы с диагональным преобладанием

Было проведено исследование метода сопряженных градиентов на матрицах, число обусловленности которых регулировалось за счет изменения диагонального преобладания. Матрицы строились по следующей формуле:

$$a_{i,i} = \begin{cases} -\sum_{j \neq i} a_{i,j} & i > 1 \\ -\sum_{j \neq i} a_{i,j} + 1 & i = 1 \end{cases}$$

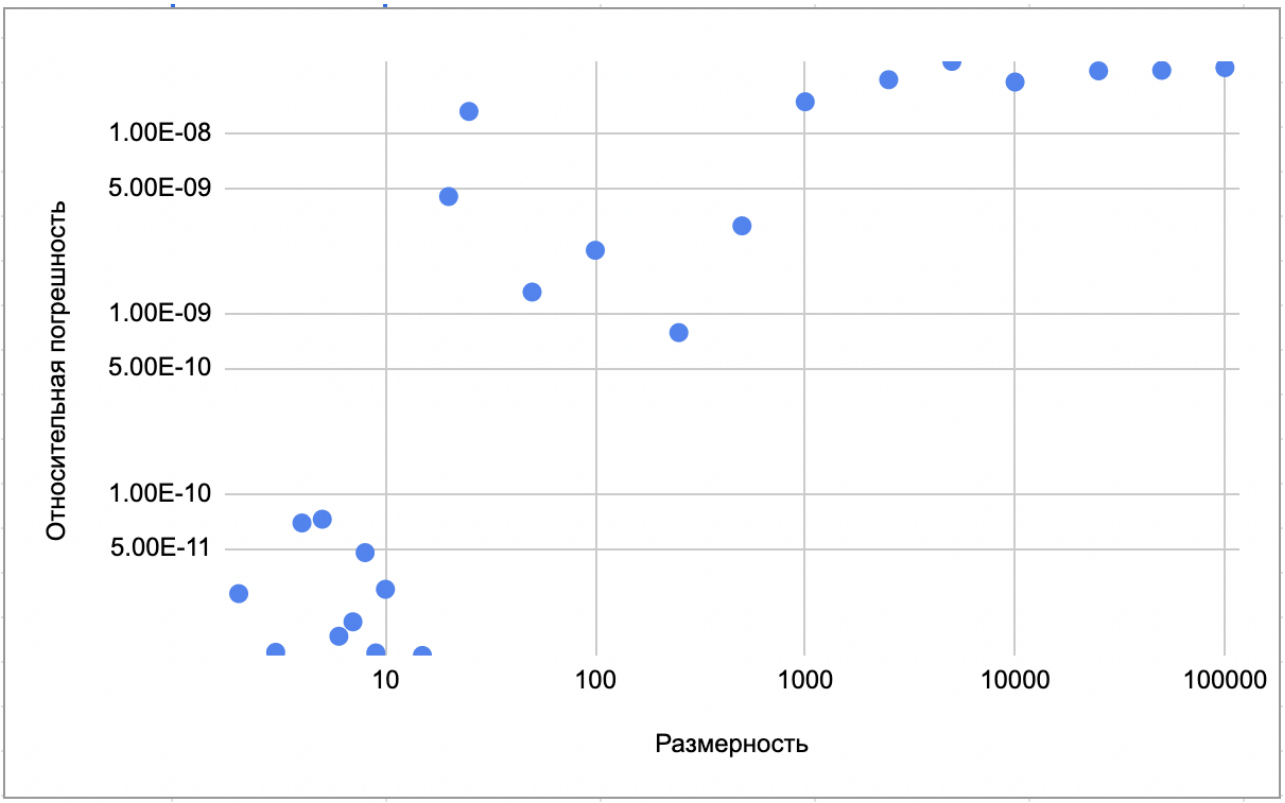
Где $a_{i,j} \in \{0, 1, 2, 3, 4\}$ выбирались случайным образом. Вектор правой части СЛАУ получали умножением матрицы A на вектор ответа $x^* = (1, 2, \ldots, n)$.

Размерность	Итерации	$\|\|x^* - x_k\|\|$	$\frac{\|\|x^* - x_k\|\|}{\|\|x^*\|\|}$	$cond(A) \geq$
2	3	9.93014e-16	4.44089e-16	0.412311
3	4	6.28037e-16	1.67850e-16	0.769972
4	5	9.74217e-16	1.77867e-16	0.728554
5	6	3.10862e-15	4.19167e-16	1.11993
6	7	4.36248e-15	4.57313e-16	0.724794
7	8	4.21300e-15	3.56064e-16	2.48694
8	9	1.40433e-15	9.83230e-17	0.585046
9	10	1.72052e-14	1.01915e-15	3.94909
10	11	4.24216e-15	2.16200e-16	0.566137
15	16	1.77982e-14	5.05436e-16	0.797809
20	18	1.64425e-11	3.06921e-13	0.576511
25	19	7.86205e-12	1.05772e-13	0.476377
50	18	4.91523e-11	2.37240e-13	0.510683
100	17	9.62704e-11	1.65504e-13	0.508835
250	15	1.05608e-09	4.61366e-13	0.507746
500	14	2.80452e-09	4.33824e-13	0.510542
1000	13	1.00424e-08	5.49632e-13	0.587581
2500	12	4.85135e-08	6.72022e-13	0.980875
5000	12	5.17865e-07	2.53663e-12	32.5045
10000	11	9.19106e-06	1.59182e-11	32.6590



Так же, было проведено исследование на аналогичных матрицах, но, все элементы, для которых не выполнено $|i - j| \leq 10$ были обнулены.

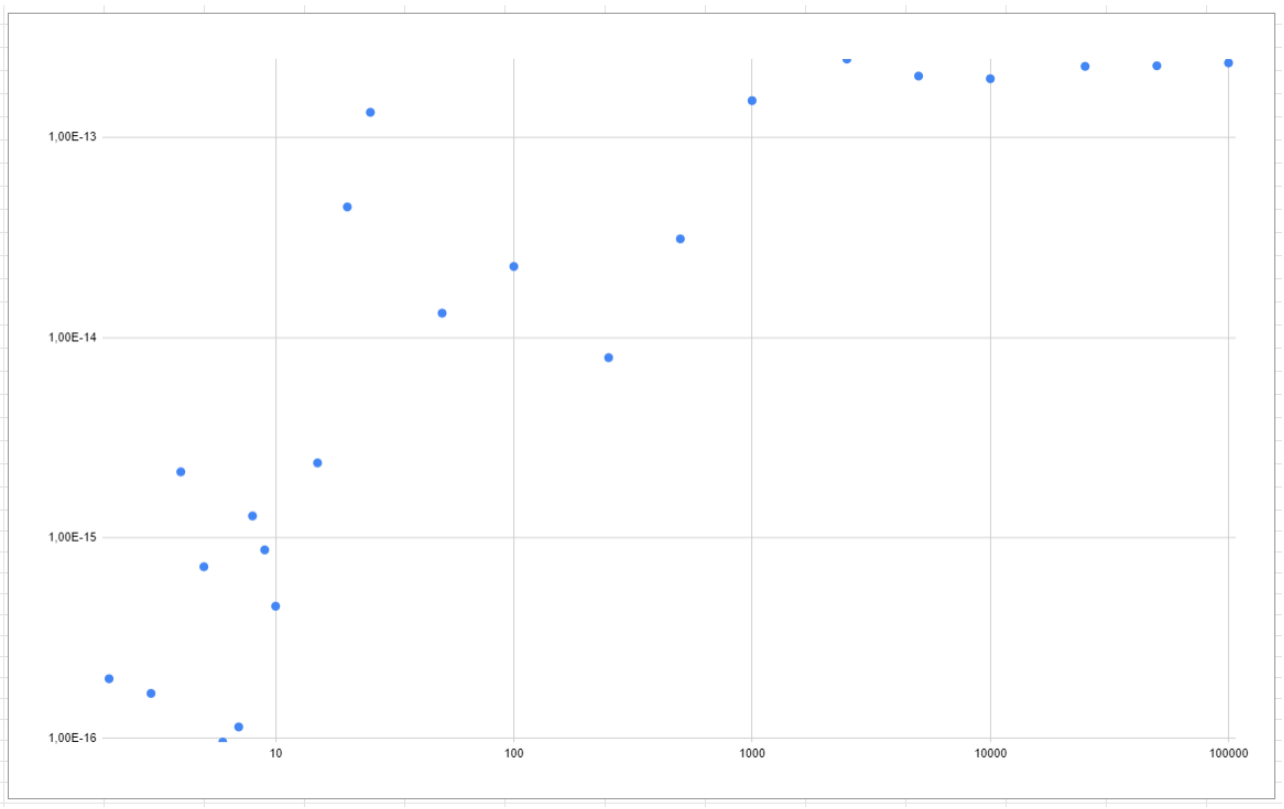
Размерность	Итерации	$\|\|x^* - x_k\|\|$	$\frac{\|\|x^* - x_k\|\|}{\|\|x^*\|\|}$	$cond(A) \geq$
2	3	6,28037e-16	2,80867e-16	0,554700
3	4	4,96507e-16	1,32697e-16	0,631791
4	5	3,79430e-15	6,92741e-16	0,369456
5	6	5,38887e-15	7,26635e-16	0,238755
6	7	1,55431e-15	1,62936e-16	0,374701
7	8	2,31821e-15	1,95925e-16	0,440126
8	9	6,77600e-15	4,74415e-16	1,17334
9	10	2,22322e-15	1,31692e-16	0,569939
10	11	5,82523e-15	2,96881e-16	0,665859
15	16	4,49057e-15	1,27524e-16	0,376919
20	20	2,40128e-12	4,48230e-14	0,267972
25	23	9,87875e-12	1,32903e-13	0,168114
50	31	2,74075e-12	1,32286e-14	0,0769214
100	37	1,31188e-11	2,25533e-14	0,0499021
250	56	1,80365e-11	7,87957e-15	0,0310406
500	84	1,99249e-10	3,08213e-14	0,0425260
1000	98	2,74796e-09	1,50399e-13	0,199575
2500	102	1,43903e-08	1,99338e-13	0,250628
5000	98	5,14105e-08	2,51821e-13	0,253546
10000	96	1,11736e-07	1,93517e-13	0,247084
25000	91	5,09269e-07	2,23144e-13	0,252470
50000	88	1,45078e-06	2,24750e-13	0,243091
100000	85	4,24415e-06	2,32460e-13	0,245986


1.00E-08
5.00E-09
1.00E-09
5.00E-10
1.00E-10
5.00E-11
Относительная погрешность
10
100
1000
10000
100000
Размерность

5.2. Обратный знак внедиагональных элементов

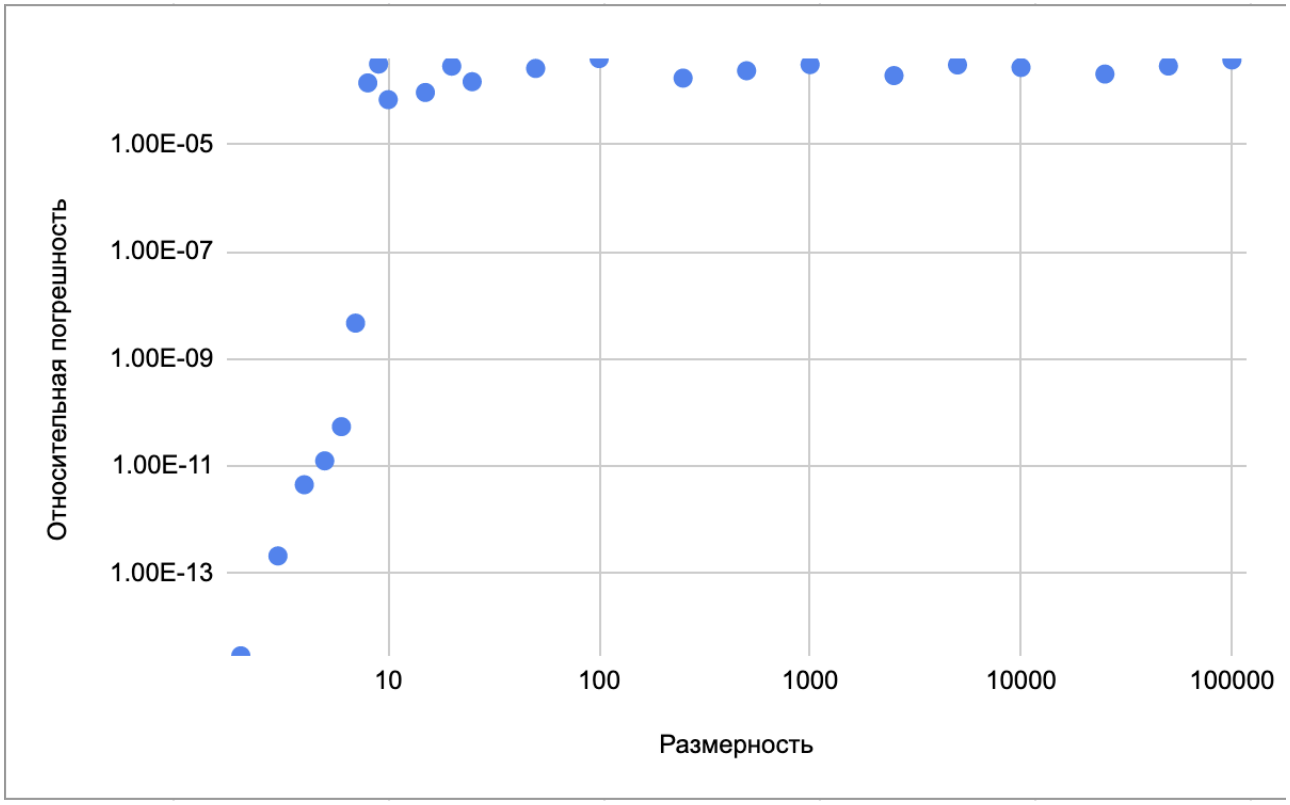
В этом пункте были исследованы те же самые матрицы, что и в предыдущем пункте, но с обратным знаком внедиагональных элементов.

Размерность	Итерации	$\|\|x^* - x_k\|\|$	$\frac{\|\|x^* - x_k\|\|}{\|\|x^*\|\|}$	$cond(A) \geq$
2	3	4,44089e-16	1,98603e-16	0,632076
3	4	6,28037e-16	1,67850e-16	0,672062
4	5	1,17180e-14	2,13940e-15	0,330900
5	6	5,33369e-15	7,19195e-16	0,297984
6	7	9,15513e-16	9,59719e-17	1,21374
7	8	1,35064e-15	1,14150e-16	0,395937
8	9	1,84123e-14	1,28912e-15	3,56059
9	10	1,47304e-14	8,72556e-16	1,99736
10	11	8,96535e-15	4,56917e-16	0,866213
15	16	8,34770e-14	2,37059e-15	3,58344
20	20	2,40449e-12	4,48830e-14	0,268379
25	23	9,89324e-12	1,33098e-13	0,168127
50	31	2,74390e-12	1,32438e-14	0,0767542
100	37	1,31699e-11	2,26412e-14	0,0498228
250	56	1,81865e-11	7,94508e-15	0,0309929
500	84	2,01026e-10	3,10962e-14	0,0425130
1000	99	2,78120e-09	1,52218e-13	0,199731
2500	102	1,77264e-08	2,45551e-13	0,249571
5000	99	4,12177e-08	2,01894e-13	0,258483
10000	96	1,13072e-07	1,95831e-13	0,246926
25000	91	5,15103e-07	2,25700e-13	0,252333
50000	88	1,46669e-06	2,27215e-13	0,243020
100000	85	4,28873e-06	2,34902e-13	0,245937


1,00E-13
1,00E-14
1,00E-15
1,00E-16
10
100
1000
10000
100000

5.3. Матрицы Гильберта

Размерность	Итерации	$\|\|x^* - x_k\|\|$	$\frac{\|\|x^* - x_k\|\|}{\|\|x^*\|\|}$	$cond(A) \geq$
2	3	6,39319e-15	2,85912e-15	1,23371
3	4	7,84641e-13	2,09704e-13	0,729763
4	6	2,45390e-11	4,48020e-12	6,00042
5	8	9,27316e-11	1,25039e-11	661,300
6	11	5,19394e-10	5,44473e-11	110546
7	14	5,52819e-08	4,67217e-09	5,67113e+06
8	14	0,00204409	0,000143115	8,61373e+09
9	14	0,00543457	0,000321916	1,50964e+09
10	19	0,00136710	6,96740e-05	3,91686e+10
15	21	0,00334281	9,49293e-05	1,75815e+08
20	23	0,0157595	0,000294171	1,59713e+09
25	25	0,0112051	0,000150747	2,86272e+08
30	25	0,0259429	0,000266802	6,61678e+08
35	23	0,0497683	0,000407581	6,48326e+08
40	30	0,0261696	0,000175877	1,73751e+09
45	30	0,0427951	0,000241526	2,00997e+09
50	29	0,0651385	0,000314400	4,77411e+08
100	41	0,114113	0,000196180	3,42877e+08
200	48	0,508023	0,000309937	4,91763e+08
300	53	0,836914	0,000278276	2,88505e+08
400	66	0,969961	0,000209610	1,69737e+09
500	61	1,91233	0,000295813	6,78603e+08
600	58	3,26194	0,000383943	3,98762e+08
700	73	2,60394	0,000243265	4,65755e+08
800	72	3,83489	0,000293273	6,05946e+08
900	69	5,35921	0,000343507	4,58340e+08
1000	87	3,81897	0,000209017	5,68989e+08



5.4. Итоги исследований МСГ

В силу особенностей метода сопряженных градиентов его точность превосходит точность методов LU и Гаусса, так как выход из итерационного процесса просиходит по достижению заданной точности решения.

На матрицах Гильберта экспоненциальный рост оценки на число обусловленности матрицы с увеличением размерности не приводил к замедлению сходимости метода.

6. Программная реализация

6.1. Основные интерфейсы

```
/**
 * Common interface for copyable objects
 */
public interface Copyable<T> {
    T copy();
}
```

```
/**
 * Common interface for read-only matrix
 */
public interface MatrixView {
    /**
     * Gets element from profile matrix.
     *
     * @param i row index.
     * @param j column index.
     * @return elements value.
     */
    double get(final int i, final int j);

    /**
     * @return matrix dimension
     */
    int getN();

    /**
     * Default multiplication implementation.
     */
    default Vector mul(Vector b) {
        throw new UnsupportedOperationException();
    }
}
```

```
/**
 * Matrix interface.
 */
public interface Matrix<T> extends MatrixView, Copyable<T> {
    /**
     * Sets value to an element in profile matrix.
     *
     * @param i     row index.
     * @param j     column index.
     * @param value given value to set.
```

```
     */
    void set(final int i, final int j, final double value);

    /**
     * Computes LU decomposition
     */
    default LUView luDecomposition() {
        double sum;
        int n = getN();
        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                sum = 0;
                for (int k = 0; k < i; k++) {
                    sum += get(i, k) * get(k, j);
                }
                set(i, j, get(i, j) - sum);
            }
            // L
            for (int j = i + 1; j < n; j++) {
                sum = 0;
                for (int k = 0; k < i; k++) {
                    sum += get(j, k) * get(k, i);
                }
                set(j, i, (1.0 / get(i, i)) * (get(j, i) - sum));
            }
        }
        return new LUView(this);
    }
}
```

6.2. Матрица в профильном формате

```
/**
 * Representation of profile matrix.
 */
public class ProfileMatrix implements Matrix<ProfileMatrix> {
    private static final double EPS = 1e-7;

    /**
     * Matrix dimension.
     */
    final int n;

    /**
     * Matrix elements starting from first non-zero above main
   diagonal.
     */
    final double[] au;
```

```
    /**
     * Elements located on right part of equatation.
     */
    final double[] r;

    /**
     * Matrix elements starting from first non-zero below main
diagonal.
     */
    final double[] al;


    /**
     * "Matrix profile".
     */
    final int[] ia;

    /**
     * Main diagonal values.
     */
    final double[] di;

    private ProfileMatrix(
            final int n,
            final double[] au,
            final double[] al,
            final int[] ia,
            final double[] di,
            final double[] r
    ) {
        this.n = n;
        this.au = au;
        this.al = al;
        this.ia = ia;
        this.di = di;
        this.r = r;
    }

    private ProfileMatrix(final double[][] matrix) {
        di = new double[matrix.length];
        List<Double> au = new ArrayList<>();
        List<Double> al = new ArrayList<>();
        List<Integer> ia = new ArrayList<>(List.of(1));

        for (int i = 0; i < matrix.length; i++) {
            di[i] = matrix[i][i];
            int firstNonZero = -1;
            for (int j = 0; j < i; j++) {
                if (firstNonZero == -1 && (Math.abs(matrix[i][j]) >
EPS || Math.abs(matrix[j][i]) > EPS)) {
```

```
                    firstNonZero = j;
                    ia.add(ia.get(ia.size() - 1) + (i - firstNonZero
));
                }
                if (firstNonZero != -1) {
                    au.add(matrix[j][i]);
                    al.add(matrix[i][j]);
                }
            }
            if (firstNonZero == -1) {
                ia.add(ia.get(ia.size() - 1));
            }
        }

        this.n = matrix.length;
        this.au = au.stream().mapToDouble(i -> i).toArray();
        this.al = al.stream().mapToDouble(i -> i).toArray();
        this.ia = ia.stream().mapToInt(i -> i).toArray();
        this.r = null;

    }

    @SuppressWarnings("CopyConstructorMissesField")
    public ProfileMatrix(ProfileMatrix another) {
        this(
                another.n,
                Arrays.copyOf(another.au, another.au.length),
                Arrays.copyOf(another.al, another.al.length),
                Arrays.copyOf(another.ia, another.ia.length),
                Arrays.copyOf(another.di, another.di.length),
                (another.r == null ? null : Arrays.copyOf(another.r,
another.r.length))
        );
    }

    public static ProfileMatrix of(
            final int n,
            final double[] au,
            final double[] al,
            final int[] ia,
            final double[] di,
            final double[] r
    ) {
        return new ProfileMatrix(n, au, al, ia, di, r);
    }

    public static ProfileMatrix of(final double[][] matrix) {
        return new ProfileMatrix(matrix);
    }
```

```
    /**
     * Gets element from profile matrix.
     *
     * @param i row index.
     * @param j column index.
     * @return elements value.
     */
    public double get(int i, int j) {
        if (i == j) {
            return di[i];
        } else if (i < j) {
            int profileNum = ia[j + 1] - ia[j];
            int firstInProfile = j - profileNum;
            return i < firstInProfile ? 0 : au[ia[j] + i -
firstInProfile - 1];
        } else {
            int profileNum = ia[i + 1] - ia[i];
            int firstInProfile = i - profileNum;
            return j < firstInProfile ? 0 : al[ia[i] + j -
firstInProfile - 1];
        }
    }

    /**
     * @return Matrix dimension
     */
    @Override
    public int getN() {
        return n;
    }

    /**
     * Sets value to an element in profile matrix.
     *
     * @param i     row index.
     * @param j     column index.
     * @param value given value to set.
     */
    public void set(int i, int j, double value) {
        if (i == j) {
            di[i] = value;
        } else if (i < j) {
            int profileNum = ia[j + 1] - ia[j];
            int firstInProfile = j - profileNum;
            if (i >= firstInProfile) {
                au[ia[j] + i - firstInProfile - 1] = value;
            }
        } else {
            int profileNum = ia[i + 1] - ia[i];
```

```
            int firstInProfile = i - profileNum;
            if (j >= firstInProfile) {
                al[ia[i] + j - firstInProfile - 1] = value;
            }
        }
    }

    /**
     * Multiplication profile matrix on vector.
     *
     * @param vector given vector.
     * @return multiplied vector.
     */
    public Vector mul(final Vector vector) {
        final double[] result = new double[n];
        int n = getN();
        for (int i = 0; i < n; i++) {
            result[i] = di[i] * vector.get(i);
        }

        for (int i = 1; i < n; ++i) {
            for (int j = ia[i]; j < ia[i + 1]; ++j) {
                int k = j - ia[i];
                int profileStart = i - ia[i + 1] + ia[i];
                result[i] += al[j - 1] * vector.get(profileStart + k
);
                result[profileStart + k] += au[j - 1] * vector.get(i
);
            }
        }
        return Vector.of(result);
    }

    @Override
    public String toString() {
        final String lineSeparator = System.lineSeparator();

        return "n=" + n + lineSeparator +
                "au=" + Arrays.toString(au) + lineSeparator +
                "al=" + Arrays.toString(al) + lineSeparator +
                "ia=" + Arrays.toString(ia) + lineSeparator +
                "di=" + Arrays.toString(di) + lineSeparator +
                "r=" + Arrays.toString(r) + lineSeparator;
    }

    @Override
    public ProfileMatrix copy() {
        return new ProfileMatrix(this);
    }
}
```

6.3. Разреженная матрица

```
/**
 * Sparse matrix implementation
 */
public class SparseMatrix implements Matrix<SparseMatrix> {
    /**
     * Zero element threshold
     */
    private static final double EPS = 1e-12;

    /**
     * Matrix dimension
     */
    private final int n;

    /**
     * di — main diagonal.
     * al — Matrix elements starting from first non-zero below
   main diagonal.
     * au — Matrix elements starting from first non-zero above
   main diagonal.
     * ia — Profile.
     * ja — Profile indices.
     */
    private final double[] di, al, au;
    private final int[] ia, ja;

    public SparseMatrix(int n, double[] di, double[] al, double[] au
   , int[] ia, int[] ja) {
        this.n = n;
        this.di = di;
        this.al = al;
        this.au = au;
        this.ia = ia;
        this.ja = ja;
    }

    /**
     * Converts from full matrix to sparse matrix.
     */
    public SparseMatrix(double[][] a) {
        List<Double> al = new ArrayList<>(), au = new ArrayList<>();
        List<Integer> ia = new ArrayList<>(List.of(1)), ja = new
   ArrayList<>();
        n = a.length;
        di = new double[n];
        for (int i = 0; i < n; i++) {
            assert a[i].length == n;
            di[i] = a[i][i];
```

```
            int nonZeroCount = 0;
            for (int j = 0; j < i; j++) {
                if ((Math.abs(a[i][j]) > EPS || Math.abs(a[j][i]) >
EPS)) {
                    nonZeroCount++;

                    ja.add(j);
                    al.add(a[i][j]);
                    au.add(a[j][i]);
                }
            }

            ia.add(ia.get(ia.size() - 1) + nonZeroCount);
        }

        this.au = au.stream().mapToDouble(i -> i).toArray();
        this.al = al.stream().mapToDouble(i -> i).toArray();
        this.ia = ia.stream().mapToInt(i -> i).toArray();
        this.ja = ja.stream().mapToInt(i -> i).toArray();
    }

    /**
     * Setter.
     * @param i     row index.
     * @param j     column index.
     * @param value given value to set.
     */
    @Override
    public void set(int i, int j, double value) {
        if (i == j) {
            di[i] = value;
            return;
        }

        if (i < j) {
            int index = Arrays.binarySearch(ja, ia[j] - 1, ia[j + 1]
- 1, i);
            if (index < 0) {
                throw new IllegalArgumentException();
            } else {
                au[index] = value;
            }
        } else {
            int index = Arrays.binarySearch(ja, ia[i] - 1, ia[i + 1]
- 1, j);
            if (index < 0) {
                throw new IllegalArgumentException();
            } else {
                al[index] = value;
            }
```

```
        }
    }

    /**
     * @param i row index.
     * @param j column index.
     * @return a[i][j]
     */
    @Override
    public double get(int i, int j) {
        if (i == j) {
            return di[i];
        }

        if (i < j) {
            int index = Arrays.binarySearch(ja, ia[j] - 1, ia[j + 1]
- 1, i);
            if (index < 0) {
                return 0;
            } else {
                return au[index];
            }
        } else {
            int index = Arrays.binarySearch(ja, ia[i] - 1, ia[i + 1]
- 1, j);
            if (index < 0) {
                return 0;
            } else {
                return al[index];
            }
        }
    }

    /**
     * @return dimension.
     */
    @Override
    public int getN() {
        return n;
    }

    /**
     * Optimized vector multiplication
     */
    @Override
    public Vector mul(Vector b) {
        final double[] result = new double[n];
        int n = getN();
        for (int i = 0; i < n; i++) {
            result[i] = di[i] * b.get(i);
```

```
        }

        for (int i = 1; i < n; ++i) {
            for (int j = ia[i]; j < ia[i + 1]; ++j) {
                result[i] += al[j - 1] * b.get(ja[j - 1]);
                result[ja[j - 1]] += au[j - 1] * b.get(i);
            }
        }
        return Vector.of(result);
    }

    @Override
    public SparseMatrix copy() {
        return new SparseMatrix(
                n,
                Arrays.copyOf(di, di.length),
                Arrays.copyOf(al, al.length),
                Arrays.copyOf(au, au.length),
                Arrays.copyOf(ia, ia.length),
                Arrays.copyOf(ja, ja.length)
        );
    }

    public double[] getAl() {
        return Arrays.copyOf(al, al.length);
    }

    public double[] getAu() {
        return Arrays.copyOf(au, au.length);
    }
}
```

6.4. Плотная матрица

```
/**
 * Full matrix implementation.
 */
public class FullMatrix implements Matrix<FullMatrix> {

    /**
     * Matrix.
     */
    private final double[][] matrix;

    /**
     * Dimension.
     */
    private final int n;
```

```
    public FullMatrix(final double[][] matrix) {
        this.matrix = matrix;
        this.n = matrix.length;
    }

    /**
     * Gets element from matrix.
     *
     * @param i row index.
     * @param j column index.
     * @return elements value.
     */
    public double get(final int i, final int j) {
        return matrix[i][j];
    }

    /**
     * @return matrix size
     */
    @Override
    public int getN() {
        return n;
    }

    /**
     * Sets value to an element in profile matrix.
     *
     * @param i     row index.
     * @param j     column index.
     * @param value given value to set.
     */
    public void set(final int i, final int j, double value) {
        matrix[i][j] = value;
    }

    /**
     * Swaps elements in matrix
     *
     * @param i first row index.
     * @param j first column index.
     * @param a second row index.
     * @param b second column index.
     */
    public void swap(final int i, final int j, final int a, final
int b) {
        double tmp = matrix[i][j];
        matrix[i][j] = matrix[a][b];
        matrix[a][b] = tmp;
    }
```

```
    /**
     * Finds main element in current matrix with given k.
     *
     * @param k given k.
     * @return index of main element.
     */
    public int findMainElement(final int k) {
        int ind = k;
        for (int i = k; i < n; i++) {
            if (Math.abs(get(i, k)) > Math.abs(get(ind, k))) {
                ind = i;
            }
        }
        return ind;
    }

    /**
     * Copies current matrix.
     *
     * @return copy of current matrix.
     */
    public FullMatrix copy() {
        double[][] m = new double[n][n];
        for (int i = 0; i < n; i++) {
            m[i] = Arrays.copyOf(matrix[i], n);
        }

        return new FullMatrix(m);
    }
}
```

6.5. Вспомогательные типы матриц

```
/**
 * LU-view of in-place LU-decomposition.
 */
public class LUMatrix implements MatrixView {
    private final LUView luView;

    public LUMatrix(LUView luView) {
        this.luView = luView;
    }

    /**
     * @return if i <= j, element from U, or element from L if i > j
     .
     */
    @Override
    public double get(int i, int j) {
```

```
        if (i <= j) {
            return luView.getU().get(i, j);
        } else {
            return luView.getL().get(i, j);
        }
    }

    /**
     * @return matrix size
     */
    @Override
    public int getN() {
        return luView.getL().getN();
    }
}
```

```
/**
 * Result of LU-decomposition
 */
public class LUView {
    private final MatrixView a;

    public LUView(MatrixView a) {
        this.a = a;
    }

    /**
     * @return MatrixView of L matrix
     */
    public MatrixView getL() {
        return new MatrixView() {
            @Override
            public double get(int i, int j) {
                return i < j ? 0.0 : (i == j ? 1.0 : a.get(i, j));
            }

            @Override
            public int getN() {
                return a.getN();
            }
        };
    }

    /**
     * @return MatrixView of U matrix
     */
    public MatrixView getU() {
        return new MatrixView() {
            @Override
```

```
            public double get(int i, int j) {
                return (i > j ? 0.0 : a.get(i, j));
            }

            @Override
            public int getN() {
                return a.getN();
            }
        };
    }
}
```

6.6. Генератор матриц

```
/**
 * Profile matrix generator.
 */
public class ProfileMatrixGenerator {

    /**
     * Generates matrix in profile format.
     *
     * @return profile format matrix.
     */
    public static ProfileMatrix generateMatrix() {
        var rnd = new Random(System.currentTimeMillis());
        int n = Math.abs(rnd.nextInt()) % 10;
        double[] di = new double[n];

        double[] r = new double[n];

        for (int i = 0; i < n; i++) {
            di[i] = rnd.nextDouble();
            r[i] = rnd.nextDouble();
        }

        int[] ia = new int[n + 1];
        ia[0] = ia[1] = 1;

        final List<Double> au = new ArrayList<>();
        final List<Double> al = new ArrayList<>();

        for (int i = 2; i <= n; i++) {
            int idx = Math.abs(rnd.nextInt()) % i;
            int t = i - idx - 1;
            ia[i] = ia[i - 1] + t;
            for (int j = 0; j < t; j++) {
                au.add((rnd.nextInt() % 2 == 0) ?  rnd.nextDouble()
   * (rnd.nextInt() % 100) : 0);
```

```
                al.add((rnd.nextInt() % 2 == 0) ?  rnd.nextDouble()
* (rnd.nextInt() % 100) : 0);
            }
        }

        return ProfileMatrix.of(
                n,
                au.stream().mapToDouble(i -> i).toArray(),
                al.stream().mapToDouble(i -> i).toArray(),
                ia,
                di,
                r
        );
    }

    /**
     * Generates dense matrix dimension n.
     * @param n given n.
     * @param k given k for computing.
     * @return dense matrix.
     */
    public static double[][] generateDenseMatrix(int n, int k) {
        double[][] matrix = new double[n][n];
        Random random = new Random();
        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                if (i != j) {
                    matrix[j][i] = -random.nextInt(5);
                    matrix[i][j] = -random.nextInt(5);
                }
            }
        }

        for (int i = 0; i < n; i++) {
            double sum = 0;
            for (int j = 0; j < n; j++) {
                if (i != j) {
                    sum += matrix[i][j];
                }
            }
            matrix[i][i] = -sum;
        }

        matrix[0][0] += Math.pow(10.0, -k);

        return matrix;
    }

    /**
     * Generates Gilbert matrix dimension n.
```

```
     * @param n given dimension.
     * @return Gilbert matrix.
     */
    public static double[][] generateGilbertMatrix(final int n) {
        double[][] matrix = new double[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                matrix[i][j] = 1.0 / ((i + 1) + (j + 1) - 1);
            }
        }

        return matrix;
    }

    /**
     * Main method. Generates matrix and writes it to file.
     * @param args program arguments
     * @throws IOException if some exception with files occuired.
     */
    public static void main(String[] args) throws IOException {
        System.out.print("Enter directory name: ");
        final var sc = new Scanner(System.in);
        final String dirName = sc.next();
        Files.createDirectories(Path.of("tests"));
        final Path dirPath = Path.of("tests", dirName);
        final Path filePath = dirPath.resolve(Path.of("test.txt"));
        Files.createDirectories(dirPath);
        final FileWriter fl = new FileWriter(filePath.toString());
        fl.write(ProfileMatrixGenerator.generateMatrix().toString())
   ;
        fl.close();
    }
}
```

6.7. Общий интерфейс решателей

```
/**
 * Common interface for linear equations solvers
 * @param <T> Matrix type
 */
public interface Solver<T extends MatrixView> {
    Vector solve(T a, Vector b);
}
```

6.8. Гаусс-решатель

```java
/**
 * Gauss solver
 */
public class GaussSolver implements Solver<FullMatrix> {
    private static final double DEFAULT_EPS = 1e-12;

    /**
     * Solves Ax = b with custom EPS
     */
    public Vector solve(FullMatrix a, Vector b, double eps) {
        final FullMatrix m = a.copy();
        final Vector v = b.copy();
        int n = m.getN();
        assert n == v.size();

        final double[] result = new double[n];
        for (int i = 0; i < n; i++) {
            final int ind = m.findMainElement(i);
            if (Math.abs(m.get(ind, i)) < eps) {
                return null;
            }
            v.swap(i, ind);
            for (int j = i; j < n; j++) {
                m.swap(i, j, ind, j);
            }

            for (int j = i + 1; j < n; j++) {
                double mul = m.get(j, i) / m.get(i, i);
                for (int k = i; k < n; k++) {
                    m.set(j, k, m.get(j, k) - mul * m.get(i, k));
                }
                v.set(j, v.get(j) - mul * v.get(i));
            }
        }

        for (int i = n - 1; i >= 0; i--) {
            result[i] = v.get(i);
            for (int j = i + 1; j < n; j++) {
                result[i] -= m.get(i, j) * result[j];
            }
            result[i] /= m.get(i, i);
        }

        return Vector.of(result);
    }

    /**
     * Solves Ax = b with default EPS
     */
    @Override
```

```
    public Vector solve(FullMatrix a, Vector b) {
        return solve(a, b, DEFAULT_EPS);
    }
}
```

6.9. LU-решатель

```
/**
 * Solves linear system. Use performed LU-decomposition
 */
public class LUSolver implements Solver<LUMatrix> {
    @Override
    public Vector solve(LUMatrix a, Vector b) {
        int n = a.getN();

        double[] y = new double[n];
        double sum;
        for (int i = 0; i < n; i++) {
            sum = 0;
            for (int k = 0; k < i; k++) {
                sum += a.get(i, k) * y[k];
            }
            y[i] = b.get(i) - sum;
        }

        double[] x = new double[n];
        for (int i = n - 1; i >= 0; i--) {
            sum = 0;
            for (int k = i + 1; k < n; k++) {
                sum += a.get(i, k) * x[k];
            }
            x[i] = (y[i] - sum) / (a.get(i, i));
        }
        return Vector.of(x);
    }
}
```

```
/**
 * The same as LUSolver but performs LU-decomposition
 */
public class LUCompleteSolver implements Solver<ProfileMatrix> {
    private static final LUSolver luSolver = new LUSolver();

    private LUView luView;

    @Override
    public Vector solve(ProfileMatrix a, Vector b) {
        a = a.copy();
```

```
        luView = a.luDecomposition();
        return luSolver.solve(new LUMatrix(getLUView()), b);
    }

    public LUView getLUView() {
        return luView;
    }
}
```

6.10. Метод сопряженных градиентов

```
/**
 * Conjugate Gradient Solver
 */
public class ConjugateGradientSolver implements Solver<SparseMatrix>
    {
    private static final int MAX_ITERATIONS = 10000;
    private static final double EPS = 1e-12;

    private final List<Vector> x = new ArrayList<>();
    private final List<Vector> r = new ArrayList<>();
    private final List<Double> rScalarSqr = new ArrayList<>();
    private final List<Vector> z = new ArrayList<>();
    private final double eps;

    public ConjugateGradientSolver(Vector x0) {
        this(x0, EPS);
    }

    public ConjugateGradientSolver(Vector x0, double eps) {
        this.x.add(x0);
        this.eps = eps;
    }

    private static void checkSymmetry(SparseMatrix a) {
        double[] al = a.getAl();
        double[] au = a.getAu();
        assert al.length == au.length;
        for (int i = 0; i < al.length; i++) {
            if (Math.abs(al[i] - au[i]) > EPS) {
                throw new IllegalArgumentException();
            }
        }
    }

    public int getIterationCount() {
        return x.size();
    }
```

```
    /**
     * Solves ax = b
     */
    @Override
    public Vector solve(SparseMatrix a, Vector b) {
        checkSymmetry(a);
        r.add(b.sub(a.mul(x.get(0))));
        rScalarSqr.add(r.get(0).scalarMul(r.get(0)));
        z.add(r.get(0));

        for (int k = 1; k < MAX_ITERATIONS; k++) {
            Vector rK1 = r.get(k - 1);
            Vector zK1 = z.get(k - 1);
            double alpha = rScalarSqr.get(k - 1) / a.mul(zK1).
   scalarMul(zK1);

            x.add(x.get(k - 1).add(zK1.mul(alpha)));

            Vector rK = rK1.sub(a.mul(zK1).mul(alpha));

            if (rK.norm() / b.norm() < eps) {
                return x.get(k);
            }

            r.add(rK);
            rScalarSqr.add(rK.scalarMul(rK));

            double beta = rScalarSqr.get(k) / rScalarSqr.get(k - 1);
            Vector zK = rK.add(zK1.mul(beta));
            z.add(zK);
        }

        return x.get(MAX_ITERATIONS - 1);
    }
}
```

6.11. Представление вектора

```
/**
 * Util class for one dimension matrix.
 */
public class Vector implements Copyable<Vector> {

    /**
     * Vector.
     */
    private final double[] vector;
```

```
    /**
     * Size.
     */
    private final int n;

    private Vector(final double... vector) {
        this.vector = vector;
        this.n = vector.length;
    }

    public static Vector of(final double... vector) {
        return new Vector(vector);
    }

    public static Vector zero(int n) {
        return Vector.of(new double[n]);
    }

    public static Vector natural(int n) {
        double[] v = new double[n];
        for (int i = 0; i < n; i++) {
            v[i] = i + 1;
        }
        return Vector.of(v);
    }

    /**
     * Size of current vector.
     *
     * @return size.
     */
    public int size() {
        return n;
    }

    /**
     * Get element by index.
     *
     * @param i index.
     * @return value on index.
     */
    public double get(final int i) {
        return vector[i];
    }

    /**
     * Sets value on given index.
     *
     * @param i     index.
```

```
     * @param value value
     */
    public void set(final int i, final double value) {
        vector[i] = value;
    }

    public Vector sub(final Vector a) {
        double[] result = new double[n];
        for (int i = 0; i < n; i++) {
            result[i] = vector[i] - a.get(i);
        }
        return Vector.of(result);
    }

    /**
     * Norm of current vector.
     *
     * @return norm.
     */
    public double norm() {
        double res = 0;
        for (int i = 0; i < n; i++) {
            double a = get(i);
            res += a * a;
        }
        return Math.sqrt(res);
    }

    /**
     * Swaps two elements in vector.
     *
     * @param i first index.
     * @param j second index.
     */
    public void swap(final int i, final int j) {
        double tmp = vector[i];
        vector[i] = vector[j];
        vector[j] = tmp;
    }

    /**
     * Multiplies vector to a number. Returns new vector.
     */
    public Vector mul(double t) {
        double[] result = new double[n];
        for (int i = 0; i < n; i++) {
            result[i] = get(i) * t;
        }
        return Vector.of(result);
    }
```

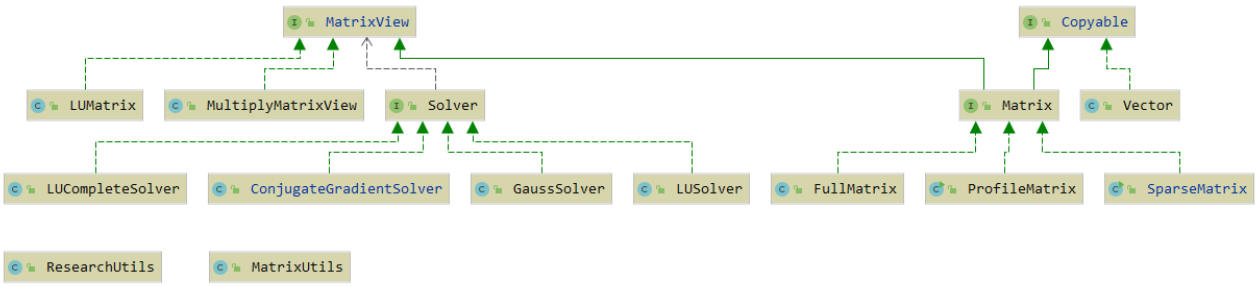
```java
    /**
     * Scalar multiplication. Returns new vector.
     */
    public double scalarMul(Vector b) {
        assert b.size() == n;
        double result = 0.0;
        for (int i = 0; i < n; i++) {
            result += get(i) * b.get(i);
        }
        return result;
    }

    /**
     * Two vectors addition. Returns result vector.
     */
    public Vector add(Vector b) {
        assert b.size() == n;
        double[] result = new double[n];
        for (int i = 0; i < n; i++) {
            result[i] = get(i) + b.get(i);
        }
        return Vector.of(result);
    }

    /**
     * Copies current vector.
     *
     * @return new copy.
     */
    public Vector copy() {
        return Vector.of(Arrays.copyOf(vector, n));
    }

    @Override
    public String toString() {
        return "Vector{" +
                "vector=" + Arrays.toString(vector) +
                ", n=" + n +
                '}';
    }
}
```

7. Диаграмма классов



8. Выводы

Проведя исследования на различных методах решений СЛАУ, а также принимая во внимание различные форматы хранения матриц, мы пришли к следующим выводам.

- Метод Гаусса работает асимптотически аналогично методу LU-разложения, за $O(n^3)$. Однако, скрытая в асимптотике константа в методе Гаусса меньше, чем у LU-разложения, поэтому на достаточно больших матрицах он работает быстрее.
- Метод LU-разложения подходит, когда получаемая матрица хранится в профильном формате. Методу не требуется изменять элементы вне профиля, таким образом сокращается требуемая память для работы.
- Метод LU-разложения работает медленнее, чем метод Гаусса, но показывает несколько б*о*льшую точность.
- Число обусловленности является одним из основных факторов влияния на ошибку вычислений. При увеличении размерности ошибка будет также увеличиваться.

  Так же приближение элементов матрицы к нулю влечет за собой заметное увеличение погрешности из-за появления неточных сложений с числами, близкими к нулю.