МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Дальневосточный федеральный университет»
(ДВФУ)

ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ

Кафедра математического и компьютерного моделирования

Кулахсзян Сергей Грайрович
Захватов Дмитрий Алексеевич

ОТЧЁТ ПО ВЫПОЛНЕНИЮ КУРСОВОГО ПРОЕКТА ПО ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКЕ

Направление подготовки 02.03.01сцт Сквозные цифровые технологии,
бакалавриатская программа «Математика и компьютерные науки»
Очной (заочной) формы обучения

Студенты группы Б9123-02.03.01сцт2
_____ С. Г. Кулахсзян
_____ Д. А. Захватов

Руководитель проекта
_____ Т. В. Пак

Владивосток
2026

Содержание

1 Введение

В ходе выполнения данного курсового проекта предстоит познакомиться и реализовать по 2 прямых и итерационных метода решения систем линейных алгебраических уравнения и проанализировать данные методы в сравнении по выбранным критериям. В качестве критериев сравнения методов были избраны: **скорость выполнения**, **погрешность решения** (как сильно получившийся в ходе решения ответ отличается от правильного), **устойчивость решения** (число относительного возмущения решения при возмущении вектора правой части b на некоторый процент ϵ).

2 Анализ методов

Для проведения анализа методов было принято решение разделить задачу на этапы:

1. Создание цикла с итерацией по всем размерам матриц от 1 до указанного max_n;
2. Генерация случайного массива левой части A размера $n \times n$, генерация вектора x_{true} размера n, получение вектора правой части b в качестве результата произведения $A \cdot x$;
3. Проведение анализа по каждому критерию для каждого метода;
4. Построение графиков.

2.1 Анализ скорости выполнения

На этом этапе происходит запуск решения каждого метода с попутным вычислением затраченного время.

Функция анализа времени принимает аргументы:

- `A` – матрица левой части;
- `b` – вектор правой части;
- `method` – функция с реализованным методом.

И возвращает:

- `end` – затраченное время;
- `x` – вектор решения.

Код:

```
1  def runtime_check(A: numpy.array, b: numpy.array, method:
   callable) -> tuple[float, numpy.array]:
2      start = time.time()
3      x = method(A.copy(), b.copy())
4      end = time.time() - start
5
6      return end, x
```

2.2 Анализ погрешности решения

В этом блоке происходит сравнения полученного решения x с правильным решением x_{true}. Для численного сравнения векторов вычисляется норма $||\cdot||_2$ разности $x - x_{true}$.

Функция анализа погрешности принимает аргументы:

- `x` – вектор решения;
- `x_true` – вектор правильного решения;

И возвращает значение нормы разности векторов.

Код:

```
1 def error_check(x: numpy.array, x_true: numpy.array) ->
  float:
2     return numpy.linalg.norm(x - x_true)
```

2.3 Анализ устойчивости решения

В последнем критерии создаётся случайное возмущение вектора правой части δb, для этого сначала генерируется случайный вектор размера n, после чего нормируется и умножается на параметр ϵ. После происходит расчёт нового вектора решения δx, полученный решением системы $A \cdot \delta x = b + \delta b$. В конце вычисляется относительная погрешность решения: $\frac{||x - \delta x||_2}{||x||_2}$.

Функция анализа устойчивости принимает аргументы:

- `A` – матрица левой части;
- `b` – вектор правой части;
- `x` – вектор решения;
- `method` – функция с реализованным методом;

- `epsilon` – параметр обусловленности.

И возвращает значение относительной погрешности решения.

Код:

```
1  def stability_check(A: numpy.array, b: numpy.array, x:
   numpy.array, n: int, method: callable, epsilon: float) ->
   float:
2      delta_b = numpy.random.rand(n)
3      delta_b = delta_b / numpy.linalg.norm(delta_b) *
       epsilon
4
5      delta_x = method(A.copy(), b + delta_b)
6
7      solution_relative_perturbation =
       numpy.linalg.norm(delta_x - x) / numpy.linalg.norm(x)
8      return solution_relative_perturbation
```

3 Прямые методы

Для реализации были избраны 2 прямых метода:

- Метод Гаусса с выбором максимального элемента по матрице для решения СЛАУ.
- Метод Гаусса с выбором максимального элемента по столбцу для решения СЛАУ;

3.1 Метод Гаусса с выбором максимального элемента по матрице для решения СЛАУ

Перед началом описания метода введём некоторые определения.

Элементарная нижняя треугольная матрица для матрицы A размера $n \times m$ – это матрица размера $n \times n$ вида:

$$L_i = \begin{pmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \dfrac{1}{a_{ii}} & 0 & \cdots & 0 \\ & & & & & \\ 0 & \cdots & -\dfrac{a_{i+1,i}}{a_{ii}} & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -\dfrac{a_{ni}}{a_{ii}} & 0 & \cdots & 1 \end{pmatrix}, i = \overline{1, n},$$

где $a_{ji}, j = \overline{i, n}$– это элементы матрицы A на позиции ji.

Элементарная матрица перестановок P_{ij} – матрица, полученная из единичной матрицы перестановкой i-ой и j-ой строк.

Суть данного метода заключается в том, чтобы при k-ом шаге прямого хода метода Гаусса в качестве элемента a_{kk} стоял максимальный элемент по модулю в матрице $A_k^{(k-1)}$, который является подматрицей $A^{(k-1)}$ с элементами, индексы строк и столбцов которых больше или равен k:

$A^{(k-1)} = (a^{(k-1)})_{i,j}^n$

$A_k^{(k-1)} = (a^{(k-1)})_{i,j}^n, \forall i, j \geq k$

, где $A^{(k-1)}$ – изменённая матрица A на $(k-1)$-ом шаге прямого хода метода Гаусса.

У нас имеется СЛАУ вида:

$Ax = b,$

где A – матрица $n \times n$, b – вектор из $\mathbb{R}^n$.

На первом шаге мы должны найти элемент $a_{ij} = \max|A|$, затем переместить его на место a_{11}. Для этого сначала нужно поменять местами

1-ю и i-ю строки, точно также местами поменяются элементы вектора b. Это эквивалентно тому, чтобы домножить на матрицу A и вектор b элементарную матрицу перестановок P_{1i} слева:

$P_{1i}Ax = P_{1i}b$

Далее необходимо поменять местами 1-й и j-й столбцы, точно также местами поменяются элементы вектора x. Это эквивалентно тому, чтобы домножить на матрицу A элементарную матрицу перестановок P_{1j} справа и домножить на вектор x обратную элементарную матрицу перестановок P_{1j}^{-1} слева:

$P_{1i}AP_{1j}(P_{1j}^{-1}x) = P_{1i}b$

Таким образом совершается эквивалентное преобразование исходной системы, приводящее к тому, чтобы на позиции a_{11} стоял максимальный элемент матрицы A по модулю. Далее в прямом ходе метода Гаусса совершается эквивалентное преобразование, которое заключается в том, чтобы a_{11} стал 1, а все элементы ниже него занулились. Для этого 1-я строка умножается на $\dfrac{1}{a_{11}}$, а i-я строка умножается на $-\dfrac{a_{11}}{a_{i1}}$ и суммируется с 1-й строкой. Это эквивалентно тому, чтобы домножить на матрицу $P_{1i}AP_{1j}$ и на вектор $P_{1i}b$ её элементарную нижнюю треугольную матрицу L_1 слева:

$L_1P_{1i}AP_{1j}(P_{1j}^{-1}x) = L_1P_{1i}b$

Таким образом получим СЛАУ, эквивалентная исходной, вида:

$A^{(1)}x^{(1)} = b^{(1)}$,

где $A^{(1)} = L_1P_{1i}AP_{1j}$, $x^{(1)} = P_{1j}^{-1}x$, $b^{(1)} = L_1P_{1i}b$

На $(k-1)$-ом шаге метода у нас будет СЛАУ:

$A^{(k-1)}x^{(k-1)} = b^{(k-1)}$

Действуя аналогично, приходим к новой СЛАУ вида:

$L_k P_{ki} A^{(k-1)} P_{kj} (P_{kj}^{-1} x^{(k-1)}) = L_k P_{ki} b^{(k-1)}$

Важное уточнение, что i и j на этом шаге отличаются от тех, которые были на первом шаге, под этими индексами обозначается местоположение максимального элемента подматрицы.

Совершив таких n шагов, получим итоговую СЛАУ:

$L_n P_{ni} A^{(n-1)} P_{nj} (P_{nj}^{-1} x^{(n-1)}) = L_n P_{ni} b^{(n-1)}$

$A^{(n)} x^{(n)} = b^{(n)}$

Этим завершается прямой ход метода Гаусса. На обратном ходе Гаусса совершается вычисление элементов вектора $x^{(n)}$:

$$x_n^{(n)} = \frac{b_n^{(n)}}{a_{nn}^{(n)}}$$

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

Таким образом получим значения для $x^{(n)}$, вспомним, что:

$x^{(n)} = P_{nj_n}^{-1} x^{(n-1)} = P_{nj_n}^{-1} P_{n-1,j_{n-1}}^{-1} x^{(n-2)} = \ldots = P_{nj_n}^{-1} P_{n-1,j_{n-1}}^{-1} \ldots P_{1j_1}^{-1} x$

Тогда x можно найти по формуле:

$x = P_{1j_1} P_{2j_2} \ldots P_{nj_n} x^{(n)}$

Код:

```
# Функция вычисления элементарной нижнетреугольной матрицы
def elementary_lower_triangular_matrix(i: int, A:
numpy.array) -> numpy.array:
    n = A.shape[0]
    L = numpy.eye(n)

    L[i, i] = 1.0 / A[i, i]
    for j in range(i + 1, n):
        L[j, i] = -A[j, i] * L[i, i]

```

```
    return L

# Функция вычисления элементарной перестановочной матрицы
def elementary_permutation_matrix(i: int, j: int, n: int)
-> numpy.array:
    P = numpy.eye(n)
    if i != j:
        P[[i, j]] = P[[j, i]]
    return P

# Метод Гаусса с выбором максимального элемента по матрице
def gaussian_method_with_maximum_element_by_matrix(A:
numpy.array, b: numpy.array) -> numpy.array:
    n = A.shape[0]
    P_x = numpy.eye(n)

    # Прямой ход
    for k in range(n):
        A_k = numpy.abs(A[k:, k:])
        max_i, max_j =
        numpy.unravel_index(numpy.argmax(A_k), A_k.shape)

        max_i += k
        max_j += k

        if (A[max_i, max_j] == 0):
            raise ValueError("Система не имеет
            единственного решения")

        A[[k, max_i]] = A[[max_i, k]]
        b[k], b[max_i] = b[max_i], b[k]

        A[:, [k, max_j]] = A[:, [max_j, k]]
        P = elementary_permutation_matrix(k, max_j, n)
        P_x = P_x @ P

        L_k = elementary_lower_triangular_matrix(k, A)
```

```
43
44              A = L_k @ A
45              b = L_k @ b
46
47          # Обратный ход
48          y = numpy.zeros(n)
49          y[n - 1] = b[n - 1] / A[n - 1, n - 1]
50          for k in range(n - 2, -1, -1):
51              y[k] = (b[k] - (A[k, k + 1:] @ y[k + 1:])) / A[k,
                k]
52
53          # Восстановление x
54          x = P_x @ y
55          return x
```

3.2 Метод Гаусса с выбором максимального элемента по столбцу для решения СЛАУ

Данный метод похож на предыдущий с одним отличием, что поиск максимального элемента происходит не по всей матрице, а лишь по столбцу. Таким образом на каждом k-ом шаге преобразованное СЛАУ выглядит так:

$L_k P_{ki} A^{(k-1)} x = L_k P_{ki} b^{(k-1)}$

Тогда итоговое СЛАУ имеет вид:

$L_n P_{ni} A^{(n-1)} x = L_n P_{ni} b^{(n-1)}$

$A^{(n)} x = b^{(n)}$

В связи с этим после обратного хода получается решение исходной системы, так как никаких преобразований вектора x не происходило.

Код:

```python
# Функция вычисления элементарной нижнетреугольной матрицы
def elementary_lower_triangular_matrix(i: int, A: numpy.array) -> numpy.array:
    n = A.shape[0]
    L = numpy.eye(n)

    L[i, i] = 1.0 / A[i, i]
    for j in range(i + 1, n):
        L[j, i] = -A[j, i] * L[i, i]

    return L

# Метод Гаусса с выбором максимального элемента по столбцу
def gaussian_method_with_maximum_element_by_column(A: numpy.array, b: numpy.array) -> numpy.array:
    n = A.shape[0]

    # Прямой ход
    for k in range(n):
        A_k = numpy.abs(A[k:, k])
        max_i = numpy.unravel_index(numpy.argmax(A_k), A_k.shape)[0]
        max_i += k

        if (A[max_i, k] == 0):
            raise ValueError("Система не имеет единственного решения")

        A[[k, max_i]] = A[[max_i, k]]
        b[k], b[max_i] = b[max_i], b[k]

        L_k = elementary_lower_triangular_matrix(k, A)
        A = L_k @ A
        b = L_k @ b

    # Обратный ход
```

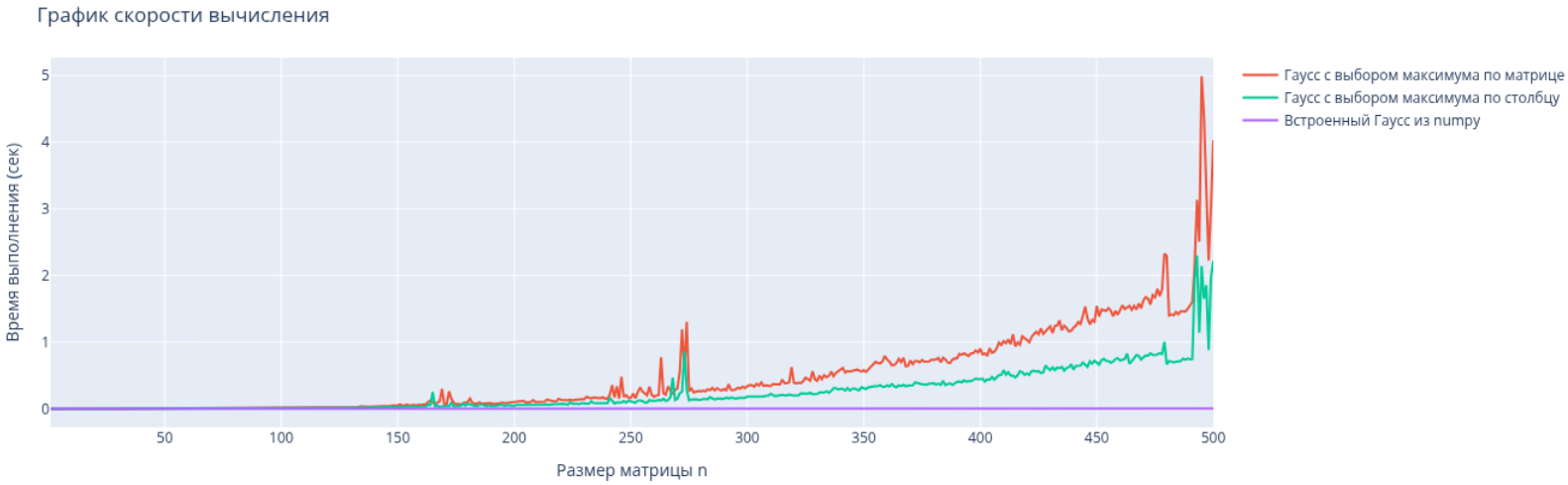
```
33      x = numpy.zeros(n)
34      x[n - 1] = b[n - 1] / A[n - 1, n - 1]
35      for k in range(n - 2, -1, -1):
36          x[k] = (b[k] - (A[k, k + 1:] @ x[k + 1:])) / A[k,
            k]
37
38      return x
```

3.3 Анализ

После реализации прямых методов был проведён анализ по выше описанным критериям. Кроме тех методов, которые были реализованы, к анализу также был добавлен метод Гаусса из библиотеки NumPy, который также в качестве главного элемента использует максимум по столбцу. Для анализа использовались матрицы с размерностями от 1 до 500.

3.3.1 Анализ скорости выполнения

По скорости выполнения методов был получен следующий график:

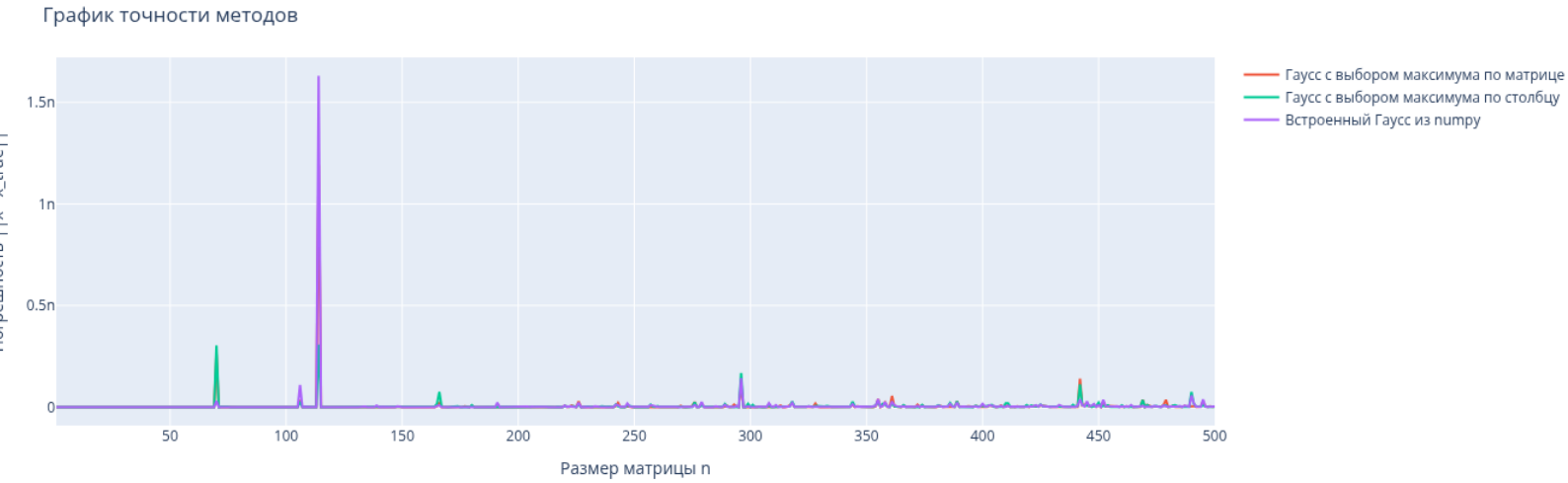


Как можно заметить, метод Гаусса с выбором максимального элемента по матрице является самым долгим, в связи с тем, что в нём необходимо выполнять дополнительные операции: поиск элемента не по столбцу, а по матрице, смена местами не только строк, но и столбцов, необходимость восстановления ответа. Это даёт большие дополнительные нагрузки, в связи чем метод Гаусса с выбором максимального элемента по столбцу явно быстрее.

Однако более интересным может показаться график скорости функции из NumPy, так как его график стремится к оси абсцисс. Столь высокая разница в скорости объясняется тем, что библиотека NumPy написана на языке программирования Си, тогда как реализованные нами методы написаны на Python, что никак не позволит приблизиться к скорости встроенного метода.

3.3.2 Анализ погрешности решения

По погрешности решения был получен следующий график:



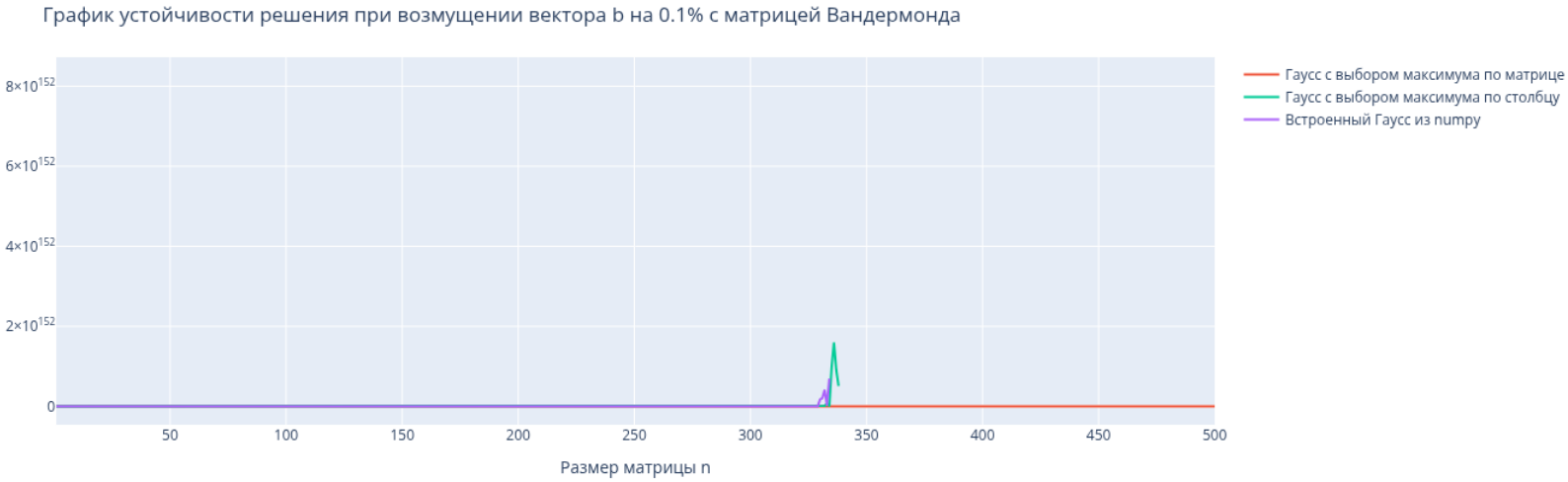
Стоит сразу разъяснить, что буква n в значениях оси ординат означает 10^{-9}, таким образом можно понять, что все решения являются достаточно точными.

В основном точность каждого метода очень схожа, однако в среднем наименее точным является встроенный метод, тогда как явного победителя в этом эксперименте нет, так как всё сильно зависит от исходных данных, где-то лучше метод Гаусса с выбором максимального по столбцу, а где-то – по матрице. Хотя даже так эти значения некритичны, так как они все невероятно малы.

3.3.3 Анализ устойчивости решения

Стоит добавить, что в данном эксперименте для матрицы левой части создавалась матрицы Вандермонда V размера $n \times n$ для n точек $x_i, i = \overline{1, n}$, равноудалённых друг от друга на отрезке $[0; 1]$, а не сгенерированная матрица A. Такое решение было принято, так как эта матрица плохо обусловлена и может дать более показательные результаты.

По устойчивости решения был получен следующий график:



По началу может показаться, что примерно до 330 размерности матрицы число обусловленности близко к нулю, однако это в корне не так. Так может показаться в связи с тем, что значения на оси Oy значительно превышают значения Ox. Поэтому на самом деле на графике везде изображены очень высокие значения.

Примерно до 330 размерности значения числа обусловленности методов примерно равны, однако с 330 размерности начинаются видные колебания графиков метода Гаусса с выбором максимального элемента по столбцу и встроенного метода. Через несколько итераций матриц их графики вовсе прерываются. Это связано с тем, что к тому моменту

решения этих методов начинают расходиться.

Однако можно заметить, что график метода Гаусса с выбором элемента по матрице непрерывный и не имеет таких видных колебаний. Связано это вероятнее всего с более высокой численной устойчивостью метода, так как в нём выбирается в качестве главного самый большой по всей подматрице и риск переполнения при совершении вычислений снижается, нежели при выборе по столбцу.

3.4 Вывод

Исходя из экспериментов, можно прийти к заключению, что для большинства задач лучше подойдёт метод Гаусса с выбором максимального элемента по столбцу, так как она работает достаточно точно и быстрее, чем с выбором по матрице, однако в скорости с встроенным методом из NumPy не сравнится. В учебниках также говорится, что зачастую поиска максимального элемента по столбцу достаточно для точного решения задачи.

Однако если обстоятельства требуют большей численной устойчивости, так как известно, что метод Гаусса с выбором максимального элемента по столбцу может расходиться, лучшим решением будет метод Гаусса с выбором максимального элемента по матрице.

4 Итерационные методы

Для реализации были избраны 2 итерационные метода:

- Метод биортогонализации;

- Метод Якоби.

4.1 Метод биортогонализации

Данный метод заключается в построении двух систем линейно независимых векторов $p_0, p_1, \ldots, p_{n-1}$ и $q_0, q_1, \ldots, q_{n-1}$ таких, что $(Ap_i, q_j) = 0, i \neq j$ и $(Ap_i, q_i) \neq 0$.

Вектора p_0 и q_0 выбираются произвольно, но так, чтобы скалярное произведение было отлично от 0, остальные вектора строятся по рекуррентным соотношениям:

$p_1 = Ap_0 - \gamma_0 p_0 \qquad q_1 = A^T q_0 - \gamma_0 q_0$

$p_{i+1} = Ap_i - \gamma_i p_i - \delta_i p_{i-1} \qquad q_{i+1} = A^T q_i - \gamma_i q_i - \delta_i q_{i-1},$

где $\gamma_i = \dfrac{(Ap_i, A^T q_i)}{(Ap_i, q_i)}, \quad \delta_i = \dfrac{(Ap_i, q_i)}{(Ap_{i-1}, q_{i-1})}$

В случае, если $(p_r, q_r) = 0, r < n$, то, если $p_r = q_r = 0$, то можно достроить систему. Определяются новые векторы $p_0^{(1)}, q_0^{(1)}$, которые строятся по формулам:

$p_0^{(1)} = y - \sum\limits_{i=0}^{r-1} \dfrac{(q_i, Ay)}{(Ap_i, q_i)} p_i$

$q_0^{(1)} = y - \sum\limits_{i=0}^{r-1} \dfrac{(y, Ap_i)}{(Ap_i, q_i)} q_i,$

где y – произвольный вектор.

После аналогично строим остальные вектора, если вектора снова получаются нулевые, то повторяется процесс, пока не будет получено $2n$ векторов. В случае если $(p_r, q_r) = 0$ и при этом хотя бы один из них ненулевой, то выбираются новые p_0 и q_0, далее система строится заново.

После построения системы ответ на решение системы $Ax = b$ получается так:

$$x = \sum_{i=0}^{n-1} \frac{(b, q_i)}{(Ap_i, q_i)} p_i$$

Код:

```
def gamma_i(i: int, A: numpy.array, P: numpy.array, Q: numpy.array) -> float | None:
    return numpy.dot(A @ P[i], A.T @ Q[i]) / numpy.dot(A @ P[i], Q[i])

def delta_i(i: int, A: numpy.array, P: numpy.array, Q: numpy.array) -> float:
    return numpy.dot(A @ P[i], Q[i]) / numpy.dot(A @ P[i - 1], Q[i - 1])

def get_new_p_0_and_q_0(A: numpy.array, P: numpy.array, Q: numpy.array, i: int) -> tuple[numpy.array, numpy.array]:
    while True:
        y = numpy.random.rand(A.shape[0])

        p_0 = y - sum(numpy.dot(Q[j], A @ y) / numpy.dot(A @ P[j], Q[j]) * P[j] for j in range(i))
        q_0 = y - sum(numpy.dot(y, A @ P[j]) / numpy.dot(A @ P[j], Q[j]) * Q[j] for j in range(i))

        if numpy.dot(p_0, q_0) != 0:
            break

    return p_0, q_0

def build_bases(A: numpy.array) -> tuple[numpy.array, numpy.array]:
    n = A.shape[0]
    P = numpy.zeros((n, n))
    Q = numpy.zeros((n, n))

    P[0] = Q[0] = numpy.random.rand(n)

```

```
    if numpy.dot(P[0], Q[0]) == 0:
        return None

    if n == 1:
        return P, Q

    P[1] = A @ P[0] - gamma_i(0, A, P, Q) * P[0]
    Q[1] = A.T @ Q[0] - gamma_i(0, A, P, Q) * Q[0]

    if numpy.dot(P[1], Q[1]) == 0:
        return None

    for i in range(2, n):
        P[i] = A @ P[i - 1] - gamma_i(i - 1, A, P, Q) *
        P[i - 1] - delta_i(i - 1, A, P, Q) * P[i - 2]
        Q[i] = A.T @ Q[i - 1] - gamma_i(i - 1, A, P, Q) *
        Q[i - 1] - delta_i(i - 1, A, P, Q) * Q[i - 2]

        if numpy.dot(P[i], Q[i]) == 0:
            if numpy.linalg.norm(P[i]) == 0 and
            numpy.linalg.norm(Q[i]) == 0:
                P[i], Q[i] = get_new_p_0_and_q_0(A, P, Q,
                i)
            else:
                return None

    return P, Q


def biorthogonalization_method(A: numpy.array, b:
numpy.array) -> numpy.array:
    while True:
        res = build_bases(A)
        if res is not None:
            P, Q = res
            break

```

```
58      x = numpy.zeros(A.shape[0])
59      for i in range(A.shape[0]):
60          x += numpy.dot(b, Q[i]) / numpy.dot(A @ P[i],
        Q[i]) * P[i]
61
62      return x
```

4.2 Метод Якоби

Метод Якоби является итерационным методом решения СЛАУ вида $Ax = b$.

Предполагается, что все диагональные элементы матрицы левой части ненулевые, то есть: $a_{ii} \neq 0, \quad i = \overline{1,n}$.

Идея метода заключается в том, что каждая переменная системы выражается из соответствующего уравнения через остальные неизвестные. Для каждого уравнения системы получаем:

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j \right), \quad i = \overline{1,n}.$$

Далее задаётся начальное приближение $x^{(0)} = (x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)})^T$, чаще всего выбираемое в виде нулевого вектора. Последовательность приближений строится по итерационной формуле:

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

На каждой итерации новые значения всех неизвестных вычисляются исключительно на основе значений предыдущей итерации. Итерационный процесс продолжается до выполнения условия $\|x^{(k+1)} - x^{(k)}\| < \varepsilon$,

где $\varepsilon > 0$ — заданная точность, либо до достижения максимального числа итераций.

Одним из достаточных условий сходимости метода Якоби является диагональное преобладание матрицы A, то есть выполнение неравенств: $|a_{ii}| > \sum\limits_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}|, \quad i = \overline{1, n}.$

При нарушении данного условия метод может расходиться, и последовательность приближений не будет сходиться к решению системы.

Код:

```
def jacobi_method(A: numpy.array, b: numpy.array, eps:
float = 1e-4, max_iter: int = 100) -> numpy.array:
    n = A.shape[0]

    # Проверка ненулевых диагональных элементов
    if numpy.any(numpy.diag(A) == 0):
        raise ValueError("На диагонали матрицы
        присутствует нулевой элемент")

    # Начальное приближение
    x = numpy.zeros(n)
    x_new = numpy.zeros(n)

    for _ in range(max_iter):
        for i in range(n):
            s = 0.0
            for j in range(n):
                if j != i:
                    s += A[i, j] * x[j]

            x_new[i] = (b[i] - s) / A[i, i]

        # Проверка сходимости
        if numpy.linalg.norm(x_new - x) < eps:
```

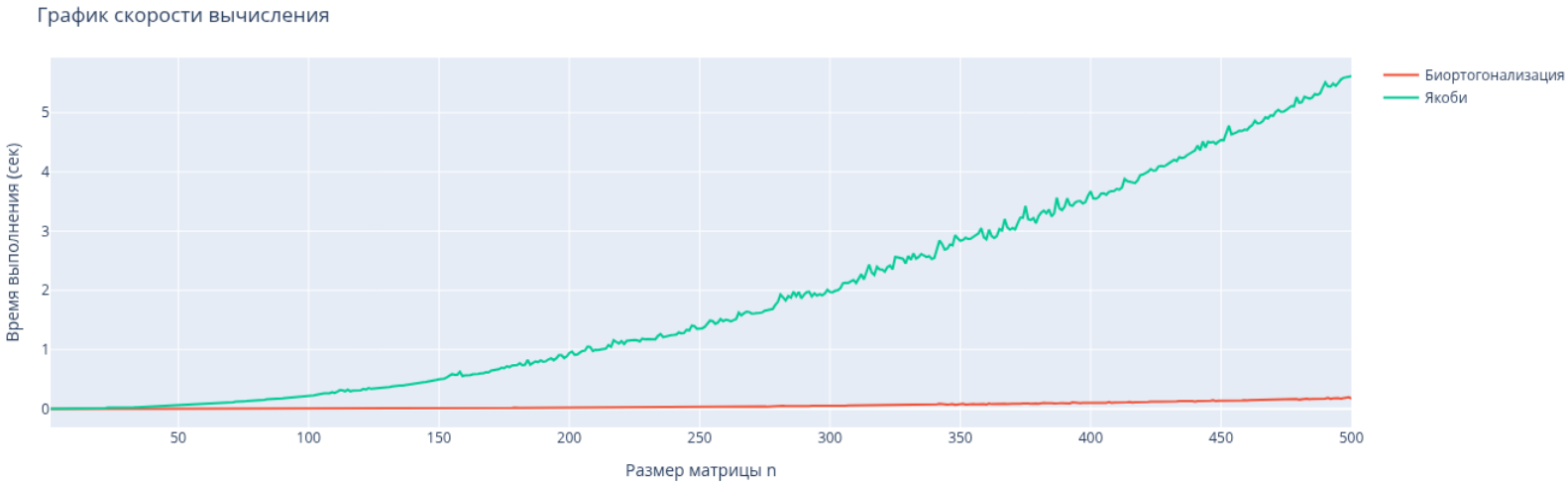
```
            return x_new

        x[:] = x_new

    return x
```

4.3 Анализ

После реализации итерационных методов был проведён анализ по выше описанным критериям. Для анализа использовались матрицы с размерностями от 1 до 500. Также стоит упомянуть, что в связи с ограничением метода Якоби при генерации матрицы левой части A она специально будет генерироваться таким образом, чтобы в ней присутствовало диагональное преобладание.

4.3.1 Анализ скорости выполнение

По скорости выполнения методов был получен следующий график:

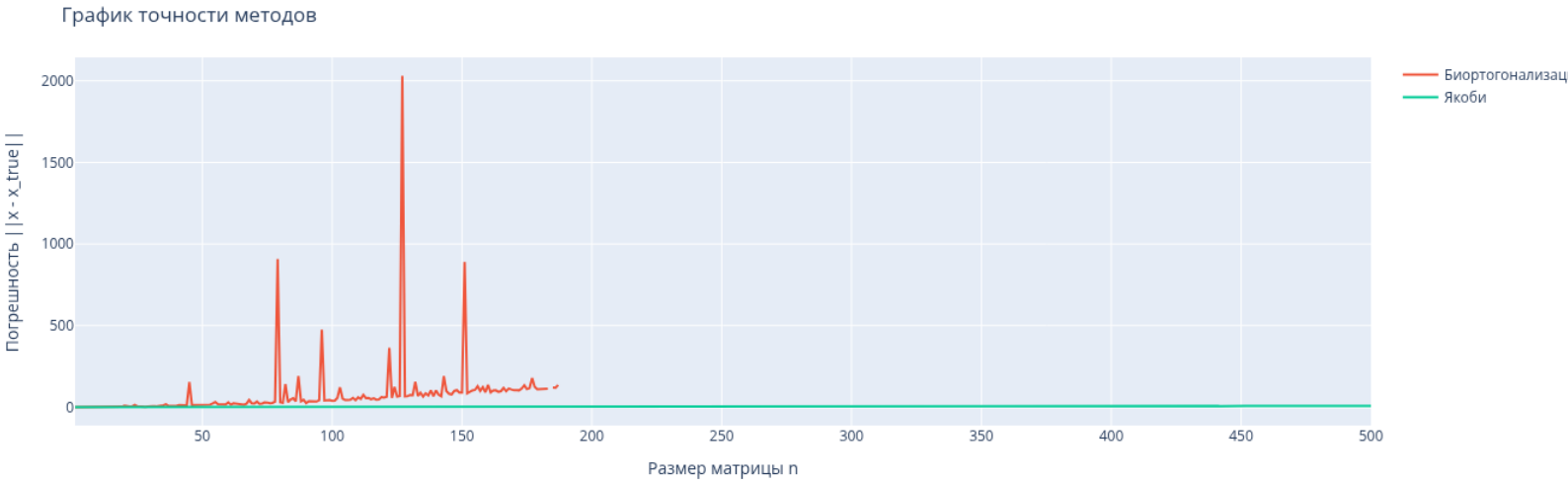


Как можно заметить, скорость выполнения метода биортогонализации значительно выше скорости метода Якоби и близится к нулю. Это

достигается за счёт того, что в методе биортогонализации, если нет необходимости перестроить ортогональную систему, всегда n итераций, тогда как в методе Якоби число итераций ограничивается достижением условия $\|x^{(k+1)} - x^{(k)}\| < \varepsilon$ или указанным максимальным числом итераций.

4.3.2 Анализ погрешности решения

По погрешности решения был получен следующий график:



По графику можно сразу заметить как большие колебания графика метода биортогонализации, так и то, что примерно с 180 итерации график разрывается, так как метод сходится.

При этом метод Якоби достаточно точно находит решение, с учётом того, что для него в коде намеренно указана точность $\varepsilon = 10^{-4}$ и ограничение числа итераций $max\_n = 100$. Результат может быть и более точным при уменьшении значения ε и/или увеличении числа итераций, однако это приводит за собой значительные дополнительные нагрузки и сильно увеличивает время выполнения.

4.3.3 Анализ устойчивости решения

По устойчивости решения был получен следующий график:

Из предыдущего графика можно вспомнить о сходимости метода биортогонализации примерно на 180 итерации, что видно также и здесь. Относительное возмущение решения не так велико, как в эксперименте с прямыми методами, однако это связано с тем, что тут не применяется матрица Вандермонда, так как у него отсутствует диагональное преобладание.

У метода Якоби график достаточно стабильный график, и он также имеет небольшие значения относительного возмущения решения.

4.4 Вывод

В результате экспериментов можно явно увидеть, что метод биортогонализации в основном хорошо себя показывает для матриц примерно до размерности 100, особенно с учётом его высокой скорости выполнения, хотя не без выбросов, однако для более больших матриц он начинает расходиться.

Метод Якоби себя хорошо показывает так, где важна точность, однако он требует продолжительного времени для решения системы, что может не подходить для матриц небольших размерностей, тем более, что метод ограничен возможным числом матриц, для которого будет работать.

5 Задача №8

5.1 Условие задачи

Показать, что существует система уравнений третьего порядка, для которой метод Якоби сходится, а метод Гаусса-Зейделя расходится. (Рассмотреть матрицы симметричные или с диагональным преобладанием)

5.2 Решение

Для примера рассмотрим матрицу:

$$A = \begin{pmatrix} 1 & 0 & 1 \\ -1 & 1 & 0 \\ 1 & 2 & -3 \end{pmatrix}$$

Можем заметить, что она обладает диагональным преобладанием:

$|a_{11}| = 1 \geq |a_{22}| + |a_{33}| = 1$

$|a_{22}| = 1 \geq |a_{11}| + |a_{33}| = 1$

$|a_{33}| = 3 \geq |a_{11}| + |a_{22}| = 3$

Представим $A = D + L + U$:

$$D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -3 \end{pmatrix}, \quad L = \begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 0 \\ 1 & 2 & 0 \end{pmatrix} \quad U = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Метод Якоби:

$B_J = -D^{-1}(L+U)$

$$D^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{3} \end{pmatrix}, \quad L+U = \begin{pmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 1 & 2 & 0 \end{pmatrix}$$

$$B_J = \begin{pmatrix} 0 & 0 & -1 \\ 1 & 0 & 0 \\ \frac{1}{3} & \frac{2}{3} & 0 \end{pmatrix}$$

$\rho(B_J) = \max(|-0.747|, |0.374+i0.867|, |0.374-i0.867|) = |0.374+i0.867| \approx 0.944 < 1 \Rightarrow$ Метод Якоби сходится

Метод Гаусса-Зейделя:

$B_{GS} = -(D+L)^{-1}U$

$$D+L = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 1 & 2 & -3 \end{pmatrix}, \quad (D+L)^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & \frac{2}{3} & -\frac{1}{3} \end{pmatrix}$$

$$B_{GS} = \begin{pmatrix} 0 & 0 & -1 \\ 0 & 0 & -1 \\ 0 & 0 & -1 \end{pmatrix}$$

$\rho(B_{GS}) = \max(|0|, |-1|) = 1 \Rightarrow$ Метод Гаусса-Зейделя может расходится, а может нет.

Сделаем проверку. Возьмём:

$$b = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

Для такой системы $Ax = b$ решением системы является:

$$x = \begin{pmatrix} \frac{1}{3} \\ \frac{4}{3} \\ \frac{2}{3} \end{pmatrix}$$

$$G = (D + L)^{-1}b = \begin{pmatrix} 1 \\ 2 \\ \frac{4}{3} \end{pmatrix}$$

Начнём итерации:

№ итерации k	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
0	0.000 000	0.000 000	0.000 000
1	1.000 000	2.000 000	1.333 333
2	−0.333 333	0.666 667	0.000 000
3	1.000 000	2.000 000	1.333 333
4	−0.333 333	0.666 667	0.000 000
5	1.000 000	2.000 000	1.333 333
6	−0.333 333	0.666 667	0.000 000
7	1.000 000	2.000 000	1.333 333
8	−0.333 333	0.666 667	0.000 000
9	1.000 000	2.000 000	1.333 333
10	−0.333 333	0.666 667	0.000 000

Можно заметить цикличность итераций, что доказывает, что метод Гаусса-Зейделя может расходится.

Ответ: $A = \begin{pmatrix} 1 & 0 & 1 \\ -1 & 1 & 0 \\ 1 & 2 & -3 \end{pmatrix}$