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

ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ (ШКОЛА)

Департамент математического и компьютерного моделирования

ОТЧЕТ

к лабораторной работе №2 по дисциплине

«Вычислительная математика»

Направление подготовки
01.03.02 «Прикладная математика и информатика»

Выполнил студент гр.

Б9121-01.03.02сп (2)

Беляков О.В. _____
(Ф.И.О.) *(подпись)*

г. Владивосток

2023

Содержание

1. Введение

В данной лабораторной работе будут рассмотрены примеры использования методов решения систем линейных алгебраических уравнений, вычисления обратных матриц и определителей и вы числения собственных значений и собственных векторов матриц.

2. Точные методы решения

2.1. Схема Гаусса с выбором главного элемента

2.1.1. Описание метода

Процесс решения системы линейных алгебраических уравнений по методу Гаусса с выбором главного элемента сводится к построению системы с треугольной матрицей, эквивалентной исходной системе.

$$M = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1j} & \dots & a_{1q} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2j} & \dots & a_{2q} & \dots & a_{2n} & b_2 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a_{p1} & a_{p2} & \dots & a_{pj} & \dots & a_{pq} & \dots & a_{pn} & b_p \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nj} & \dots & a_{nq} & \dots & a_{nn} & b_n \end{bmatrix}$$

Выберем наибольший по модулю элемент a_{pq} , не принадлежащий столбцу свободных членов матрицы M. Этот элемент называется главным элементом. Строка матрицы M, содержащая главный элемент, называется главной строкой. Столбец матрицы M. содержащий главный элемент, называется главным столбцом. Далее, производя некоторые операции, построим матрицу $M^{(1)}$ с меньшим на единицу числом строк и столбцов. Матрица $M^{(1)}$ получатся преобразованием из M, при котором главная строка и главный столбец матрицы M исключается. Над матрицей $M^{(1)}$ повторяем те же операции, что и над матрицей M, после чего получаем матрицу $M^{(2)}$, и т.д. Таким образом, мы построим последовательность матриц M, $M^{(1)}$, $M^{(2)}$, ..., $M^{(n-1)}$, последняя из которых представляет собой двухэлементную матрицу - строку; ее также считаем главной строкой. Для получения системы с треугольной матрицей, эквивалентной системе, объединяем все главные строки матриц последовательности, начиная с последней $M^{(n-1)}$

2.1.2. Код

```
import numpy as np


def findPivot(A, p, n):
    max_row = p
    max_val = abs(A[p][p])

    for i in range(p + 1, n):
        if abs(A[i][p]) > max_val:
            max_row = i
            max_val = abs(A[i][p])

    return max_row


def gaussianElimination(A, b):
    n = len(b)
    x = np.zeros(n, dtype=float)

    # ПрямойходметодаГаусса
    for p in range(n - 1):
        pivot_row = findPivot(A, p, n)

        A[[p, pivot_row]] = A[[pivot_row, p]]
        b[[p, pivot_row]] = b[[pivot_row, p]]

        for i in range(p + 1, n):
            factor = A[i][p] / A[p][p]

            for j in range(p, n):
                A[i][j] = A[i][j] - factor * A[p][j]

            b[i] = b[i] - factor * b[p]

    # ОбратныйходметодаГаусса
    x[n - 1] = b[n - 1] / A[n - 1][n - 1]

    for i in range(n - 2, -1, -1):
        sum_val = np.sum(A[i][i + 1:n] * x[i + 1:n])
        x[i] = (b[i] - sum_val) / A[i][i]

    return x


A = np.array([[0.411, 0.421, -0.333, 0.313, -0.141, -0.381, 0.245],
              [0.241, 0.705, 0.139, -0.409, 0.321, 0.0625, 0.101],
              [0.123, -0.239, 0.502, 0.901, 0.243, 0.819, 0.321],
              [0.413, 0.309, 0.801, 0.865, 0.423, 0.118, 0.183],
              [0.241, -0.221, -0.243, 0.134, 1.274, 0.712, 0.423],
              [0.281, 0.525, 0.719, 0.118, -0.974, 0.808, 0.923],
              [0.246, -0.301, 0.231, 0.813, -0.702, 1.223, 1.105]])
```

```
b = np.array([0.096, 1.252, 1.024, 1.023, 1.155, 1.937, 1.673])

x = gaussianElimination(A, b)
print("Решение уравнения: ", x)

result = A @ x

print('Погрешность: ', max(abs(result - b)))
```

2.2. Метод оптимального исключения

2.2.1. Описание метода

Пусть дана система уравнений $Ax = b$. Обозначим b_i через a_{in+1}, преобразуем эту систему к эквивалентной системе более простого вида. Допустим, что $a_{11} \neq 0$. Разделим все коэффициенты первого уравнения системы на a_{11}, который назовем ведущим элементом первого шага, тогда

$$x_1 + a_{12}^{(1)} \cdot x_2 + ... + a_{1n}^{(1)} = a_{1n+1}^{(1)}$$

$$a_{1j}^{(1)} = a_{1j}/a_{11}, j = 2, 3, ..., n + 1$$

Предположим, что после преобразования первых $k(k{\geq}1)$ уравнений система приведена к эквивалентной системе

$$\begin{cases} x_1 + ... + a_{1k+1}^{(k)} x_{k+1} + ... + a_{1n}^{(k)} x_n = a_{1n+1}^{(k)} \\ x_2 + ... + a_{2k+1}^{(k)} x_{k+1} + ... + a_{2n}^{(k)} x_n = a_{2n+1}^{(k)} \\ ... \\ x_k + ... + a_{kk+1}^{(k)} x_{k+1} + ... + a_{kn}^{(k)} x_n = a_{kn+1}^{(k)} \\ a_{k+11} x_1 + a_{k+12} x_2 + ... + a_{k+1n+1} x_k + ... + a_{k+1n} x_n = a_{k+1n+1} \\ ... \\ a_{n1} x_1 + a_{n2} x_2 + ... + a_{nk+1} x_k + ... + a_{nn} x_n = a_{nn+1} \end{cases}$$

Исключим неизвестные $x_1, x_2, ..., x_k$ из $(k + 1)$ уравнения посредством вычитания из него первых k уравнений, умноженных соответственно на числа $a_{k+11}, ..., a_{k+1k}$, и разделив вновь полученное уравнение на коэффициенты при x_{k+1}. Теперь $(k + 1)$ уравнение примет вид:

$$x_{k+1} + a_{k+1k+2}^{(k+1)} \cdot x_{k+2} + ... + a_{k+1n}^{(k+1)} \cdot x_n = a_{k+1n+1}^{k+1}$$

Исключая с помощью этого уравнения неизвестное x_{k+1} из первых k уравнений, получаем опять систему, но с заменой индекса k на $(k+1)$, причем

$$a_{i1}^{(1)} = \frac{a_{1i}}{a_{11}}, i = 2, 3, ..., n+1$$

$$a_{k+1p}^{k+1} = \frac{a_{k+1p} - \sum_{r=1}^{k} a_{rp}^{(k)} a_{k+1r}}{a_{k+1k+1} - \sum_{r=1}^{k} a_{rk+1}^{(k)} a_{k+1r}}$$

$$a_{ip}^{(k+1)} = a_{ip}^{(k)} - a_{k+1p}^{(k+1)} a_{ik+1}^{(k)}, i = 1, ..., k;$$

$$p = k+2, ..., n+1; k = 0, ..., n-1$$

После преобразования всех уравнений находим решение исходной системы.

Метод оптимального исключения работает в случае неравенства нулю ведущих элементов.

2.2.2. Код

```
import numpy as np

A = np.array([[0.411, 0.421, -0.333, 0.313, -0.141, -0.381, 0.245],
              [0.241, 0.705, 0.139, -0.409, 0.321, 0.0625, 0.101],
              [0.123, -0.239, 0.502, 0.901, 0.243, 0.819, 0.321],
              [0.413, 0.309, 0.801, 0.865, 0.423, 0.118, 0.183],
              [0.241, -0.221, -0.243, 0.134, 1.274, 0.712, 0.423],
              [0.281, 0.525, 0.719, 0.118, -0.974, 0.808, 0.923],
              [0.246, -0.301, 0.231, 0.813, -0.702, 1.223, 1.105]])

b = np.array([0.096, 1.252, 1.024, 1.023, 1.155, 1.937, 1.673])


def optimalElimination(matrix, b):
    n = len(matrix)
    A = np.hstack((matrix, b.reshape(-1, 1)))
    A[0, :] /= A[0, 0]

    for k in range(n - 1):
        A_temp = A.copy()
        for p in range(k + 2, n + 1):
            s1 = np.dot(A_temp[:k + 1, p], A_temp[k + 1, :k + 1])
            s2 = np.dot(A_temp[:k + 1, k + 1], A_temp[k + 1, :k + 1])
            A[k + 1, p] = (A_temp[k + 1, p] - s1) / (A_temp[k + 1, k + 1] - s2)
            for i in range(k + 1):
                A[i, p] = A_temp[i, p] - A[k + 1, p] * A_temp[i, k + 1]
        for j in range(k + 1):
            A[k + 1, j] = 0
```

```
            A[j, k + 1] = 0
        A[k + 1, k + 1] = 1
    print(A)
    return A[:, n]


x = optimalElimination(A, b)

result = A @ x

print('Решение: ', x, '\n')

print('Погрешность: ', max(abs(result - b)))
```

2.3. Метод LU разложения

2.3.1. Описание метода

Дана схема $Ax = b$

Представим матрицу A в виде произведения двух матриц B и C

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} = \begin{pmatrix} b_{11} & 0 & \dots & 0 \\ b_{21} & b_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix} \begin{pmatrix} 1 & c_{12} & \dots & c_{1n} \\ 0 & 1 & \dots & c_{2n} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

B — нижнетреугольная матрица

C — верхнетреугольная матрица

Элементы b_{ij}, c_{ij} определяются по формуле:

$$\begin{cases} b_{i1} = a_{i1}, c_{1j} = \frac{a_{1j}}{b_{11}}, \\ b_{ij} = a_{ij} - \sum_{k=1}^{j-1} b_{ik}c_{kj}, c_{ij} = \frac{1}{b_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} b_{ik}c_{kj} \right), \\ (i \geq j > 1), (1 < i < j). \end{cases}$$

У нас получается система $BCx = b$. Представим, что $Cx = y$, для начала найдём , затем у подставим в исходную систему и получим $By = b$ и отсюда уже найдём y

Так как матрицы B и C - треугольные, то

$$y_1 = \frac{a_{1n+1}}{b_{11}};$$

$$y_i = \left(a_{in+1} - \sum_{k=1}^{i-1} b_{ik}y_k \right) / b_{ii}, (i > 1),$$

$$x_n = y_n;$$

$$x_i = y_i - \sum_{k=i+1}^{n} c_{ik}x_k, (i < n).$$

2.3.2. Код

```
import numpy as np

def lu(matrix):
    n = len(matrix)
    L = np.zeros((n, n))
    U = np.zeros((n, n))
    for i in range(n):
        for k in range(i, n):
            sum = 0
            for j in range(i):
                sum += (L[i][j] * U[j][k])
            U[i][k] = matrix[i][k] - sum

        for k in range(i, n):
            if i == k:
                L[i][i] = 1
            else:
                sum = 0
                for j in range(i):
                    sum += (L[k][j] * U[j][i])
                L[k][i] = (matrix[k][i] - sum) / U[i][i]

    return L, U


def solveLu(L, U, b):
    n = len(L)

    y = np.zeros(n)
    for i in range(n):
        y[i] = b[i]
        for j in range(i):
            y[i] -= L[i][j] * y[j]

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

    return x


matrix = np.array([[0.411, 0.421, -0.333, 0.313, -0.141, -0.381, 0.245],
                   [0.241, 0.705, 0.139, -0.409, 0.321, 0.0625, 0.101],
                   [0.123, -0.239, 0.502, 0.901, 0.243, 0.819, 0.321],
                   [0.413, 0.309, 0.801, 0.865, 0.423, 0.118, 0.183],
                   [0.241, -0.221, -0.243, 0.134, 1.274, 0.712, 0.423],
                   [0.281, 0.525, 0.719, 0.118, -0.974, 0.808, 0.923],
                   [0.246, -0.301, 0.231, 0.813, -0.702, 1.223, 1.105]])
```

```
b = np.array([0.096, 1.252, 1.024, 1.023, 1.155, 1.937, 1.673])
L, U = lu(matrix)
x = np.array(solveLu(L, U, b))
print(x, '\n')

print(f'L: {L}', '\n')

print(f'U: {U}', '\n')

print(x, '\n')

result = np.array(matrix @ x)

print('Погрешность: ', max(abs(result - b)))
```

2.4. Метод квадратного корня

2.4.1. Описание метода

Этот метод используется для решения систем, у которых матрица A симметрична. В этом случае матрицу A можно разложить в произведение двух транспонированных друг другу треугольных матриц $A = S^{'} \cdot S$

$$S = \begin{bmatrix} s_{11} & s_{12} & \dots & s_{1n} \\ 0 & s_{22} & \dots & s_{2n} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & s_{nn} \end{bmatrix}$$

Формулы для определения S_{ij}:

$$s_{11} = \sqrt{a_{11}}, s_{1j} = \frac{a_{1j}}{s_{11}}, (j > 1),$$

$$s_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} s_{ki}^2} (i > 1),$$

$$s_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} s_{ki} \cdot s_{kj}}{s_{ii}} (j > i),$$

$$s_{ij} = 0 (i > j).$$

После того, как матрица S найдена, решают систему $S^{'} \cdot y = b$, а затем $S \cdot x = y$. Так как обе системы имеют треугольную форму,то они легко решаются:

$$y_1 = \frac{b_1}{s_{11}}, y_i = \frac{b_i - \sum_{k=1}^{i-1} s_{ki} y_k}{s_{ii}}, (i > 1)$$

$$x_n = \frac{y_n}{s_{11}}, x_i = \frac{y_i - \sum_{k=i+1}^{n} s_{ik} x_k}{s_{ii}}, (i < n)$$

2.4.2. Код

```
import numpy as np
import cmath

A = np.array([[2.2, 4, -3, 1.5, 0.6, 2, 0.7],
              [4, 3.2, 1.5, -0.7, -0.8, 3, 1],
              [-3, 1.5, 1.8, 0.9, 3, 2, 2],
              [1.5, -0.7, 0.9, 2.2, 4, 3, 1],
              [0.6, -0.8, 3, 4, 3.2, 0.6, 0.7],
              [2, 3, 2, 3, 0.6, 2.2, 4],
              [0.7, 1, 2, 1, 0.7, 4, 3.2]])
B = np.array([3.2, 4.3, -0.1, 3.5, 5.3, 9.0, 3.7])

n = len(A)
Y = np.zeros(n, dtype=(object))
X = np.zeros(n, dtype=(object))
S = np.zeros((n, n), dtype=(object))

for i in range(n):
    for j in range(n):

        if (i == 0):
            if (j == 0):
                S[i][j] = cmath.sqrt(A[i][j])
            else:
                if (j > i):
                    S[i][j] = A[0][j] / S[0][0]
        else:
            if (i == j):
                S[i][i] = cmath.sqrt(A[i][i] - sum((S[k][i]) ** 2 for k in range(0, i)))
            else:
                if (j > i):
                    S[i][j] = (A[i][j] - sum(S[k][i] * S[k][j] for k in range(0, i))) / S[i][i]

Y[0] = B[0] / S[0][0]
for i in range(1, n):
    Y[i] = (B[i] - sum(S[k][i] * Y[k] for k in range(0, i))) / S[i][i]

X[n - 1] = Y[n - 1] / S[n - 1][n - 1]

for i in range(n - 2, -1, -1):  # 3,2,1,0
    X[i] = (Y[i] - sum(S[i][k] * X[k] for k in range(i + 1, n))) / S[i][i]

print(S)

print('Решение: ', X, '\n')

print('Погрешность: ', max(abs(A @ X - B)))
```

2.5. Метод окаймления

2.5.1. Описание метода

Запишем матрицу A в виде следующих блоков:

$$A = \begin{pmatrix} A_{n-1} & | & u_n \\ - & - & - \\ v_n & | & a_{nn} \end{pmatrix}$$

где вектор-столбец $u_n = (a_{1n}, ..., a_{(n-1)}n) >$, вектор-строка $v_n = (a_{n1}, ..., a_{n(n-1)})$, а матрица

$$A_{n-1} = \begin{pmatrix} a_{1,1} & a_{1,2} & ... & a_{1,n-1} \\ a_{2,1} & a_{2,2} & ... & a_{2,n-1} \\ ... & ... & ... & ... \\ a_{n-1,1} & a_{n-1,2} & ... & a_{n-1,n-1} \end{pmatrix}$$

Метод окаймления позволяет найти A^{-1}, если известна обратная матрица A_{n-1}^{-1}. Будем искать обратную матрицу в следующем блочном виде:

$$A^{-1} = \begin{pmatrix} P_{n-1} & | & r_n \\ - & - & - \\ q_n & | & \frac{1}{a_n} \end{pmatrix}$$

где P_{n-1} квадратная матрица порядка $n-1$, r_n-вектор-столбец и q_n-вектор-строка размерности $n-1$, a_n-число. По определению обратной матрицы должно выполняться равенство $A \cdot A^{-1} = E$. Применяя правило умножения блочных матриц, запишем последнее равенство в виде:

$$\begin{pmatrix} A_{n-1} \cdot P_{n-1} + u_n \cdot q_n & | & A_{n-1} \cdot r_n + \frac{1}{a_n} u_n \\ - & - & - \\ v_n \cdot P_{n-1} + a_{nn} \cdot q_n & | & v_n \cdot r_n + a_{nn} \cdot \frac{1}{a_n} \end{pmatrix} = \begin{pmatrix} E' & | & 0 \\ - & - & - \\ 0 & | & 1 \end{pmatrix}$$

Приравнивая соответствующие блоки, получаем

$$A_{n-1} \cdot P_{n-1} + u_n \cdot q_n = E^{'}(a) \quad A_{n-1} \cdot r_n + \frac{1}{a_n} u_n = 0(c)$$

$$v_n \cdot P_{n-1} + a_{nn} q_n = 0(b) \quad v_n \cdot r_n + a_{nn} \frac{1}{a_n} = 1(d)$$

Откуда получаем

$$r_n = -\frac{1}{a_n} A_{n-1}^{-1} \cdot u_n$$

$$a_n = a_{nn} - v_n \cdot A_{n-1}^{-1} \cdot u_n$$

$$P_{n-1} = A_{n-1}^{-1} - A_{n-1}^{-1} \cdot u_n \cdot q_n$$

$$q_n = -\frac{1}{a_n} v_n \cdot A_{n-1}^{-1}$$

Теперь обратная матрица A^{-1} представима в следующем виде:

$$A^{-1} = \begin{pmatrix} A_{n-1}^{-1} + \frac{1}{a_n} A_{n-1}^{-1} \cdot u_n \cdot v_n \cdot A_{n-1}^{-1} & | & -\frac{1}{a_n} A_{n-1}^{n-1} \cdot u_n \\ - & - & - \\ -\frac{1}{a_n} v_n \cdot A_{n-1}^{-1} & | & \frac{1}{a_n} \end{pmatrix}$$

Таким образом, если нам известна обратная матрица главного минора $(n{-}1)$ порядка A_{n-1}^{-1}, то можем найти и A^{-1}. Для нахождения A_{n-1}^{-1} нужно знать обратную матрицу A_{n-2}^{-1} главного минора $(n{-}2)$ порядка матрицы A. Проводя аналогичные рассуждения, получим, что для определения A_2^{-1} нужно знать A_1^{-1}. Но $A_1^{-1} = \left(\frac{1}{a_{11}}\right)$

2.5.2. Код

```
import numpy as np

np.set_printoptions(precision=7, suppress=True)


def getInv(A, depth=0):
    n = len(A)
```

```
    k = n - 1

    if n == 1:
        return np.matrix([[1.0 / A[0, 0]]])

    Ap = A[:k, :k]
    V, U = A[k, :k], A[:k, k].reshape(-1, 1)

    Ap_inv = getInv(Ap, depth + 1)

    alpha = 1.0 / (A[k, k] - np.dot(V, np.dot(Ap_inv, U))).item()
    Q = -np.dot(V, Ap_inv) * alpha
    P = Ap_inv - np.dot(np.dot(Ap_inv, U), Q)
    R = -np.dot(Ap_inv, U) * alpha

    AInv = np.zeros((n, n))
    AInv[:k, :k] = P
    AInv[k, :k] = Q
    AInv[:k, k] = R.T
    AInv[k, k] = alpha

    return AInv


a1 = np.matrix([[3, 1, 0],
                [3, 2, 1],
                [1, 1, 1]]).astype(float)
b = np.matrix([2, 1, 1]).astype(float)
n = len(a1)
a_1 = getInv(a1)

print("Обратная матрицаметодомокаймления  : ", a_1)

x = a_1 * b.T

print("Решение: ", x, '\n')

solution = a1 * x

print('Погрешность: ', max(abs(np.ravel(solution) - np.ravel(b))))
```

2.6. Метод отражения

2.6.1. Описание метода

Метод отражений решения системы уравнений $Ax = f$ состоит в выполнении $n-1$ шагов (n - порядок матрицы), в результате чего матрица A системы приводится к верхней треугольной форме, и последующем решении системы с верхней треугольной матрицей. Пусть в результате выполнения $k-1$ шагов матрица A привелась к виду:

$$A_{k-1} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & ... & a_{1k-1}^{(1)} & a_{1k}^{(1)} & a_{1k+1}^{(1)} & ... & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & ... & a_{2k-1}^{(2)} & a_{21}^{(2)} & a_{2k+1}^{(2)} & ... & a_{2n}^{(2)} \\ 0 & 0 & ... & ... & ... & ... & ... & ... \\ 0 & 0 & 0 & a_{k-1k-1}^{(k-1)} & a_{k-1k}^{(k-1)} & a_{k-1k+1}^{(k-1)} & ... & a_{k-1n}^{(k-1)} \\ 0 & 0 & 0 & 0 & a_{kk}^{(k-1)} & a_{kk+1}^{(k-1)} & ... & a_{kn}^{(k-1)} \\ 0 & 0 & 0 & 0 & ... & ... & ... & ... \\ 0 & 0 & 0 & 0 & ... & ... & ... & ... \\ 0 & 0 & 0 & 0 & a_{nk}^{(k-1)} & a_{nk+1}^{(k-1)} & ... & a_{nn}^{(k-1)} \end{bmatrix}$$

Опишем k-й шаг процесса. Цель k-го шага - обнулить все поддиагональные элементы k-го столбца. Для этого определим вектор нормали:

$$p(k) = (0, ..., 0, p_k^{(k)}, p_{k+1}^{(k)}, ..., p_n^{(k)})^T$$

$$p_k^{(k)} = a_{kk}^{(k-1)} + \sigma_k \sqrt{\sum_{l=k}^{n} \left(a_{lk}^{(k-1)}\right)^2}, \sigma_k = \begin{cases} 1, a_{kk}^{(k-1)} \geq 0, \\ -1, a_{kk}^{(k-1)} \leq 0 \end{cases}$$

$$p_l^{(k)} = a_{lk}^{(k-1)}, l = k+1, ..., n.$$

Определим теперь матрицу отражения P_k с элементами:

$$p_{ij}^{(k)} = \sigma_{ij} - 2p_i^{(k)} p_j^{(k)} / \sum_{l=k}^{n} \left(p_l^{(k)}\right)^2$$

σ - символ Кронеккера

Легко проверить, что матрица $A_k = P_k A_{k-1}$ имеет вид:

$$A_{k-1} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & ... & a_{1k-1}^{(1)} & a_{1k}^{(1)} & a_{1k+1}^{(1)} & ... & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & ... & a_{2k-1}^{(2)} & a_{21}^{(2)} & a_{2k+1}^{(2)} & ... & a_{2n}^{(2)} \\ 0 & 0 & ... & ... & ... & ... & ... & ... \\ 0 & 0 & 0 & a_{k-1k-1}^{(k-1)} & a_{k-1k}^{(k-1)} & a_{k-1k+1}^{(k-1)} & ... & a_{k-1n}^{(k-1)} \\ 0 & 0 & 0 & 0 & 0 & a_{kk+1}^{(k-1)} & ... & a_{kn}^{(k-1)} \\ 0 & 0 & 0 & 0 & ... & ... & ... & ... \\ 0 & 0 & 0 & 0 & ... & ... & ... & ... \\ 0 & 0 & 0 & 0 & a_{nk}^{(k-1)} & a_{nk+1}^{(k-1)} & ... & a_{nn}^{(k-1)} \end{bmatrix}$$

т.е. поддиагональные элементы ее k-ого столбца равны нулю, а первые $k-1$ строк и столбцов ее совпадают, с соответствующими строками и столбцами матрицами A_{k-1}. Кроме того, можно показать, что остальные элементы вычисляются по формулам:

$$a_{kk}^{(k)} = -\sigma_k \sqrt{\sum_{l=k}^{n} \left(a_{lk}^{(k-1)}\right)^2}, a_{ij}^{(k)} = a_{ij}^{(k-1)} - 2p_j^{(k)} \frac{\sum_{l=k}^{n} \left(p_l^{(k)} a_{lj}^{(k-1)}\right)}{\sum_{l=k}^{n} \left(p_l^{(k)}\right)^2}$$

$i = k, k+1, ..., n; j = k+1, ..., n$

В результате выполнения всех $n-1$ шагов матрица A приведется к верхней треугольной матрице $A_{n-1} = P_{n-1}P_{n-2}...P_1 A$, которую мы в дальнейшем будем обозначать через $R : R = A_{n-1}$. Обозначив еще $Q = P_1 P_2 ... P_{n-1}$, приходим к равенству $A = QR$, которое удобно использовать для получения решения системы $Ax = f$.

Обратимся теперь к решению системы $Ax = f$. Если мы получили разложение $A = QR$, то для решения этой системы нам, очевидно, достаточно решить систему $Rx = Q \cdot f$ с треугольной матрицей R и правой частью $g = Q \cdot f$.

Решение этой системы находится по простым явным формулам:

$$x_n = g_n / r_{nn}, x_i = \frac{g_i - \sum_{j=i+1}^{n} r_{ij} x_j}{r_{ij}}, i = n-1, n-2, ..., 1.$$

Однако прежде чем находить решение по этим формулам, нам необходимо сначала вычислить правую часть преобразованной системы, т.е. вектор

$$g = P_{n-1} P_{n-2} ... P_1 f$$

Обозначим $f^{(k)} = P_{n-1} P_{n-2} ... P_1 f$. Тогда $f^{(k)} = P_k f^{(k-1)}$. Предположим, что вектор $f^{(k-1)}$ имеет вид: $f^{(k)} = \left(f_1^{(1)}, f_2^{(2)}, f_{k-1}^{(k-1)}, f_k^{(k)}, f_{k+1}^{(k)}, ..., f_n^{(k)} \right)^T$, где элементы $f_i^{(k)}$ вычисляются по формулам:

$$f_i^{(k)} = f_i^{(k-1)} - 2 p_l^{(k)} \frac{\sum_{l=k}^{n} p_l^{(k)} f_l^{(k-1)}}{\sum_{l=k}^{n} \left(p_l^{(k)} \right)^2}, i = k, k+1, ..., n$$

2.6.2. Код

```
import math
import numpy as np

np.set_printoptions(precision=7)

A = np.array([[2.12, 0.48, 1.34, 0.88, 11.172],
              [0.42, 3.95, 1.87, 0.43, 0.115],
              [1.34, 1.87, 2.98, 0.46, 9.009],
              [0.88, 0.43, 0.46, 4.44, 9.349]]).astype(float)
AllMatrix = [A]
print(A)
RMatrix = []
n = len(A)


def vectornormali(a, k):
    pStepkB = np.zeros(k)  # [0]
    pStepkK = np.zeros(n - k)  # [].. size = 2 [0,0]

    a = [row[k] for row in a]
    a = a[k:n]

    for i in range(n - k):  # 0..2  #1...2
        if (i == 0):
            if (a[i] >= 0):
                sigmaK = 1
                pStepkK[0] = a[i] + sigmaK * (math.sqrt(sum(a[l] ** 2 for l in range(i, n - k))))
```

```
            else:
                sigmaK = -1
                pStepkK[0] = a[i] + sigmaK * (math.sqrt(sum(a[l] ** 2 for l in range(i, n - k))))
        else:
            pStepkK[i] = a[i]  # pStepkK[1] = a[1][0]

    pStepk = np.hstack((pStepkB, pStepkK))
    return pStepk


def solveR(aB, k, r, pStepk):
    for i in range(k + 1, n + 1):  # 1,2,3
        aForDot = [row[i] for row in aB]
        for j in range(n):  # 0,1,2
            r[i - 1][j] = 2 * pStepk.dot(aForDot) * pStepk[j] / (sum(pStepk[l] ** 2 for l in range(k
                , n)))
    r = np.transpose(r)

    return r


def solveA(aB, pStepk, k):
    newA = np.zeros((n, n + 1))
    if (k > 0):
        for i in range(0, k):
            newA[i] = aB[i]

    sigma = 1
    newA[k][k] = -sigma * math.sqrt(sum(aB[l][k] ** 2 for l in range(k, n)))

    for i in range(k, n):
        for j in range(k + 1, n + 1):
            newA[i][j] = aB[i][j] - 2 * pStepk[i] * (
                    (sum(pStepk[l] * aB[l][j] for l in range(k, n))) / (sum(pStepk[l] ** 2 for l in
                        range(k, n))))
    return newA


def solveX(g, r):
    x = np.zeros(n)

    x[n - 1] = g[n - 1] / r[n - 1][n - 1]

    for i in range(n - 2, -1, -1):
        x[i] = (g[i] - sum(r[i][j] * x[j] for j in range(i + 1, n))) / r[i][i]

    return x


for k in range(n - 1):
    aB = np.copy(A)
    pStepk = vectornormali(aB, k)
    s = np.linalg.norm((pStepk)) ** 2
```

```
        r = np.zeros((n, n))
        r = solveR(aB, k, r, pStepk)
        RMatrix.append(r)

        A = solveA(aB, pStepk, k)
        AllMatrix.append(A)

A = AllMatrix[n - 1]
R = A[:, :n]
b = [row[-1] for row in A]
x = solveX(b, R)
print("Решение: ", x)

A = AllMatrix[0]
a = A[:, :n]
b = [row[-1] for row in A]
print(np.linalg.solve(a, b))

solution = np.dot(a, x)

print("Погрешность: ", max(abs(solution - b)))
```

2.7. Вывод

2.7.1. Погрешности

Сравним погрешности всех методов для матрицы A и столбца b кроме метода квадратного корня, так как для него нужна симметричная матрица.

```
A = np.array([[0.411, 0.421, −0.333, 0.313, −0.141, −0.381, 0.245],
[0.241, 0.705, 0.139, −0.409, 0.321, 0.0625, 0.101],
[0.123, −0.239, 0.502, 0.901, 0.243, 0.819, 0.321],
[0.413, 0.309, 0.801, 0.865, 0.423, 0.118, 0.183],
[0.241, −0.221, −0.243, 0.134, 1.274, 0.712, 0.423],
[0.281, 0.525, 0.719, 0.118, −0.974, 0.808, 0.923],
[0.246, −0.301, 0.231, 0.813, −0.702, 1.223, 1.105]])

b = np.array([0.096, 1.252, 1.024, 1.023, 1.155, 1.937, 1.673])
```

Под погрешностью подразумевается максимум по модулю от $(Ax–b)$.

1. Погрешность метода Гаусса с выбором главного элемента:
   4.440892098500626e-16

2. Погрешность метода оптимального исключения:
   6.8833827526759706e-15

3. Погрешность метода LU разложения:
   6.661338147750939e-15

4. Погрешность метода окаймления:
   9.992007221626409e-15

5. Погрешность метода отражений:
   4.746394649474949e-15

Самый точный метод оказался метод Гаусса с выбором главного элемента.

Теперь возьмём симметричную матрицу A для всех методов:

```
A = np.array([[2.12, 0.42, 1.34, 0.88],
              [0.42, 3.95, 1.87, 0.43],
              [1.34, 1.87, 2.98, 0.46],
              [0.88, 0.43, 0.46, 4.44]])

b = np.array([11.172, 0.115, 9.009, 9.349])
```

1. Погрешность метода Гаусса с выбором главного элемента:
   4.440892098500626e-16

2. Погрешность метода оптимального исключения:
   1.096345236817342e-15

3. Погрешность метода LU разложения:
   1.096345236817342e-15

4. Погрешность метода квадратного корня:
   1.7763568394002505e-15

5. Погрешность метода окаймления:
   3.552713678800501e-15

6. Погрешность метода отражений:
   1.7763568394002505e-15

Снова самый точный метод оказался метод Гаусса с выбором главного элемента, причём точность для другой матрицы оказалось точно такая же.

2.7.2. Заключение

Проведенные лабораторные работы позволили ознакомиться с различными методами решения систем линейных уравнений.

Метод Гаусса с выбором главного элемента является классическим и эффективным способом решения систем линейных уравнений. Он обеспечивает точные результаты и позволяет избежать проблем с делением на ноль. Однако, его главным недостатком является сложность при работе с большими системами уравнений.

Метод оптимального исключения является усовершенствованным вариантом метода Гаусса. Он позволяет уменьшить количество элементарных операций и ускорить процесс решения системы линейных уравнений.

Метод LU разложения основывается на разложении исходной матрицы на произведение нижней треугольной и верхней треугольной матрицы. Этот метод позволяет использовать разложение для быстрого нахождения обратной или решения новой системы линейных уравнений без необходимости повторного выполнения разложения.

Метод квадратного корня применяется к системам линейных уравнений с симметричной положительно определенной матрицей. Он обеспечивает стабильные и точные результаты, однако требует дополнительных вычислений для выполения квадратного корня и обратного хода.

Метод окаймления является альтернативой методу Гаусса и позволяет решать системы линейных уравнений с большой погрешностью в коэффициентах. Он обеспечивает быстрые результаты, однако может приводить к неточным или неустойчивым решениям.

Метод отражений, также известный как метод Хаусхолдера, используется для нахождения QR разложения матрицы. Он является эффективным способом решения систем линейных уравнений и обеспечивает точные результаты. Однако его основным недостатком является сложность при реализации.

В целом, каждый из рассмотренных методов имеет свои преимущества и недостатки, и выбор конкретного метода зависит от требуемой точности, размера системы и вычислительных возможностей. Однако, все они позволяют решать системы линейных уравнений и являются важными инструментами в научных и инженерных расчетах.

3. Итерационные методы

Пусть дана система n линейных алгебраических уравнений с n неизвестными

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 \\ ... \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n \end{cases} \tag{1}$$

Итерационные методы дают решение системы линейных алгебраических уравнений в виде предела последовательности некоторых векторов, построение которых осуществляется при помощи единообразного процесса, называемого процессом итерации. Для решения системы линейных алгебраических уравнений итерационными методами предварительно приводят систему к виду удобному для итерации:

$$\overline{x} = \overline{\beta} + \alpha\overline{x} \tag{2}$$

Сделать это можно несколькими способами. Например, если в матрице A элементы $|a_{ii}| > \sum\limits_{j \neq i} |a_{ij}|$ для каждого $i = 1, .., n$, то удобно разрешить каждое из уравнений системы относительно диагонального неизвестного, поделив обе части i-го уравнения на a_{ii} и перенеся все члены его, кроме x_i, вправо.

Можно также привести систему (1) к виду (2), если прибавить к обеим частям i-го уравнения x_i, а затем перенести свободные члены влево.

Сходящийся процесс обладает важным свойством самоисправляемости, т.е. отдельная ошибка в вычислениях не отразится на окончательном результате, так как ошибочное приближение можно рассматривать как новый начальный вектор.

Обычно итерации продолжаются до тех пор, пока $||\overline{x}^{(k+1)} - \overline{x}^{(k)}|| \leq \epsilon$, где ϵ - заданная точность.

3.1. Простая итерация

3.1.1. Описание метода

В методе простой итерации в качестве начального приближения берем произвольный вектор $\overline{x}^{(0)}$ и подставляем в правую часть системы, приведенной к виду, удобному для итерации. Получаем некоторый вектор $\overline{x}^{(1)}$. Продолжая этот процесс, получим последовательность векторов:

$$\overline{x}^{(1)} = \overline{\beta} + \alpha\overline{x}^{(0)}$$

$$\overline{x}^{(2)} = \overline{\beta} + \alpha\overline{x}^{(1)}$$

$$\dots$$

$$\overline{x}^{(k)} = \overline{\beta} + \alpha\overline{x}^{(k-1)}$$

Если при $k \to \inf \overline{x}^{(k)} \to \overline{x}$, то вектор $\overline{x}^{(k)}$ будет решением системы (2), т.е. системы (1).

Для сходимости процесса простой итерации достаточно, чтобы какая-либо из норм матрицы α была меньше единицы

$$||\alpha||_\infty = \max_i \sum_{j=1}^{n} |a_{ij}| < 1 \tag{3}$$

$$||\alpha||_1 = \max_j \sum_{i=1}^{n} |a_{ij}| < 1 \tag{4}$$

$$||\alpha||_2 = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} |a_{ij|^2}} < 1 \tag{5}$$

Следствие. Для системы $A\overline{x} = \overline{b}$ метод простой итерации сходится, если

$$|a_{ii|} > \sum_{i \neq j} |a_{ij}| (i = 1, 2, ..., n) \tag{6}$$

Или

$$|a_{jj}| > \sum_{i \neq j} |a_{ij}| (j = 1, 2, ..., n) \tag{7}$$

3.1.2. Код

```
# Методпростойитерации

import numpy as np

np.set_printoptions(linewidth=200)
np.random.seed(123)

n = 4

A = np.array(
    [
        [3.82, 1.02, 0.75, 0.81],
        [1.05, 4.53, 0.98, 1.53],
        [0.73, 0.85, 4.71, 0.81],
        [0.88, 0.81, 1.28, 3.50]
    ]
)

b = np.array([15.655, 22.705, 23.480, 16.110])

mtx_copy = A.copy()
vector_copy = b.copy()

e = 1e-15

steps = 0

for i in range(n):
    b[i] /= A[i, i]
    A[i] /= -A[i, i]
    A[i, i] = 0.0


x = np.zeros(n)
x_old = np.ones(n)

while np.linalg.norm(x-x_old) > e:
    x_old = x.copy()
    # print(np.linalg.norm(mtx_copy.dot(x) - vector_copy, np.inf))
    x = b + A.dot(x)
    steps += 1


print("Initial mtx:\n", mtx_copy)
print("Initial vector:\n", vector_copy, "\n\n")
print("Steps:\n", steps)
print("Answer(x):\n", x)
print("A*x error:\n", abs(mtx_copy.dot(x) - vector_copy))
```

3.2. Релаксация

3.2.1. Описание метода

Приведем систему (1) к виду, удобному для итераций:

$$x = Bx + c.$$

Самый простой способ приведения системы к такому виду состоит в следующем. Из первого уравнения системы (1) выразим неизвестное x_1:

$$x_1 = a_{11}^{-1}(b_1 - a_{12}x_2 - a_{13}x_3 - \ldots - a_{1m}x_m);$$

из второго уравнения – неизвестное x_2 :

$$x_2 = a_{22}^{-1}(b_2 - a_{21}x_1 - a_{23}x_3 - \ldots - a_{2m}x_m);$$

и так далее. Получим систему:

$$\begin{cases} x_1 = 0 + b_{12}x_2 + b_{13}x_3 + \ldots + b_{1,m-1}x_{m-1} + b_{1m}x_m + c_1; \\ x_2 = b_{21}x_1 + 0 + b_{23}x_3 + \ldots + b_{2,m-1}x_{m-1} + b_{2m}x_m + c_2; \\ \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \\ x_m = b_{m1}x_1 + b_{m2}x_2 + b_{m3}x_3 + \ldots + b_{m,m-1}x_{m-1} + 0 + c_m; \end{cases}$$

где $b_{ij} = -a_{ij}/a_{ii}, \quad c_i = b_i/a_{ii}, \quad (i, j = 1, 2, \ldots, m, j \neq i)$.

Суть метода релаксации состоит в следующем. После вычитания очередной i-ой компоненты $(k+1)$-го приближения по формуле метода Зейделя:

$$x_i^{(k+1)} = b_{i1}x_1^{(k+1)} + b_{i2}x_2^{(k+1)} + \ldots + b_{i,i-1}x_{i-1}^{(k+1)} + b_{i,i+1}x_{i+1}^{(k)} + \ldots + b_{im}x_m^{(k)} + c_i,$$

производят дополнительно смещение этой компоненты на величину $(\omega - 1)(x_i^{(k+1)} - x_i^{(k)})$, где ω – параметр релаксации. Таким образом, i-я компонента $(k+1)$-го приближения вычисляется по формуле:

$$x_i^{(k+1)} = x_i^{(k+1)} + (\omega - 1)(x_i^{(k+1)} - x_i^{(k)}) = \omega x_i^{(k+1)} + (1 - \omega)x_i^{(k)}.$$

При $\omega = 1$ метод релаксации совпадает с методом Зейделя. При $\omega > 1$ – метод последовательной верхней релаксации, в противном случае – нижней. Если A – симметричная положительно определенная матрица, то при любом значении параметра $\omega(0 < \omega < 2)$ метод релаксации сходится.

3.2.2. Код

```
# Методрелаксации

import numpy as np

np.set_printoptions(linewidth=200)
np.random.seed(123)
n = 7
#n = 4

A = np.array(
     [
          [10, 0.421, -0.333, 0.313, -0.141, -0.381, 0.245],
          [0.241, 10, 0.139, -0.409, 0.321, 0.0625, 0.101],
          [0.123, -0.239, 10, 0.901, 0.243, 0.819, 0.321],
          [0.413, 0.309, 0.801, 10, 0.423, 0.118, 0.183],
          [0.241, -0.221, -0.243, 0.134, 10, 0.712, 0.423],
          [0.281, 0.525, 0.719, 0.118, -0.974, 10, 0.923],
          [0.246, -0.301, 0.231, 0.813, -0.702, 1.223, 10],
     ]
 )

#A = np.array(
#     [
#          [3.82, 1.02, 0.75, 0.81],
#          [1.05, 4.53, 0.98, 1.53],
#          [0.73, 0.85, 4.71, 0.81],
#          [0.88, 0.81, 1.28, 3.50]
#     ]
#)

b = np.array([0.096, 1.252, 1.024, 1.023, 1.155, 1.937, 1.673])

#b = np.array([15.655, 22.705, 23.480, 16.110])

e = 1e-14
w = 1.8

steps = 0
x = np.zeros(n)
x_old = np.ones(n)

while np.linalg.norm(x - x_old) > e:
    # print(np.linalg.norm(x - x_old) > e)
```

```
    x_old = x.copy()
    for i in range(n):
        x[i] = (1 - w) * x[i] + w * (
                b[i]
                - sum(A[i, j] * x[j] for j in range(0, i))
                - sum(A[i, j] * x[j] for j in range(i + 1, n))
        ) / A[i, i]
    steps += 1

ans = x

print("Initial mtx:\n", A)
print("Initial vector:\n", b)
print("Steps:\n", steps, "\n\n")
print("Answer(x):\n", ans)
print("A*x error:\n", abs(A.dot(ans) - b))
```

3.3. Градиентный спуск

3.3.1. Описание метода

Пусть дана система с симметричной положительно определённой матрицей:

$$Ax = b, \quad A = A^T, \quad A > 0.$$

Введем квадратичный функционал:

$$F(x, x) = (Ax, x) - 2(b, x).$$

Пусть x^* – решение системы (1). Тогда:

$$F(x, x) = (Ax, x) - (Ax^*, x) - (Ax^*, x) = (A(x - x^*), x - x^*) - (Ax^*, x^*);$$

$$F(x, x) = ||x - x^*||_A^2 - ||x^*||_A^2; \tag{8}$$

где $||\cdot||$ – энергетическая норма, которая порождается симметричной положительно определенной матрицей A. Она определяется следующим образом:

$$||x||_A = \sqrt{(Ax, x)}.$$

Таким образом из представления F в виде (8) решением исходной системы является минимум функционала F. Минимум находится итерационным процессом:

$$x^{(k+1)} = x^{(k)} - \alpha_k \nabla F\left(x^{(k)}\right),$$

где ∇F – градиент, а α_n – некоторый параметр. Значения градиента:

$$\frac{\partial}{\partial x_m}\left(\sum_{i,j=1}^{n} a_{ij}x_i x_j - 2\sum_{i=1}^{n} b_i x_i\right) = 2\left(\sum_{j=1}^{n} a_{mj-b_m}\right);$$

$$\nabla F(x) = 2(Ax - b);$$

$$x^{(k+1)} = x^{(k)} - \delta_k(Ax^{(k)} - b), \quad \delta_k = 2\alpha_k.$$

Параметр δ_k выбирается так, чтобы $F(x)$ был минимален. Решение методом градиентного спуска строится следующим образом:

$$x^{(k+1)} = x^{(k)} - \delta_k \underbrace{(Ax^{(k)} - b)}_{\zeta^k}, \quad \delta = \frac{||\zeta^k||^2}{||\zeta^k||_A^2}.$$

3.3.2. Код

```
# Градиентныйспуск

import numpy as np

np.set_printoptions(linewidth=200)
np.random.seed(123)

n = 4

A = np.array(
    [
        [3.82, 1.02, 0.75, 0.81],
        [1.05, 4.53, 0.98, 1.53],
        [0.73, 0.85, 4.71, 0.81],
        [0.88, 0.81, 1.28, 3.50]
    ]
)

# b = np.array([0.096, 1.252, 1.024, 1.023, 1.155, 1.937, 1.673])

b = np.array([15.655, 22.705, 23.480, 16.110])

tolerance = 1e-14

X = np.zeros(n)
r = b - A.dot(X)
a = np.dot(r, r) / np.dot(A.dot(r), r)
steps = 0

while np.linalg.norm(A.dot(X) - b) > tolerance:
    # print(np.linalg.norm(A.dot(X) - b))
    X = X + a * r
    r = r - a * A.dot(r)
    a = np.dot(r, r) / np.dot(A.dot(r), r)
    steps += 1

ans = X

print("Answer(x):\n", ans)
print("Steps:", steps, "\n\n")
print("A*x error:\n", abs(A.dot(ans) - b), "\n\n")
```

3.4. Ричардсон

3.4.1. Описание метода

Рассмотрим систему (1). Она может быть преобразована к эквивалентной системе:

$$x = Bx + c,$$

чтобы к ней применить итерационный метод, прибавим к исходной системе и вычтем из нее Dx, где D– невырожденная матрица:

$$Dx = (D - A)x + b;$$

$$x = D^{-1}(D - A)x + D^{-1}b = \underbrace{(I - D^{-1}A)x}_{=B} + \underbrace{D^{-1}b}_{=c}.$$

Рассматриваем D такую, что $D^{-1} = \tau I$:

$$x = (I - \tau A)x + \tau b. \tag{9}$$

В данном виде к системе легко применим метод простой итерации:

$$x^{(k+1)} = (I - \tau A)x^{(k)} + \tau b, \quad \frac{x^{(k+1)} - x^{(k)}}{\tau} + Ax^{(k)} = b. \tag{10}$$

Если в данном методе на каждом шаге рассматривать разные τ_n, то можно ускорить процесс. Для этого рассматривается процесс, обобщающий метод простой итерации.

Для решения системы линейных уравнений (1) с симметричной положительно определенной матрицы A применим метод Ричардсона. Итерационный процесс которого имеет вид:

$$x^{(k+1)} = x^{(k)} - \tau_{k+1}\left(Ax^{(k)} - b\right), \quad \frac{x^{(k+1)} - x^{(k)}}{\tau_{k+1}} + Ax^{(k)} = b. \tag{11}$$

Если λ – собственное значение A, то $(1 - \lambda\tau)$– собственное значение $(I - \tau A.)$ Требуется, чтобы $|1 - \tau\lambda| < 1$. Поэтому метод (10) сходится, если

$\tau \in (0, \frac{2}{M})$, где M– максимальное собственное значение матрицы A. Пусть μ – минимальное собственное значение. В качестве оптимальных параметров сходимости для минимизации погрешности рассматриваются следующие значения, полученные из свойств многочлена Чебышева:

$$\tau_0 = \frac{2}{M+\mu}, \quad \tau_k = \frac{\tau_0}{1+\rho\upsilon_k};$$

$$\rho = \frac{1-\frac{\mu}{M}}{1+\frac{\mu}{M}}, \quad \upsilon_k = \cos\frac{(2k-1)\pi}{2N}, \quad k = 1, 2, \ldots, N.$$

3.4.2. Код

```
# Ричардсон
import math

import numpy as np

class Rotation:
    def __init__(self, n: int, mtx: np.ndarray, p: int):
        self.k = 0
        self.n = n
        self.mtx = mtx
        self.p = p

    def sign(self, num):
        if num > 0:
            return 1
        return -1

    def index_of_largest(self, matrix):
        largest = abs(matrix[0][1])
        i_1, j_1 = 0, 1
        for i in range(self.n):
            for j in range(self.n):
                if i == j:
                    continue
                if abs(matrix[i, j]) > largest:
                    largest = abs(matrix[i, j])
                    i_1, j_1 = i, j
        return i_1, j_1

    def check_tol(self, matrix):

        # Преграда
        self.tol = np.sqrt(max(abs(np.diag(matrix)))) * (10 ** (-self.k))

        #Проверка внедиагональныхэлементов
        for i in range(self.n):
```

```
            for j in range(self.n):
                if i == j:
                    continue
                if abs(matrix[i, j]) >= self.tol:
                    return False

        if self.k < self.p:
            self.k += 1
            return False

        return True

    def solve(self):
        mtx = self.mtx
        steps = 0

        while not self.check_tol(mtx):
            q, p = self.index_of_largest(mtx)

            d = abs(mtx[p, p] - mtx[q, q]) / np.sqrt(
                (mtx[p, p] - mtx[q, q]) ** 2 + 4 * mtx[p, q] ** 2
            )

            c = np.sqrt(0.5 * (1 + d))
            s = self.sign(mtx[p, q] * (mtx[p, p] - mtx[q, q])) * np.sqrt(0.5 * (1 - d))

            R = np.eye(self.n)
            R[p, p] = R[q, q] = c
            R[p, q] = -s
            R[q, p] = s
            mtx = R.T.dot(mtx).dot(R)
            mtx[p, q] = mtx[q, p] = 0
            steps += 1

        return steps, np.diag(mtx)

def getMaxError(mat, x, vector):
    errors = mat.dot(x) - vector
    return np.max(np.abs(errors))


n = 2
mtx = np.array([[2,1],[1,2]])
vector = np.array([4, 5])

e = 1e-11

n = mtx.shape[0]
ans = np.zeros(n)
eigs = Rotation(n, mtx, 20).solve()[1]

lmin = np.min(eigs)
lmax = np.max(eigs)
```

```
tau0 = 2 / (lmin + lmax)
nnn = lmin / lmax

p0 = (1 - nnn) / (1 + nnn)
p1 = (1 - math.sqrt(nnn)) / (1 + math.sqrt(nnn))

maxiters = np.log(2 / e) / np.log(1 / p1)
print(maxiters)

steps = 0
tau = tau0
iters = round(maxiters) + 1

while steps < iters:
    steps += 1
    v = np.cos(2 * steps - 1) * np.pi / (2 * maxiters)
    tau = tau0 / (1 + p0 * v)
    ans -= tau * (mtx.dot(ans) - vector)

    if steps == iters:
        if getMaxError(mtx, ans, vector) > e:
            iters += (round(maxiters)+1)


print("Steps:\n", steps, "\n\n")
print("Answer(x):\n", ans, "\n\n")
print("A*x error:\n", abs(mtx.dot(ans) - vector), "\n\n")
```

3.5. Вывод

3.5.1. Количество итераций

Входные данные:

```
A = np.array([
    [28, 5, 3, 9, 2],
    [4, 30, 2, 7, 8],
    [1, 4, 22, 9, 4],
    [1, 7, 3, 21, 1],
    [1, 9, 8, 7, 31],
])

A = np.array([
    [72, 9, 8, 5, 1, 1, 2, 5, 6, 1, 7, 1, 3, 9, 9],
    [0, 54, 6, 7, 3, 0, 3, 1, 2, 4, 9, 6, 4, 7, 0],
    [0, 3, 50, 2, 9, 7, 0, 2, 0, 4, 4, 1, 6, 4, 4],
    [8, 2, 0, 70, 4, 4, 3, 9, 6, 0, 7, 5, 6, 7, 6],
    [9, 3, 3, 4, 72, 3, 3, 8, 8, 0, 1, 6, 4, 2, 9],
    [3, 9, 0, 9, 7, 90, 8, 4, 7, 5, 7, 9, 4, 8, 2],
    [3, 5, 5, 9, 8, 7, 74, 5, 5, 9, 0, 5, 3, 0, 2],
    [4, 9, 1, 2, 9, 2, 2, 73, 3, 7, 4, 9, 4, 4, 5],
    [4, 9, 3, 3, 3, 7, 9, 6, 69, 8, 4, 0, 2, 1, 7],
    [2, 5, 4, 1, 8, 7, 5, 3, 5, 61, 4, 3, 6, 0, 1],
    [8, 9, 8, 0, 0, 6, 5, 6, 6, 1, 64, 0, 3, 4, 3],
    [0, 1, 7, 3, 1, 3, 6, 4, 5, 1, 3, 40, 0, 4, 0],
    [0, 4, 6, 7, 2, 7, 5, 1, 1, 1, 6, 6, 64, 8, 0],
    [6, 0, 8, 3, 4, 4, 7, 8, 8, 0, 3, 0, 5, 57, 0],
    [2, 5, 7, 0, 0, 5, 0, 0, 1, 6, 9, 7, 5, 5, 61],
])
```

Количество итераций с заданной точностью e^{-10}: $n = 5$:

1. Метод Зейделя $\omega = 1$:

   11

2. Верхняя релаксация $\omega = 1.25$:

   18

3. Нижняя релаксация $\omega = 0.75$:

   26

4. Простая итерация:

   55

5. Метод Ричардсона:

   30

$n = 15$:

1. Метод Зейделя $\omega = 1$:

   14

2. Верхняя релаксация $\omega = 1.25$:

   24

3. Нижняя релаксация $\omega = 0.75$:

   20

4. Простая итерация:

   215

5. Метод Ричардсона:

   67

3.5.2. Заключение

Из результатов результатов использования различных численных итерационных методов для решения систем линейных алгебраических уравнений можно сделать следующие выводы:

1. Простая итерация:

   - Для всех значений n метод простой итерации показал сравнительно высокое количество итераций.

2. Метод Релаксаций:

   - Метод Зейделя (Метод релаксации совпадает с ним при $\omega = 1$) показал наилучшие результаты. Для методов верхней и нижней релаксаций потребовалось больше итераций для решения.

3. Метод Ричардсона:

    - При увеличении размера системы количество итераций метода Ричардсона увеличивается более медленно, чем у метода простой итерации. В целом, он все еще уступает методам релаксаций по количеству итераций.

4. Градиентный спуск:

    - Количество итераций выше, чем у метода релаксации, но метод все равно показывает неплохие результаты.

Таким образом, методы релаксаций оказались более предпочтительными по сравнению с другими методами, благодаря их сходимости и точности решения при относительно небольшом количестве итераций.

4. Частичная проблема собственных значений

4.1. Вращения с преградами

4.1.1. Описание метода

Элементарный шаг каждого эрмитова процесса заключается в преобразовании подобия посредством матрицы вращения:

$$T_{ij} = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & c & \dots & \dots & \dots & -s & \\ & & & \ddots & & & & \\ & & & & 1 & & & \\ & & & & & \ddots & & \\ & & s & \dots & \dots & \dots & c & \\ & & & & & & & \ddots \\ & & & & & & & & 1 \end{bmatrix};$$

при $c^2 + s^2 = 1$. Эти матрицы принадлежат к классу ортогональных матриц, т.е $T_{ij} \cdot T_{ij}^* = E$.

Процесс состоит в построении последовательности матриц $A_0 = A, A_1, \dots,$ каждая из которых получается из предыдущей с помощью элементарного шага. Эти элементарные шаги должны быть подобраны так, чтобы $A^{(n+1)}$ безгранично приближалась к диагональной матрице при $m \to \infty$. Дадим расчетные формулы $(m+1)$ шага (при котором $A^{(m+1)} = T_{ij}^* A^m T_{ij}$).

Для удобства введем обозначения $C = A^{m+1}$, тогда:

$$C_{kl} = a_{kl}^{(m)}, \quad k \neq i, k \neq j, l \neq i, l \neq j;$$

$$C_{ki} = C_{ik} = ca_{ki}^{(m)} + sa_{kj}^{(m)}, \quad k \neq i, k \neq j;$$

$$C_{kj} = C_{jk} = -sa_{ki}^{(m)} + ca_{kj}^{(m)}, \quad k \neq i, k \neq j;$$

$C_{ii} = c^2 a_{ii}^{(m)} + 2csa_{ij}^{(m)} + s^2 a_{jj}^{(m)}, C_{jj} = s^2 a_{ii}^{(m)} - 2csa_{ij}^{(m)} + c^2 a_{jj}^{(m)};$

$C_{ij} = C_{ji} = 0.$

Числа c, s определяются по формулам:

$$c = \sqrt{\frac{1}{2}\left(1 + \frac{|a_{ii}^{(m)} - a_{jj}^{(m)}|}{d}\right)};$$

$$s = sgn[a_{ij}^{(m)}(a_{ii}^{(m)} - a_{jj}^{(m)})]\sqrt{\frac{1}{2}\left(1 - \frac{|a_{ii}^{(m)} - a_{jj}^{(m)}|}{d}\right)}, d = \sqrt{(a_{ii}^{(m)} - a_{jj}^{(m)})^2 + 4a_{ij}^{(m)2}};$$

Матрица вращения выбирается на $(m+1)$ шаге так, чтобы элемент a_{ij} стал нулем. При этом пара индексов (ij) выбирается так, чтобы аннулировался наибольший по модулю внедиагональный элемент матрицы $A^{(m)}$, а именно $|a_{ij}^{(m)}| \geq \sigma_k$, где $\sigma_1, \sigma_2, \ldots$ монотонно убывающая к нулю последовательность чисел, называемых "преградами". Один из способов задания "преград"состоит в нахождении σ_k по формуле $\sigma_k = \sqrt{max|a_{ii}^{(m)}|} \cdot 10^{(-k)}$, где $k = 1, 2, \ldots, p$. Число p зависит от разрядности машины и требуемой точности решения поставленной задачи. После того как все внедиагональные элементы станут по модулю не больше σ_k, то "преграда" σ_k заменяется на σ_{k+1} и т. д., $k = 1, 2, \ldots, p$. Процесс заканчивается, когда все внедиагональные элементы станут меньше по модулю σ_p. Известно, что процесс с "преградами"сходится.

Так как характеристические полиномы подобных матриц совпадают, следовательно:

$$det(A - \lambda E) = det(D - \lambda E) = (d_{11} - \lambda)(d_{22} - \lambda)\ldots(d_{nn} - \lambda) = 0,$$

где D– есть диагональная матрица, полученная в результате выполнения описанного выше итерационного процесса.

Скажем несколько слов о нахождении собственных векторов матрицы A. Пусть итерационный процесс, описанный выше, доведен до того, что матрица:

$$D = \prod_m T'_{i_m i_m} \cdot A \cdot \prod_m T_{i_m j_m},$$

оказалась практически диагональной. Тогда столбцы матрицы $\prod_m T_{i_m j_m}$ будут собственными векторами исходной матрицы A. Домножив матрицу слева на $W = \prod_m T_{i_m j_m}$ и справа на $W^{'}$, получим $A = WDW^{'}$. Из этого равенства получаем, что $AW = WD$. Если расписать это равенство по столбцам, то окажется, что каждый $i-$ый столбец матрицы W является собственным вектором, соответствующим собственном значению λ_i. В отличие от прямых методов, алгоритмы которых состоят из разнородных частей: преобразования исходной матрицы, вычисления корней многочлена, нахождения собственных векторов, метод вращения позволяет в результате выполнения итерационного процесса найти собственные значения и собственные вектора.

Хотя количество умножений в этом методе весьма значительно, ошибки округления накапливаются медленно, так как умножения происходит на коэффициенты c и s по модулю меньше единицы.

4.1.2. Код

```
# Вращенияспреградами

import numpy as np
import sympy as sp
A = np.array([[2.2, 1, 0.5, 2],
              [1, 1.3, 2, 1],
              [0.5, 2, 0.5, 1.6],
              [2, 1, 1.6, 2]])
n, _ = A.shape
eps = 1e-10
x_col = np.array([sp.Symbol(f"x_{i}") for i in range(n)])
Full = np.hstack((A, x_col[np.newaxis].T))

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

Full_s = [[isinstance(Full[i, j], sp.Expr) for j in range(n + 1)]
    for i in range(n)]

x_next = np.ones(n)
alpha_next = 1
count = 0
diff = 1000000000000
symbols = [[x_col[i], 0] for i in range(n)]
```

```
while diff > eps and count < 100000:
    count += 1
    x_pre = x_next
    alpha_pre = alpha_next
    f_pre = x_pre / alpha_pre
    for i in range(n):
        symbols[i][1] = f_pre[i]
    x_next[n - 1] = Full[n - 1, n].subs(symbols) / Full[n - 1, n - 1]
    for i in range(n - 2, -1, -1):
        x_next[i] = (Full[i, n].subs(symbols) - sum([Full[i, k] * x_next[k] for k in range(i + 1, n)
            ])) / Full[i, i]
    max_x = 0
    for i in range(n):
        if abs(max_x) < abs(x_next[i]):
            max_x = x_next[i]
    alpha_next = max_x
    diff = abs((1 / alpha_pre) - (1 / alpha_next))
lamda = 1 / alpha_next
print("Count: ", count)
validate_lamda = abs(np.linalg.det(A - lamda * np.eye(n)))
print("Diff:", diff)
```

4.2. Простая итерация

4.2.1. Описание метода

Построим итерационный процесс, применяя метод простой итерации к решению системы уравнений:

$$\lambda\overline{x} = A\overline{x}.$$

Запишем эту систему, введя вспомогательный вектор y:

$$\overline{y} = A\overline{x}, \tag{12}$$

$$\lambda\overline{x} = \overline{y}. \tag{13}$$

Пусть $\overline{x}^{(0)}$ – начальное приближение собственного вектора $\overline{x}$, причем собственные векторы на каждой итерации нормированы. Вычислим $\overline{y}^{(1)}$:

$$\overline{y}^{(1)} = A\overline{x}^{(0)}.$$

Соотношение (13) используем для вычисления первого приближения $\lambda^{(1)}$, применяя умножение обеих частей равенства скалярно на $\overline{x}^{(0)}$:

$$\lambda = \frac{\overline{y} \cdot \overline{x}^{(0)}}{\overline{x}^{(0)} \cdot \overline{x}^{(0)}} = \overline{y}^{(1)} \cdot \overline{x}^{(0)}.$$

Здесь учтено, что вектор $\overline{x}^{(0)}$ нормирован. Следующее приближение собственного вектора $\overline{x}^{(1)}$ можно вычислить, нормируя вектор $\overline{y}^{(1)}$. Окончательно итерационный процесс записывается в виде:

$$\overline{y}^{(k+1)} = A\overline{x}^{(k)},$$

$$\lambda^{(k+1)} = \overline{y}^{(k+1)} \cdot \overline{x}^{(k)},$$

$$\overline{x}^{(k+1)} = \frac{\overline{y}^{(k+1)}}{|\overline{y}^{(k+1)}|}, \quad \ldots k = 0, 1, \ldots,$$

и продолжается до установления постоянных значений λ и $\overline{x}$. Найденное в результате итерационного процесса число λ является наибольшим по модулю собственным значением данной матрицы A, а $\overline{x}$– соответствующим ему собственным вектором.

4.2.2. Код

```
#Простая итерация
import numpy as np

A = np.array([[2.2, 1, 0.5, 2],
              [1, 1.3, 2, 1],
              [0.5, 2, 0.5, 1.6],
              [2, 1, 1.6, 2]])
n, _ = A.shape
eps = 1e-10
x_next = np.ones(n)
count = 0
diff = eps + 100

while diff > eps:
    count += 1
    x_prev = x_next
    y = np.matmul(A, x_prev)
    lamda = np.dot(y, x_prev)
    x_next = y / np.linalg.norm(y)
    sign = 1 if lamda > 0 else -1
    diff = abs(sign * x_next - x_prev).max()

print("Count: ", count)
print("Diff:", diff)
```

4.3. Обратная итерация

4.3.1. Описание метода

Предположим, что матрица A имеет только вещественные различные по модулю собственные значения. Пронумеруем их в порядке убывания модулей:

$$|\lambda_1| > |\lambda_2| > \dots |\lambda_n| > 0.$$

Метод обратных итераций предназначен для вычисления наименьшего по модулю собственного значения λ_n и отвечающего ему собственного вектора $u^{(n)}$ и состоит в следующем. Выберем произвольный вектор $x^{(0)}$ и построим последовательность векторов $x^{(k)}$, каждый из которых является решением системы уравнений:

$$Ax^{(k)} = x^{(k-1)}/\alpha_{k-1}, \quad k = 1, 2, \dots,$$

где α_{k-1}– наибольшая по модулю компонента вектора $x^{(k-1)}$, то есть:

$$|\alpha_{k-1} = \max_{1 \le i \le n} |x_i^{(k-1)}|.$$

Известно, что при $k \to \infty \quad 1/\alpha_k = \lambda_n, \quad x^{(k)} \to u^{(n)}$.

На практике вычисления продолжают до тех пор, пока не будет выполнено неравенство:

$$|1/\alpha_k - 1/\alpha_{k-1}| < \varepsilon,$$

где ε– заданная точность вычисления собственного значения λ_n. При этом $1/\alpha_k$ принимают за приближенное значение λ_n, а $x^{(k)}$– за приближение к $u^{(n)}$. Если за K итераций заданная точность не достигается, вычисления прекращаются.

Для решения систем уравнений $Ax^{(k)} = x^{(k-1)}/\alpha_{k-1}$ на каждом шаге можно воспользоваться методом Гаусса. Поскольку матрица у всех систем одна и та же, то ее треугольное разложение $U + MA$ следует выполнить только один раз.

Решение каждой системы $Ax^{(k)} = x^{(k-1)}/\alpha_{k-1}$ сводится, следовательно, к получению преобразованной правой части $g^{(k)} = M(x^{k-1}/\alpha_{k-1})$ и последующему решению системы с треугольной матрицей:

$$Ux^{(k)} = g^{(k)}.$$

4.3.2. Код

```
#Обратная итерация
import numpy as np
import sympy as sp

A = np.array([
    [72, 9, 8, 5, 1, 1, 2, 5, 6, 1, 7, 1, 3, 9, 9],
    [0, 54, 6, 7, 3, 0, 3, 1, 2, 4, 9, 6, 4, 7, 0],
    [0, 3, 50, 2, 9, 7, 0, 2, 0, 4, 4, 1, 6, 4, 4],
    [8, 2, 0, 70, 4, 4, 3, 9, 6, 0, 7, 5, 6, 7, 6],
    [9, 3, 3, 4, 72, 3, 3, 8, 8, 0, 1, 6, 4, 2, 9],
    [3, 9, 0, 9, 7, 90, 8, 4, 7, 5, 7, 9, 4, 8, 2],
    [3, 5, 5, 9, 8, 7, 74, 5, 5, 9, 0, 5, 3, 0, 2],
    [4, 9, 1, 2, 9, 2, 2, 73, 3, 7, 4, 9, 4, 4, 5],
    [4, 9, 3, 3, 3, 7, 9, 6, 69, 8, 4, 0, 2, 1, 7],
    [2, 5, 4, 1, 8, 7, 5, 3, 5, 61, 4, 3, 6, 0, 1],
    [8, 9, 8, 0, 0, 6, 5, 6, 6, 1, 64, 0, 3, 4, 3],
    [0, 1, 7, 3, 1, 3, 6, 4, 5, 1, 3, 40, 0, 4, 0],
    [0, 4, 6, 7, 2, 7, 5, 1, 1, 1, 6, 6, 64, 8, 0],
    [6, 0, 8, 3, 4, 4, 7, 8, 8, 0, 3, 0, 5, 57, 0],
    [2, 5, 7, 0, 0, 5, 0, 0, 1, 6, 9, 7, 5, 5, 61],
])

n, _ = A.shape
eps = 1e-10
x_col = np.array([sp.Symbol(f"x_{i}") for i in range(n)])
Full = np.hstack((A, x_col[np.newaxis].T))

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

Full_s = [[isinstance(Full[i, j], sp.Expr) for j in range(n + 1)]
    for i in range(n)]

x_next = np.ones(n)
alpha_next = 1
count = 0
diff = 1000000000000
symbols = [[x_col[i], 0] for i in range(n)]

while diff > eps and count < 100000:
    count += 1
```

```
    x_pre = x_next
    alpha_pre = alpha_next
    f_pre = x_pre / alpha_pre
    for i in range(n):
        symbols[i][1] = f_pre[i]
    x_next[n - 1] = Full[n - 1, n].subs(symbols) / Full[n - 1, n - 1]
    for i in range(n - 2, -1, -1):
        x_next[i] = (Full[i, n].subs(symbols) - sum([Full[i, k] * x_next[k] for k in range(i + 1, n)
            ])) / Full[i, i]
    max_x = 0
    for i in range(n):
        if abs(max_x) < abs(x_next[i]):
            max_x = x_next[i]
    alpha_next = max_x
    diff = abs((1 / alpha_pre) - (1 / alpha_next))

lamda = 1 / alpha_next

print("Count: ", count)

validate_lamda = abs(np.linalg.det(A - lamda * np.eye(n)))
print("Diff:", diff)
```

4.4. Вывод

4.4.1. Сравнение

Входные данные:

```
A = np.array([
    [5, 17, 14, 15, 2],
    [12, 16, 2, 14, 8],
    [3, 10, 4, 10, 6],
    [10, 2, 1, 5, 6],
    [6, 6, 4, 15, 5]
])

A = np.array([
    [11, 9, 5, 5, 12, 11, 7, 19, 12, 17, 19, 9, 13, 14, 5],
    [5, 17, 2, 5, 11, 3, 15, 11, 5, 16, 8, 10, 13, 4, 18],
    [17, 16, 19, 18, 1, 15, 8, 12, 10, 10, 10, 13, 1, 14, 13],
    [12, 17, 5, 2, 1, 9, 3, 4, 9, 7, 6, 17, 3, 3, 4],
    [19, 1, 19, 7, 17, 7, 3, 8, 11, 18, 9, 4, 8, 11, 7],
    [19, 11, 16, 3, 15, 9, 2, 18, 11, 12, 12, 2, 9, 18, 10],
    [7, 3, 2, 7, 19, 13, 17, 18, 17, 18, 6, 8, 15, 8, 5],
    [1, 8, 14, 1, 3, 7, 15, 15, 1, 1, 9, 6, 11, 18, 4],
    [8, 14, 16, 4, 8, 18, 19, 11, 5, 11, 19, 15, 17, 18, 9],
    [6, 13, 15, 10, 7, 16, 17, 11, 13, 14, 4, 16, 4, 9, 1],
    [1, 8, 6 16, 19, 12, 10, 17, 4, 6, 5, 9, 9, 10, 15],
    [9, 10, 16, 1, 2, 15, 16, 13, 9, 8, 5, 6, 8, 11, 5],
    [16, 14, 2, 7, 6, 17, 18, 6, 17, 19, 18, 17, 2, 15, 13],
    [11, 17, 18, 19, 14, 19, 3, 17, 6, 10, 10, 12, 11, 10, 9],
    [13, 11, 18, 16, 2, 5, 6, 9, 10, 15, 6, 19, 17, 15, 13]
])
```

Количество итераций и максимальная погрешность: $n = 5$:

1. Простая итерация:

   14

   8.258316253062503e-11

2. Обратная итерация:

   167

   8.445244503718641e-11

3. Вращения с преградами

   4

   4.465090627801794e-10

$n = 15$:

1. Простая итерация:
   12
   2.936589860169647e-11

2. Обратная итерация:
   14
   2.2604584870578037e-11

3. Вращения с преградами
   6
   9.046676967212455e-10

4.4.2. Заключение

Исходя из представленных данных, можно сделать следующие выводы:

- Простая итерация демонстрирует хорошие результаты по погрешности на небольших выборках ($n = 5$). Однако с увеличением размера матрицы ($n = 15$) наблюдается незначительное увеличение погрешности и снижение числа итераций.

- Обратная итерация, напротив, показывает значительное увеличение числа итераций при увеличении размера матрицы, что свидетельствует о более низкой скорости сходимости этого метода.

- Метод вращений с преградами обладает высокой точностью на всех исследованных выборках ($n = 5, 15$). При этом наблюдается некоторое увеличение числа итераций при увеличении размера матрицы, но это увеличение является незначительным по сравнению с другими методами.

Таким образом, метод вращений с преградами демонстрирует наилучшую точность и эффективность среди представленных методов на всех выборках.

5. Заключение

В этой лабораторной работе были рассмотрены примеры использования методов решения систем линейных алгебраических уравнений, вычисления обратных матриц и определителей и вы числения собственных значений и собственных векторов матриц и проведено их сравнение.