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 **5**

2 Точные методы решения **6**

2.1 Схема Гаусса с выбором главного элемента 6

2.1.1 Описание метода 6

2.1.2 Код 8

2.1.3 Описание кода 9

2.2 Метод оптимального исключения 10

2.2.1 Описание метода 10

2.2.2 Код 11

2.2.3 Описание кода 12

2.3 Метод LU разложения 13

2.3.1 Описание метода 13

2.3.2 Код 15

2.3.3 Описание кода 16

2.4 Метод квадратного корня 17

2.4.1 Описание метода 17

2.4.2 Код 18

2.4.3 Описание кода 19

2.5 Метод окаймления 20

2.5.1 Описание метода 20

2.5.2 Код 22

2.5.3 Описание кода 23

2.6 Метод отражения 24

2.6.1 Описание метода 24

2.6.2 Код 28

2.6.3 Описание кода 29

2.7 Вывод 31
2.7.1 Погрешности 31
2.7.2 Заключение 32

3 Итерационные методы **34**
3.1 Простая итерация, релаксация 35
3.1.1 Описание метода 35
3.1.2 Код 36
3.1.3 Описание кода 37
3.2 Градиентный спуск 38
3.2.1 Описание метода 38
3.2.2 Код 38
3.2.3 Описание кода 39
3.3 Вращения с преградами 41
3.3.1 Описание метода 41
3.3.2 Код 41
3.3.3 Описание кода 42
3.4 Ричардсон 44
3.4.1 Описание метода 44
3.4.2 Код 44
3.4.3 Описание кода 45
3.5 Вывод 46
3.5.1 Количество итераций 46
3.5.2 Заключение 46

4 Частичная проблема собственных значений **48**
4.1 Простая итерация 48
4.1.1 Описание метода 48
4.1.2 Код 48

4.1.3 Описание кода 49
4.2 Обратная итерация 51
4.2.1 Описание метода 51
4.2.2 Код 51
4.2.3 Описание кода 52
4.3 Вывод 53
4.3.1 Чёто там 53
4.3.2 Заключение 53

5 Заключение **54**

1. Введение

В этой 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 = find_pivot(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)
```

2.1.3. Описание кода

1. В функции findPivot реализуется выбор главного элемента. Данная функция находит индекс строки, содержащей главный элемент в столбце p среди строк от p до $n - 1$. Выбирается строка, в которой значение элемента в столбце p имеет максимальное по модулю значение.

2. В функции gaussianElimination происходит прямой ход метода Гаусса. На каждом шаге выбирается главный элемент, затем происходит перестановка строк и столбцов для обеспечения максимальной точности вычислений. Далее осуществляется вычитание строки с коэффициентом из всех нижестоящих строк для приведения матрицы к верхнетреугольному виду.

3. При прохождении обратного хода метода Гаусса вычисляются неизвестные относительно главной диагонали сверху вниз.

4. В конце, выводится решение уравнения (набор значений неизвестных) и вычисляется погрешность, сравнивая решение, полученное через умножения матрицы A на вектор x, с вектором 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 $a_{11}! = 0$. 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

$$\begin{cases} x_1 + ... + a_{1k+1}^{(k)} x_{k+1} + ... + a_{1n}^{(k)} x_n = a_{1n+1}^{(k)} \\ x_2 + ... + a_{2k+1}^{(k)} x_{k+1} + ... + a_{2n}^{(k)} x_n = a_{2n+1}^{(k)} \\ ... \end{cases}$$

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)))
```

Код программы

2.2.3. Описание кода

1. В функции optimalElimination происходит инициализация первой строки расширенной матрицы A: каждый элемент делится на значение, находящееся на главной диагонали.

2. Затем, запускается итерация, в которой происходит вычисление всего оставшегося уравнения системы методом оптимального исключения. Для этого используется вложенный цикл, который просматривает все строки в матрице. На каждом шаге происходит коррекция значений, чтобы получить нули под главной диагональю.

3. В конце, после того как метод завершает свою работу, мы получаем диагональную матрицу A.

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

— верхнетреугольная матрица

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

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

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 $y_1 = \frac{a_{1n+1}}{b_{11}}; y_i = \left(a_{in+1} - \sum_{k=1}^{i-1} b_{ik}y_k \right) /b_{ii}$ $1), x_n = y_n; x_i = y_i - \sum_{k=i+1}^{n} c_{ik}x_k, (i < n).$

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')
```

2.3.3. Описание кода

1. Метод lu(matrix) принимает на вход квадратную матрицу matrix и возвращает верхнетреугольную матрицу U и нижнетреугольную матрицу L таким образом, что matrix $= L \cdot U$.

2. Метод solveLu принимает верхнетреугольную матрицу U, нижнетреугольную матрицу L и вектор b и решает систему линейных уравнений $LU \cdot x = b$.

3. После вычисления L и U, метод solveLu вычисляет y используя прямой ход метода Гаусса, после чего использует обратный ход метода Гаусса для вычисления x.

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)))
```

Код программы

2.4.3. Описание кода

1. Сначала мы инициализируем нулями векторы Y, X и матрицу S, которая будет содержать элементы матрицы квадратного корня

2. Проходимся по матрице S и вычисляем ее элементы в соответствии с методом квадратного корня. Это включает в себя подсчет сумм и использование квадратного корня из элементов матрицы A.

3. Далее используем матрицу S для решения системы уравнений. Мы сначала находим вектор Y путем прямого хода метода Гаусса. Затем мы используем обратный ход метода Гаусса для нахождения вектора X.

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

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

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

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} по формуле **??** 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))))
```

Код программы

2.5.3. Описание кода

1. В фунции getInv мы проверяем, если размерность матрицы n равна 1, т.е. если матрица является $1x1$, мы возвращаем обратную матрицу этого элемента.

2. Если размерность больше 1, то:

   (a) Создаем подматрицу Ap, а также векторы V и U.

   (b) Рекурсивно вызываем getInv для подматрицы Ap для нахождения ее обратной матрицы ApInv.

   (c) Вычисляем alpha, Q, P и R с использованием найденных значений по формулам. Эти вычисления базируются на вашем алгоритме окаймления.

   (d) Создаем новую матрицу $AInv$ размером (n, n) и заполняем ее значениями P, Q, R и alpha.

   (e) Затем мы возвращаем найденную обратную матрицу $AInv$.

3. Потом находим x из системы $A^{-1} \cdot b = x$

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 $p(k) = (0, ..., 0, p_k^{(k)}, p$ положив:

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

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

Определим теперь матрицу отражения P_k с элементами $p_{ij}^{(k)} = \sigma_{ij} - 2p_i^{(k)} p_j^{(k)} / \sum_{l=k}^{n} \left(p_l^{(k)}\right)^2$ где σ - символ Кронеккера

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

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

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 $A_{n-1} = P_{n-1}P_{n-2}...P_1A$, которую мы в дальнейшем будем обозначать через R : $R = A_{n-1}$. Обозначив еще $Q = P_{n-1}P_{n-2}...P_1$, 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 $g = P_{n-1}P_{n-2}...P_1f$. Обозначим $f^{(k)} = P_{n-1}P_{n-2}...P_1f$. Тогда $f^{(k)} = P_kf^{(k-1)}$. 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 $f_i^{(k)} = f_i^{(k-1)} - 2p_l^{(k)} \frac{\sum_{l=k}^{n} p_l^{(k)} f_l^{(k-1)}}{\sum_{l=k}^{n} \left(p_l^{(k)}\right)^2}, i = k, k+1, ..., n$

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)))
```

2.6.3. Описание кода

1. Сначала задаются исходные данные: матрица A, параметры для вывода результатов в числовом формате и размерность матрицы n.

2. Затем объявляются необходимые функции:

   (a) Функция vectornormali(a, k) принимает матрицу a и индекс k и возвращает вектор pStepk. Она создает два вектора pStepkB и pStepkK Затем она выбирает элементы матрицы a начиная с индекса k и возвращаются элементы начиная с $(k+1)$-го индекса. Затем она проверяет знак первого элемента $a[i]$ и на основе этого присваивает значение переменной sigmaK. Затем вычисляются элементы pStepkK по формуле. В итоге функция возвращает соединение векторов pStepkB и pStepkK

   (b) Функция solveR(aB, k, r, pStepk) принимает матрицу aB, индекс k, матрицу r и вектор pStepk и возвращает матрицу r. Для каждого i от $(k+1)$ до n она создает вектор aForDot, содержащий элементы столбца i матрицы aB. Затем проходит по всем элементам $r[i-1][j]$ и присваивает им значения по формуле . Потом результаты умножаются на pStepk и результаты делятся на сумму квадратов элементов pStepk с индексами от k до n. Затем матрица r транспонируется и возвращается.

   (c) Функция solveA(aB, pStepk, k) принимает матрицу aB, вектор pStepk и индекс k и возвращает новую матрицу newA.

   (d) Функция solveX(g, r) принимает вектор g и матрицу r и возвращает вектор x. Инициализируется вектор x нулями. Для каждого i от $(n-2)$ до -1 с шагом -1 вычисляется значение $x[i]$ по формуле.

3. Происходит итерационный процесс метода отражений:

(a) Для каждого к-го шага от 0 до n-1:

   i. Создается копия матрицы A.

   ii. Вычисляется вектор pStepk с помощью функции vectornormali(aB, k).

   iii. Вычисляется значение s - норма вектора pStepk.

   iv. Создается матрица r с размерностью $n \cdot n$ путем вызова функции solveR(aB, k, r, pStepk).

   v. Матрица r добавляется в список RMatrix.

   vi. Вычисляется новая матрица A путем вызова функции solveA(aB, pStepk

   vii. Новая матрица A добавляется в список AllMatrix.

(b) Матрица A на последнем шаге равна AllMatrix[n-1].

(c) Выделяется подматрица R размерностью nxn из матрицы A.

(d) Выделяется вектор b - последний столбец матрицы A.

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, 0.48, 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]])

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 $A|a_{ii}| > \sum_{j \neq i} |a_{ij}|$ 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, релаксация

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

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

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))
```

Код программы

3.1.3. Описание кода

3.2. Градиентный спуск

3.2.1. Описание метода

3.2.2. 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")
```

Код программы

3.2.3. Описание кода

3.3. Вращения с преградами

3.3.1. Описание метода

3.3.2. Код

```
# Вращенияспрегradами

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_l, j_l = 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_l, j_l = i, j
        return i_l, j_l

    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
```

3.3.3. Описание кода

3.4. Ричардсон

3.4.1. Описание метода

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
```

3.4.3. Описание кода

3.5. Вывод

3.5.1. Количество итераций

3.5.2. Заключение

4. Частичная проблема собственных значений

4.1. Простая итерация

4.1.1. Описание метода

4.1.2. Код

```
import numpy as np

np.set_printoptions(linewidth=200)

e = 1e-8


# A = np.array(
#     [
#         [-0.1687, 0.353699, 0.00854, 0.733624],
#         [0.353699, 0.056519, -0.723182, -0.07644],
#         [0.00854, -0.723182, 0.015938, 0.342333],
#         [0.733624, -0.07644, 0.342333, -0.045744],
#     ]
# )

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[0]
x = np.ones(n)
l = 1

steps = 0
while np.max(np.abs(A.dot(x) - l * x)) > e:
    steps += 1
    i = np.where(np.abs(x) == np.max(np.abs(x)))[0][0]
    l = x[i]
    x = A.dot((x / l))

print(f"steps: {steps} \n\nl: {l} \n\nx: {x/l}\n\n")

print(
    np.linalg.eig(A).eigenvectors[:, 0]
    / np.max(np.abs(np.linalg.eig(A).eigenvectors[:, 0]))
)
print(np.linalg.eig(A).eigenvalues)

print(np.max(np.abs(A.dot(x) - l * x)))
```

Код программы

4.1.3. Описание кода

4.2. Обратная итерация

4.2.1. Описание метода

4.2.2. Код

```
import numpy as np


def EdgingMethod(mat):
    n = mat.shape[0]
    inv = np.zeros((n, n))

    inv[0][0] = 1 / mat[0][0]

    for i in range(1, n):
        u = mat[:i, i].reshape(i, 1)
        v = mat[i, :i].reshape(1, i)
        aii = mat[i, i]

        alph = 1 / (aii - v @ inv[:i, :i] @ u)[0][0]
        r = -alph * (inv[:i, :i] @ u)
        q = -alph * (v @ inv[:i, :i])
        P = inv[:i, :i] - (inv[:i, :i] @ u) @ q

        inv[:i, :i] = P
        inv[:i, i] = r.reshape(i)
        inv[i, :i] = q.reshape(i)
        inv[i, i] = alph

    return inv


e = 1e-9

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[0]
x = np.ones(n)
l = 1
invA = EdgingMethod(A)

steps = 0
while np.max(np.abs(invA.dot(x) - 1 * x)) > e:
    steps += 1
    i = np.where(np.abs(x) == np.max(np.abs(x)))[0][0]
    l = x[i]
    x = invA.dot((x / l))

print(f'iters: {steps} \n\nl: {l} \n\nx: {x / l}\n\n')
print(
    np.linalg.eig(A).eigenvectors[:, 0]
    / np.max(np.abs(np.linalg.eig(A).eigenvectors[:, 0]))
)
```

4.2.3. Описание кода

4.3. Вывод

4.3.1. Чёто там

4.3.2. Заключение

5. Заключение

В этой лабораторной работе была проведена работа по программированию и тестированию алгоритма выбора главного элемента для решения системы линейных алгебраических уравнений.