**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский Авиационный Институт»**

**(Национальный Исследовательский Университет)**

**Институт: №8 «Компьютерные науки и прикладная математика»   
Кафедра: 806 «Вычислительная математика   
и программирование»**

Лабораторная работа № 1   
по курсу «Численные методы»

Группа: М8О-309Б-22

Студент: Концебалов О. С.

Преподаватель: Шабунина А. А.

Дата: 04.06.2025

Москва, 2025

**ОГЛАВЛЕНИЕ**

[1](#_xgy0ay3rv3vf) Тема 3

[2](#_mo8tvn3t3nqj) Задание 3

[3](#_753pinjhjxeu) Теория 5

[4](#_bdk4mqpei3mu) Ход лабораторной работы 8

[5](#_uw40e425m4ja) Выводы 25

# **Тема**

Методы решения задач линейной алгебры

# **Задание**

1.1. Реализовать алгоритм LU - разложения матриц (с выбором главного элемента) в виде программы. Используя разработанное программное обеспечение, решить систему линейных алгебраических уравнений (СЛАУ). Для матрицы СЛАУ вычислить определитель и обратную матрицу.



1.2. Реализовать метод прогонки в виде программы, задавая в качестве входных данных ненулевые элементы матрицы системы и вектор правых частей. Используя разработанное программное обеспечение, решить СЛАУ с трехдиагональной матрицей.



1.3. Реализовать метод простых итераций и метод Зейделя в виде программ, задавая в качестве входных данных матрицу системы, вектор правых частей и точность вычислений. Используя разработанное программное обеспечение, решить СЛАУ. Проанализировать количество итераций, необходимое для достижения заданной точности.



1.4. Реализовать метод вращений в виде программы, задавая в качестве входных данных матрицу и точность вычислений. Используя разработанное программное обеспечение, найти собственные значения и собственные векторы симметрических матриц. Проанализировать зависимость погрешности вычислений от числа итераций.

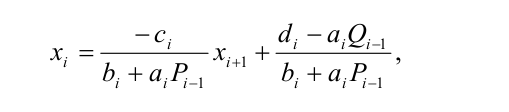


1.5. Реализовать алгоритм QR – разложения матриц в виде программы. На его основе разработать программу, реализующую QR – алгоритм решения полной проблемы собственных значений произвольных матриц, задавая в качестве входных данных матрицу и точность вычислений. С использованием разработанного программного обеспечения найти собственные значения матрицы.



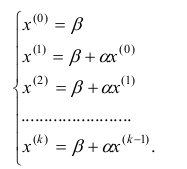
# **Теория**

**LU – разложение** матрицы A представляет собой разложение матрицы A в произведение нижней и верхней треугольных матриц, т.е. A = LU, где L - нижняя треугольная матрица (матрица, у которой все элементы, находящиеся выше главной диагонали равны нулю, l ij = 0 при i < j ), U - верхняя треугольная матрица (матрица, у которой все элементы, находящиеся ниже главной диагонали равны нулю, u ij = 0 при i > j ). В дальнейшем LU – разложение может быть эффективно использовано при решении систем линейных алгебраических уравнений вида Ax = b.

**Метод прогонки** является одним из эффективных методов решения СЛАУ с трех - диагональными матрицами, возникающих при конечно-разностной аппроксимации задач для обыкновенных дифференциальных уравнений (ОДУ) и уравнений в частных производных второго порядка и является частным случаем метода Гаусса.

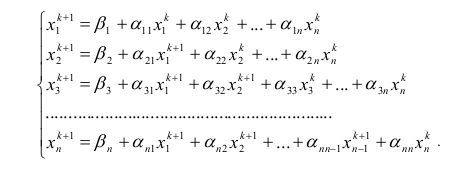


Методы последовательных приближений, в которых при вычислении последующего приближения решения используются предыдущие, уже известные приближенные решения, называются **итерационными.**



Для сходимости итерационного процесса необходимо и достаточно, чтобы спектр матрицы α эквивалентной системы лежал внутри круга с радиусом, равным единице.

Метод простых итераций довольно медленно сходится. Для его ускорения существует **метод Зейделя**, заключающийся в том, что при вычислении компонента вектора неизвестных на (k+1)-й итерации используются x1k+1, x2k+1, …, xi-1k+1, уже вычисленные на (k+1)-й итерации.



**Метод вращений Якоби** применим только для симметрических матриц и решает полную проблему собственных значений и собственных векторов

таких матриц. Он основан на отыскании с помощью итерационных процедур матрицы U в преобразовании подобия Λ = U-1AU, а поскольку для симметрических матриц A матрица преобразования подобия U является ортогональной (U-1 UT), то Λ = UTAU, где Λ - диагональная матрица с собственными значениями на главной диагонали.

При решении полной проблемы собственных значений для несимметричных матриц эффективным является подход, основанный на приведении матриц к подобным, имеющим треугольный или квазитреугольный вид. Одним из наиболее распространенных методов этого класса является **QR-алгоритм**, позволяющий находить как вещественные, так и комплексные собственные значения.

В основе QR-алгоритма лежит представление матрицы в виде A = QR, где Q-ортогональная матрица, а R - верхняя треугольная. Такое разложение существует для любой квадратной матрицы. Одним из возможных подходов к построению QR разложения является использование преобразования Хаусхолдера, позволяющего обратить в нуль группу поддиагональных элементов столбца матрицы.

# **Ход лабораторной работы**

Код был реализован на языке Python.

* 1. LU-разложение:

import numpy as np

np.set\_printoptions(precision=6, suppress=True)

def LU\_decompose(A):

n = len(A)

L = [[0 for \_ in range(n)] for \_ in range(n)]

U = np.copy(A)

for k in range(0, n):

for i in range(k, n):

L[i][k] = U[i][k] / U[k][k]

for i in range(k + 1, n):

for j in range(n):

U[i][j] -= L[i][k] \* U[k][j]

return L, U

def LU\_method(A: np.ndarray, b: np.ndarray):

A = A.copy()

b = b.copy()

n = A.shape[0]

amount\_replace = 0

for i in range(n):

max\_row = i + np.argmax(np.abs(A[i:, i]))

if abs(A[max\_row, i] - A[i, i]) > 1e-8:

A[[i, max\_row]] = A[[max\_row, i]]

b[[i, max\_row]] = b[[max\_row, i]]

amount\_replace += 1

for k in range(i + 1, n):

coefficient = A[k, i] / A[i, i]

A[k, i:] -= coefficient \* A[i, i:]

b[k] -= coefficient \* b[i]

# Обратный ход (решение Ax = b через LU)

x = np.zeros(n, dtype=np.float64)

# Прямой ход для Ly = b (L неявно в нижнем треугольнике A)

y = np.zeros(n, dtype=np.float64)

for i in range(n):

y[i] = b[i] - np.dot(A[i, :i], y[:i])

# Обратный ход для Ux = y

for i in range(n - 1, -1, -1):

x[i] = (y[i] - np.dot(A[i, i + 1:], x[i + 1:])) / A[i, i]

return x, amount\_replace, A

def get\_inverse\_matrix(A):

n = A.shape[0]

identity = np.eye(n)

inv = np.zeros\_like(A, dtype=np.float64)

for i in range(n):

e = identity[:, i]

inv[:, i], \_, \_ = LU\_method(A, e)

return inv

def check\_solution(A, b, x):

b\_calc = A @ x

print("My answer Real b")

for bi, br in zip(b\_calc, b):

print(f"{bi:.6f} {br:.6f}")

def print\_matrix(name, matrix):

print(f"{name}:")

print(np.round(matrix, 6))

print()

def main():

n = int(input('Введите размерность матрицы системы (одно положительное число): '))

print(f"Введите матрицу A размером {n}x{n} построчно (элементы разделены пробелами):")

A = np.array([list(map(np.float64, input().split())) for \_ in range(n)], dtype=np.float64)

print("Введите столбец свободных членов b (элементы через пробел):")

b = np.array(list(map(np.float64, input().split())), dtype=np.float64)

if abs(np.linalg.det(A)) < 1e-10:

print("Матрица вырождена. Система имеет бесконечно много решений, либо не имеет ни одного. Обратной матрицы не существует")

exit(0)

L, U = LU\_decompose(A)

x, amount\_replace, lu\_matrix = LU\_method(A, b)

print("\nLU-разложение:")

print\_matrix("L matrix", L)

print\_matrix("U matrix", U)

print("Проверка L \* U:")

print\_matrix("L \* U", L @ U)

print("Вектор решения x:")

for x\_i in x:

print(f"{x\_i:.6f}")

det = ((-1) \*\* amount\_replace) \* np.prod(np.diag(lu\_matrix))

print(f"\ndet(A): {det:.6f}")

if np.abs(det) < 1e-10:

print("Матрица A вырождена, обратной не существует.")

return

A\_inv = get\_inverse\_matrix(A)

print\_matrix("Обратная матрица A^-1", A\_inv)

print("Проверка A \* A^-1:")

print\_matrix("A \* A^-1", A @ A\_inv)

print("Проверка A^-1 \* A:")

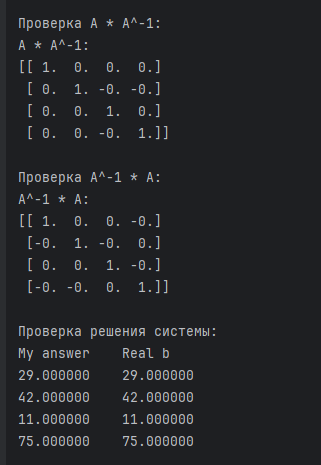
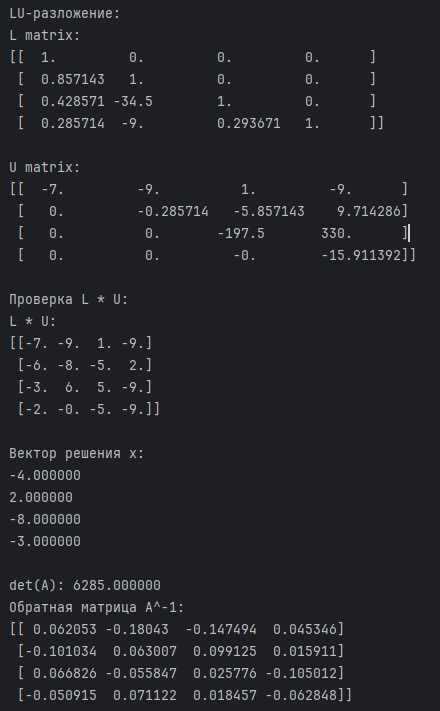
print\_matrix("A^-1 \* A", A\_inv @ A)

print("Проверка решения системы:")

check\_solution(A, b, x)

if \_\_name\_\_ == "\_\_main\_\_":

main()



1.2) Метод прогонки:

import numpy as np

def float64\_eq(a, b) -> bool:

return abs(a - b) < 1e-8

def check\_input\_matrix(matrix: np.ndarray) -> None:

matrix\_size = matrix.shape[0]

for i in range(matrix\_size):

for j in range(matrix\_size):

if abs(i - j) > 1 and not float64\_eq(matrix[i, j], 0):

print("ERROR: Метод Прогонки не применим - матрица не является трехдиагональной!")

exit(1)

def check\_diagonals(a, b, c) -> None:

is\_failed = False

is\_warning = False

if float64\_eq(c[0], 0) or float64\_eq(b[0], 0) or float64\_eq(b[-1], 0) or float64\_eq(a[-1], 0):

is\_failed = True

print("First or last line has incorrect values")

is\_b\_bigger = 0

for i in range(1, len(b) - 1):

if is\_failed:

break

if abs(b[i]) > abs(a[i]) + abs(c[i]):

is\_b\_bigger += 1

if abs(b[i]) < abs(a[i]) + abs(c[i]):

is\_warning = True

print("WARNING: One or more rows failed check: abs(b) < abs(a) + abs(c)\nМетод может расходиться")

if float64\_eq(a[i], 0):

is\_failed = True

print("One or more values in A vector equals zero")

if float64\_eq(c[i], 0):

is\_failed = True

print("One or more values in C vector equals zero")

if is\_failed:

return

if is\_b\_bigger == 0:

is\_failed = True

print("One or more rows failed check: abs(b) < abs(a) + abs(c)")

if is\_failed:

print("ERROR: Метод Прогонки не применим - матрица некорректна!")

exit(1)

def extract\_diagonals(matrix: np.ndarray) -> tuple[list[np.float64], list[np.float64], list[np.float64]]:

n = len(matrix)

a: list[np.float64] = [np.float64(0.0)] \* n

b: list[np.float64] = [np.float64(0.0)] \* n

c: list[np.float64] = [np.float64(0.0)] \* n

for i in range(n):

b[i] = matrix[i][i]

if i > 0:

a[i] = matrix[i][i - 1]

if i < n - 1:

c[i] = matrix[i][i + 1]

return a, b, c

def check\_answer(

a: np.ndarray,

b: np.ndarray,

c: np.ndarray,

d: np.ndarray,

results: np.ndarray

) -> None:

n = len(results) - 1

answer = [0.0] \* len(results)

print("\nMy answer Real answer")

answer[0] = results[0] \* b[0] + results[1] \* c[0]

print(f"{answer[0]:.6f} {d[0]:.6f}")

for i in range(1, n):

answer[i] = results[i - 1] \* a[i] + b[i] \* results[i] + c[i] \* results[i + 1]

print(f"{answer[i]:.6f} {d[i]:.6f}")

answer[n] = results[n - 1] \* a[n] + b[n] \* results[n]

print(f"{answer[n]:.6f} {d[n]:.6f}")

def main():

n = int(input('Введите размерность матрицы системы (одно положительное число): '))

print(f"Введите матрицу A размером {n}x{n} построчно (элементы разделены пробелами):")

A = np.array([list(map(np.float64, input().split())) for \_ in range(n)], dtype=np.float64)

print("Введите столбец свободных членов b (элементы через пробел):")

b = np.array(list(map(np.float64, input().split())), dtype=np.float64)

check\_input\_matrix(A)

a\_diag, b\_diag, c\_diag = extract\_diagonals(A)

check\_diagonals(a\_diag, b\_diag, c\_diag)

x: list[np.float64] = [np.float64(0.0)] \* n

p: list[np.float64] = [np.float64(0.0)] \* n

q: list[np.float64] = [np.float64(0.0)] \* n

p[0] = -(c\_diag[0] / b\_diag[0])

q[0] = b[0] / b\_diag[0]

for i in range(1, n):

divider = b\_diag[i] + a\_diag[i] \* p[i - 1]

if abs(divider) < 1e-8:

print("ERROR: Эта система не может быть решена Методом Прогонки - происходит деление на 0!")

exit(1)

p[i] = -c\_diag[i] / divider

q[i] = (b[i] - a\_diag[i] \* q[i - 1]) / divider

for val in p:

if abs(val) > 1:

print("ERROR: One or more elements of vector P evaluated out of range (-1; 1)!")

exit(1)

x[-1] = q[-1]

for i in range(n - 2, -1, -1):

x[i] = p[i] \* x[i + 1] + q[i]

print("\nAnswer:")

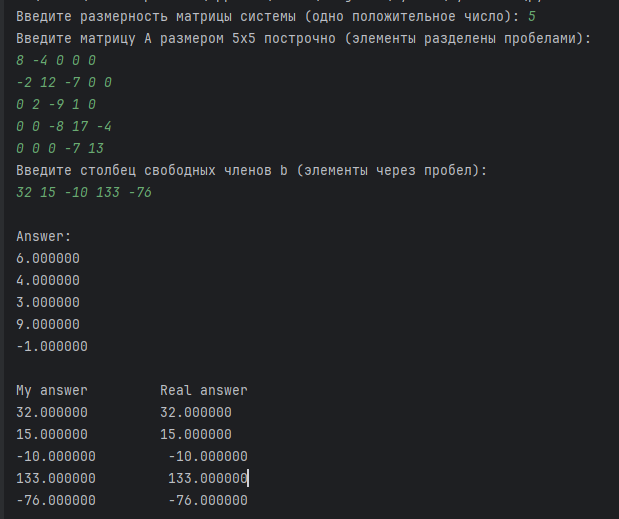
for x\_i in x:

print(f"{x\_i:.6f}")

check\_answer(a\_diag, b\_diag, c\_diag, b, x)

if \_\_name\_\_ == '\_\_main\_\_':

main()



1.3) Метод простых итераций и метод Зейделя:

### Simple Iterations Method

import numpy as np

def print\_matrix(matrix):

for row in matrix:

print(' '.join(f'{elem:.6f}' for elem in row))

print()

def print\_vector(vector):

for val in vector:

print(f"{val:.6f}")

print()

def check\_matrix\_for\_simple\_iteration(matrix):

n = len(matrix)

for i in range(n):

if matrix[i][i] == 0:

return False

row\_sum = sum(abs(matrix[i][j]) for j in range(n) if j != i)

if abs(matrix[i][i]) < row\_sum:

found = False

for h in range(i + 1, n):

row\_sum\_h = sum(abs(matrix[h][t]) for t in range(n) if t != h)

if abs(matrix[h][i]) > row\_sum and abs(matrix[h][h]) > row\_sum\_h:

matrix[i], matrix[h] = matrix[h], matrix[i]

found = True

break

if not found:

return False

return True

def get\_matrix\_c\_f\_for\_simple\_iteration(matrix):

n = len(matrix)

result = np.zeros((n, n + 1), dtype=float)

for i in range(n):

for j in range(n):

if i != j:

result[i][j] = -matrix[i][j] / matrix[i][i]

result[i][n] = matrix[i][n] / matrix[i][i]

return result

def get\_next\_vector\_of\_x(matrix, vec):

n = len(matrix)

result\_vec = np.zeros(n, dtype=float)

for i in range(n):

for j in range(n):

result\_vec[i] += vec[j] \* matrix[i][j]

result\_vec[i] += matrix[i][n]

return result\_vec

def get\_max\_diff\_for\_two\_vectors(first, second):

if len(first) != len(second):

raise ValueError("Vectors have different lengths!")

return np.max(np.abs(np.subtract(first, second)))

def find\_matrix\_norm(matrix):

n = len(matrix)

max\_norm = 0

for i in range(n):

row\_sum = sum(abs(matrix[i][j]) for j in range(n))

max\_norm = max(max\_norm, row\_sum)

return max\_norm

def check\_answer(matrix, n, results):

print("\nMy answer Real answer")

for i in range(n):

computed = sum(matrix[i][j] \* results[j] for j in range(n))

print(f"{computed:.6f} {matrix[i][n]:.6f}")

def main():

n = int(input('Введите размерность матрицы системы (одно положительное число): '))

print(f"Введите матрицу A размером {n}x{n} построчно (элементы разделены пробелами):")

A = np.array([list(map(np.float64, input().split())) for \_ in range(n)], dtype=np.float64)

print("Введите столбец свободных членов b (элементы через пробел):")

b = np.array(list(map(np.float64, input().split())), dtype=np.float64)

eps = float(input("Введите точность поиска решения (eps): "))

if abs(np.linalg.det(A)) < 1e-10:

print("ERROR: Матрица вырождена. Система имеет бесконечно много решений, либо не имеет ни одного!")

exit(0)

A = np.column\_stack((A, b))

if not check\_matrix\_for\_simple\_iteration(A):

print("ERROR: Матрица не имеет диагонального преобладания")

exit(1)

matrix\_c\_f = get\_matrix\_c\_f\_for\_simple\_iteration(A)

print("\nMatrix Alpha and Beta:")

print\_matrix(matrix\_c\_f)

vector\_result = np.zeros(n)

diff\_prev = 0.0

diff\_norm = 0.0

counter = 0

matrix\_norm = find\_matrix\_norm(matrix\_c\_f)

e\_k = 1

while e\_k > eps :

tmp = vector\_result.copy()

vector\_result = get\_next\_vector\_of\_x(matrix\_c\_f, vector\_result)

diff\_norm = get\_max\_diff\_for\_two\_vectors(vector\_result, tmp)

if matrix\_norm >= 1:

print(f"This system cannot be solved by the simple iteration method - matrix\_norm = {matrix\_norm}")

e\_k = diff\_norm

else:

e\_k = (matrix\_norm / (1 - matrix\_norm)) \* diff\_norm

counter += 1

print(f"The result is:")

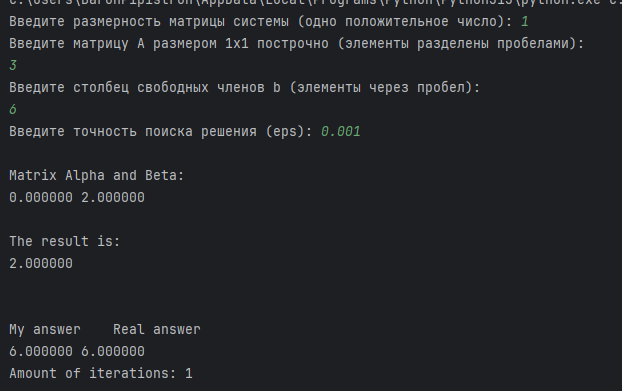
print\_vector(vector\_result)

check\_answer(A, n, vector\_result)

print(f"Amount of iterations: {counter}")

if \_\_name\_\_ == '\_\_main\_\_':

main()



### Seidel's Method

import numpy as np

from mpmath import matrix

def print\_matrix(matrix):

for row in matrix:

print(' '.join(f'{elem:.6f}' for elem in row))

print()

def print\_vector(vector):

for val in vector:

print(f"{val:.6f}")

print()

def check\_matrix\_for\_seidel\_method(matrix):

n = len(matrix)

for i in range(n):

if matrix[i][i] == 0:

return False

row\_sum = sum(abs(matrix[i][j]) for j in range(n) if j != i)

if abs(matrix[i][i]) < row\_sum:

found = False

for h in range(i + 1, n):

row\_sum\_h = sum(abs(matrix[h][t]) for t in range(n) if t != h)

if abs(matrix[h][i]) > row\_sum and abs(matrix[h][h]) > row\_sum\_h:

matrix[i], matrix[h] = matrix[h], matrix[i]

found = True

break

if not found:

return False

return True

def get\_matrix\_c\_f\_for\_seidel\_method(matrix):

n = len(matrix)

result = np.zeros((n, n + 1), dtype=float)

for i in range(n):

for j in range(n):

if i != j:

result[i][j] = -matrix[i][j] / matrix[i][i]

result[i][n] = matrix[i][n] / matrix[i][i]

return result

def get\_next\_vector\_of\_x(matrix, vec):

n = len(matrix)

result\_vec = [0.0 for \_ in range(n)]

for i in range(n):

for j in range(n):

result\_vec[i] += (vec[j] if j > i else result\_vec[j]) \* matrix[i][j]

result\_vec[i] += matrix[i][n]

return result\_vec

def get\_max\_diff(first, second):

if len(first) != len(second):

raise ValueError("Vectors must be of same length")

return max(abs(abs(a) - abs(b)) for a, b in zip(first, second))

def find\_matrix\_norm(matrix):

n = len(matrix)

max\_norm = 0

for i in range(n):

row\_sum = sum(abs(matrix[i][j]) for j in range(n))

max\_norm = max(max\_norm, row\_sum)

return max\_norm

def check\_answer(original\_matrix, result):

n = len(original\_matrix)

print("My answer Real answer")

for i in range(n):

lhs = sum(original\_matrix[i][j] \* result[j] for j in range(n))

rhs = original\_matrix[i][n]

print(f"{lhs:.6f} {rhs:.6f}")

def main():

n = int(input('Введите размерность матрицы системы (одно положительное число): '))

print(f"Введите матрицу A размером {n}x{n} построчно (элементы разделены пробелами):")

A = np.array([list(map(np.float64, input().split())) for \_ in range(n)], dtype=np.float64)

print("Введите столбец свободных членов b (элементы через пробел):")

b = np.array(list(map(np.float64, input().split())), dtype=np.float64)

eps = float(input("Введите точность поиска решения (eps): "))

if abs(np.linalg.det(A)) < 1e-10:

print("ERROR: Матрица вырождена. Система имеет бесконечно много решений, либо не имеет ни одного!")

exit(0)

A = np.column\_stack((A, b))

if not check\_matrix\_for\_seidel\_method(A):

print("ERROR: Матрица не имеет диагонального преобладания")

exit(1)

matrix\_c\_f = get\_matrix\_c\_f\_for\_seidel\_method(A)

print("\nMatrix Alpha and Beta:")

print\_matrix(matrix\_c\_f)

vector\_result = np.zeros(n)

diff\_prev = 0.0

diff\_curr = 0.0

diff\_norm = 0.0

counter = 0

matrix\_norm = find\_matrix\_norm(matrix\_c\_f)

e\_k = 1

while e\_k > eps:

tmp = vector\_result

vector\_result = get\_next\_vector\_of\_x(matrix\_c\_f, tmp)

diff\_norm = get\_max\_diff(vector\_result, tmp)

if matrix\_norm >= 1:

print(f"This system cannot be solved by the simple iteration method - matrix\_norm = {matrix\_norm}")

e\_k = diff\_norm

else:

e\_k = (matrix\_norm / (1 - matrix\_norm)) \* diff\_norm

counter += 1

print(f"The result is:")

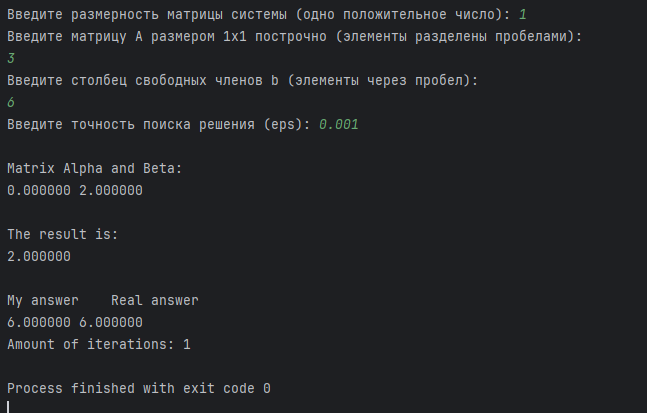
print\_vector(vector\_result)

check\_answer(A, vector\_result)

print(f"Amount of iterations: {counter}")

if \_\_name\_\_ == '\_\_main\_\_':

main()



1.4) Метод вращения:

import numpy as np

def check\_eigenvectors(

matrix,

eigenvalues,

eigenvectors

) -> bool:

for i in range(matrix.shape[0]):

check\_vector = (eigenvectors[i] @ matrix) / eigenvalues[i]

if not np.allclose(check\_vector, eigenvectors[i]):

return False

return True

def is\_symmetric\_matrix(matrix) -> bool:

matrix\_size = matrix.shape[0]

for i in range(matrix\_size):

for j in range(i + 1, matrix\_size):

if matrix[i, j] != matrix[j, i]:

return False

return True

# Позиция максимального элемента

def find\_max\_upper\_element(X):

n = X.shape[0]

i\_max, j\_max = 0, 1

max\_elem = abs(X[0][1])

for i in range(n):

for j in range(i + 1, n):

if abs(X[i][j]) > max\_elem:

max\_elem = abs(X[i][j])

i\_max = i

j\_max = j

return i\_max, j\_max

# Норма матрицы

def matrix\_norm(X):

norm = 0

for i in range(len(X[0])):

for j in range(i + 1, len(X[0])):

norm += X[i][j] \* X[i][j]

return np.sqrt(norm)

# Метод вращений

def rotation\_method(A\_, eps):

n = A\_.shape[0]

A = np.copy(A\_)

eigen\_vectors = np.eye(n)

iterations = 0

while matrix\_norm(A) > eps:

iterations += 1

i\_max, j\_max = find\_max\_upper\_element(A)

if A[i\_max][i\_max] - A[j\_max][j\_max] == 0:

phi = np.pi / 4

else:

phi = 0.5 \* np.arctan(2 \* A[i\_max][j\_max] / (A[i\_max][i\_max] - A[j\_max][j\_max]))

# Матрица вращения

U = np.eye(n)

U[i\_max][j\_max] = -np.sin(phi)

U[j\_max][i\_max] = np.sin(phi)

U[i\_max][i\_max] = np.cos(phi)

U[j\_max][j\_max] = np.cos(phi)

A = U.T @ A @ U

eigen\_vectors = eigen\_vectors @ U

eigen\_values = [A[i][i] for i in range(n)]

return eigen\_values, eigen\_vectors, iterations

def calculate\_results(matrix, eps):

try:

A = np.copy(matrix)

# Выполняем метод вращений

values, vectors, iters = rotation\_method(A, eps)

# Формируем результат

result\_text = f"\nСобственные значения:\n{', '.join(map(str, values))}\n"

result\_text += "Собственные вектора:\n"

for i in range(vectors.shape[0]):

result\_text += f"СВ {i + 1}: {vectors[:, i]}\n"

result\_text += f"Итераций: {iters}\n"

result\_text += "\nПроверка:\n"

result\_text += f"След матрицы: {sum([matrix[i][i] for i in range(matrix.shape[0])])}\n"

result\_text += f"Сумма собственных значений: {sum(values)}\n"

result\_text += "\nПроверка уравнения A \* v = λ \* v для каждого собственного вектора:\n\n"

for i in range(n):

vec = vectors[:, i]

lam = values[i]

lhs = matrix @ vec

rhs = lam \* vec

result\_text += f"Собственный вектор {i + 1}:\n"

result\_text += f"A \* v = {lhs}\n"

result\_text += f"λ \* v = {rhs}\n"

if np.allclose(lhs, rhs, atol=1e-6):

result\_text += "Проверка пройдена\n\n"

else:

result\_text += "Проверка не пройдена\n\n"

result\_text += "\nv\_i - (v\_i \* A) / λ\_i\n"

print(result\_text)

except ValueError as e:

print("Ошибка", f"\nНеверный ввод данных: {str(e)}")

exit(1)

if \_\_name\_\_ == '\_\_main\_\_':

n = int(input('Введите размерность матрицы (одно положительное число): '))

print(f'Введите матрицу {n}x{n}: ')

input\_matrix = np.empty((n, n), dtype=float)

for row in range(n):

input\_matrix[row] = np.array(list(map(

float,

input().split())))

eps = float(input("Введите точность поиска решения (eps): "))

if not is\_symmetric\_matrix(input\_matrix):

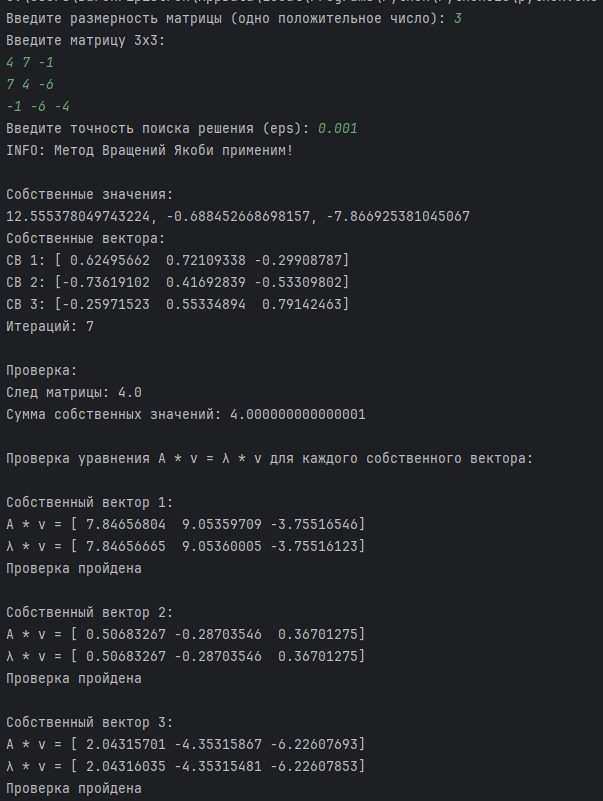
print('ERROR: Метод Вращений Якоби не применим - матрица не симметрична!')

exit(1)

else:

print('INFO: Метод Вращений Якоби применим!')

calculate\_results(input\_matrix, eps)



1.5) QR-разложение:

import numpy as np

import warnings

warnings.filterwarnings("ignore", category=np.exceptions.ComplexWarning)

def QR\_decomposition(matrix: np.ndarray) -> tuple[np.ndarray, np.ndarray]:

matrix\_size = matrix.shape[0]

matrix\_cpy = np.copy(matrix)

Q = np.eye(matrix\_size, dtype=float)

for i in range(matrix\_size):

v = np.zeros((matrix\_size, 1), dtype=float)

v[i] = matrix\_cpy[i][i] + np.sign(matrix\_cpy[i][i]) \* np.sqrt(

sum(

[matrix\_cpy[j][i] \*\* 2 for j in range(i, matrix\_size)]

)

)

for j in range(i + 1, matrix\_size):

v[j] = matrix\_cpy[j][i]

v\_transpose = np.transpose(v)

E = np.eye(matrix\_size, dtype=float)

H = E - 2 \* (v @ v\_transpose) / (v\_transpose @ v)

Q @= H

matrix\_cpy = H @ matrix\_cpy

return Q, matrix\_cpy

def QR\_algorithm(

matrix: np.ndarray,

epsilon: float =1e-10,

max\_iterations: int =1000

) -> np.ndarray:

A\_k = np.array(matrix, dtype=complex)

matrix\_size = matrix.shape[0]

for \_ in range(max\_iterations):

Q, R = QR\_decomposition(A\_k)

A\_k = R @ Q

if np.sum(np.abs(np.tril(A\_k, -1))) < epsilon:

break

eigenvalues = []

i = 0

while i < matrix\_size:

if i < matrix\_size - 1 and abs(A\_k[i + 1, i]) > epsilon:

a, b = A\_k[i, i], A\_k[i, i + 1]

c, d = A\_k[i + 1, i], A\_k[i + 1, i + 1]

trace = a + d

det = a \* d - b \* c

discr = trace \*\* 2 - 4 \* det

if discr < 0:

val1 = trace / 2 + 1j \* np.sqrt(-discr) / 2

val2 = trace / 2 - 1j \* np.sqrt(-discr) / 2

eigenvalues.extend([val1, val2])

else:

val1 = (trace + np.sqrt(discr)) / 2

val2 = (trace - np.sqrt(discr)) / 2

eigenvalues.extend([val1, val2])

i += 2

else:

eigenvalues.append(A\_k[i, i])

i += 1

return np.array(eigenvalues)

def check\_QR\_decomposition(

matrix: np.ndarray,

Q: np.ndarray,

R: np.ndarray,

eps: float = 0.001

) -> bool:

founded\_matrix = Q @ R

for i in range(matrix.shape[0]):

for j in range(matrix.shape[0]):

if abs(matrix[i, j] - founded\_matrix[i, j]) > eps:

return False

return True

if \_\_name\_\_ == '\_\_main\_\_':

n = int(input('Input matrix size (one positive number): '))

print('Input matrix:')

input\_matrix = np.empty((n, n), dtype=float)

for row in range(n):

input\_matrix[row] = np.array(list(map(

float,

input().split())))

eps = float(input('Input error rate: '))

Q, R = QR\_decomposition(input\_matrix)

print("Founded QR decomposition")

print("\nQ matrix:\n", Q)

print("\nR matrix:\n", R)

print("\nQR matrix:\n", Q @ R)

if check\_QR\_decomposition(input\_matrix, Q, R, eps):

print("\nQR decomposition founded correctly")

else:

print("\nERROR: QR decomposition is not correct")

exit(1)

eigenvalues = QR\_algorithm(input\_matrix, eps)

print('\nEigenvalues:')

for i, val in enumerate(eigenvalues):

if val.imag == 0:

print(f"λ\_{i + 1} = {val.real:.8f} {'+ 0.0i'}")

else:

print(f"λ\_{i + 1} = {val.real:.8f} {'+' if val.imag >= 0 else '-'} {abs(val.imag):.8f}i")

print()

print("Numpy eigenvalues (builtin function):")

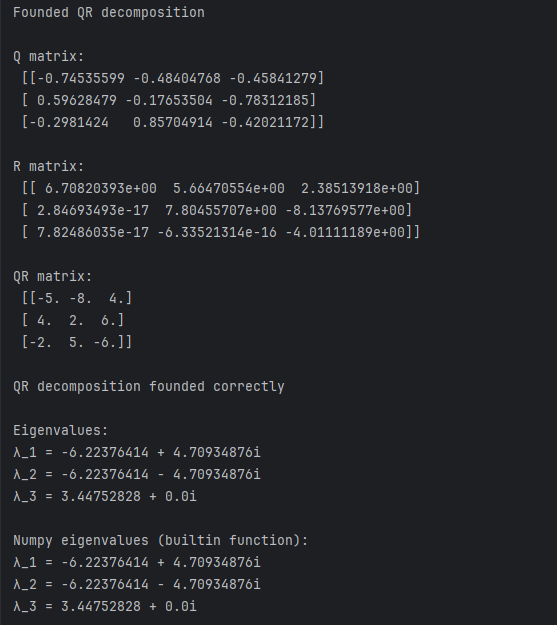
for i, val in enumerate(np.linalg.eig(input\_matrix).eigenvalues):

if val.imag == 0:

print(f"λ\_{i + 1} = {val.real:.8f} {'+ 0.0i'}")

else:

print(f"λ\_{i + 1} = {val.real:.8f} {'+' if val.imag >= 0 else '-'} {abs(val.imag):.8f}i")



# **Выводы**

В этой лабораторной работе рассматриваются численные методы решения систем линейных алгебраических уравнений (СЛАУ) и численные методы решения задач на собственные значения и собственные векторы матриц. Среди численных методов алгебры существуют прямые методы, в которых решение получается за конечное фиксированное число операций и итерационные методы, в которых результат достигается в процессе последовательных приближений.

В ходе лабораторной работы были реализованы методы LU-разложения матриц, с помощью которого решается СЛАУ, метод прогонки, метод простых итераций, метод Зейделя, метод вращения, QR-разложение и нахождение собственных значений с помощью QR разложения.