**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский Авиационный Институт»**

**(Национальный Исследовательский Университет)**

**Институт: №8 «Информационные технологии   
и прикладная математика»   
Кафедра: 806 «Вычислительная математика   
и программирование»**

Лабораторная работа №1   
по курсу «Численные методы»

Группа: М8О-307Б-22

Студент(ка): П. В. Лебедько

Преподаватель: Д. Л. Ревизников

Оценка:

Дата: 02.04.2025

Москва, 2025

**ОГЛАВЛЕНИЕ**

[**1** **Задание 1** 4](#_Toc194498711)

[**Задание** 4](#_Toc194498712)

[**Вариант** 4](#_Toc194498713)

[**Ход лабораторной работы** 4](#_Toc194498714)

[LU-разложение 4](#_Toc194498715)

[Решение СЛАУ с помощью LU-разложения 5](#_Toc194498716)

[Вычисление определителя с помощью LU-разложения 5](#_Toc194498717)

[Обращение матрицы с помощью LU-разложения 6](#_Toc194498718)

[Проверка LU-разложения 6](#_Toc194498719)

[Результаты 7](#_Toc194498720)

[**2** **Задание 2** 8](#_Toc194498721)

[**Задание** 8](#_Toc194498722)

[**Вариант** 8](#_Toc194498723)

[**Ход лабораторной работы** 8](#_Toc194498724)

[Восстановление матрицы 8](#_Toc194498725)

[Решение СЛАУ 8](#_Toc194498726)

[Результаты 9](#_Toc194498727)

[**3** **Задание 3** 9](#_Toc194498728)

[**Задание** 9](#_Toc194498729)

[**Вариант** 9](#_Toc194498730)

[**Ход лабораторной работы** 10](#_Toc194498731)

[Построение матриц alpha и beta 10](#_Toc194498732)

[Метод простых итераций 10](#_Toc194498733)

[Метод Зейделя 11](#_Toc194498734)

[Результаты 11](#_Toc194498735)

[**4** **Задание 4** 12](#_Toc194498736)

[**Задание** 12](#_Toc194498737)

[**Вариант** 12](#_Toc194498738)

[**Ход лабораторной работы** 12](#_Toc194498739)

[Метод вращения 12](#_Toc194498740)

[Результаты 13](#_Toc194498741)

[**5** **Задание 5** 14](#_Toc194498742)

[**Задание** 14](#_Toc194498743)

[**Вариант** 14](#_Toc194498744)

[**Ход лабораторной работы** 14](#_Toc194498745)

[QR-разложение 14](#_Toc194498746)

[QR-алгоритм 15](#_Toc194498747)

[Результаты 16](#_Toc194498748)

[**6** **Выводы** 17](#_Toc194498749)

# **Задание 1**

## **Задание**

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.

## **Вариант**

**Вариант 17**



## **Ход лабораторной работы**

### LU-разложение

def getLU(A):

    n = len(A)

    LU = np.copy(A)

    swaps = []

    for k in range(n): # обнуляемый столбец

        if (LU[k][k] == 0): # Ищем ненулевой элемент

            ind = -1

            for i in range(k + 1, n):

                if LU[i][i] != 0:

                    ind = i

                    break

            if ind == -1:

                continue

            LU[[k, ind]] = LU[[ind, k]] # Меняем местами строки если нашли строку с ненулевым элементом

            swaps.append((k, ind))

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

            mu = LU[i][k] / LU[k][k]

            for j in range(k, n): # текущая столбец

                if (j == k):

                    LU[i][j] = mu

                else:

                    LU[i][j] -= mu \* LU[k][j]

    return (LU, swaps)

Разложение будем реализовывать с проверкой дополнительного условия, что ведущий элемент отличен от нуля. В противном случае будем выполнять перестановки строк, которые будем запоминать в массиве.

Результирующие матрицы L и U можно хранить в одной матрице, так как одна из них нижне-треугольная, а другая – верхне-.

### Решение СЛАУ с помощью LU-разложения

def solverLU(LU, swaps, b):

    n = len(LU)

    b = np.copy(b)

    # Меняем строки в столбце свободных членов в соответствие с заменами строк в исходной матрице

    for swap in swaps:

        b[[swap[0], swap[1]]] = b[[swap[1], swap[0]]]

    # Lz = b

    z = np.zeros(n)

    for i in range(n):

        sum\_ = sum([LU[i][j] \* z[j] for j in range(i)])

        z[i] = b[i] - sum\_

    # Ux = z

    x = np.zeros(n)

    for i in range(n - 1, -1, -1):

        sum\_ = sum([LU[i][j] \* x[j] for j in range(n - 1, i, -1)])

        x[i] = (z[i] - sum\_) / LU[i][i]

    return x

На вход функции подаются матрицы L и U, массив перестановок строк, а также вектор правых частей.

### Вычисление определителя с помощью LU-разложения

def getDet(LU, swaps):

    n = len(LU)

    det = 1

    for i in range(n):

        det \*= LU[i][i]

    # Каждая замена строк исходной матрицы - смена знака у определителя

    if (len(swaps) % 2 == 1):

        det \*= -1

    return det

    return x

На вход функции так же подаются матрицы L и U и массив перестановок строк. На определитель будет влиять только четность количества перестановок строк.

### Обращение матрицы с помощью LU-разложения

def inv(LU, swaps):

    n = len(LU)

    A = []

    for i in range(n):

        A.append(solverLU(LU, swaps, np.array([(1 if j == i else 0) for j in range(n)])))

    return np.column\_stack(A)

Задачу обращения матрицы сводим к решению n штук СЛАУ, векторы правых частей в которой – столбцы единичной матрицы.

### Проверка LU-разложения

def checkLU(LU):

    n = len(LU)

    L = np.copy(LU)

    for i in range(n):

        L[i][i] = 1

        for j in range(i + 1, n):

            L[i][j] = 0

    U = np.copy(LU)

    for i in range(1, n):

        for j in range(i):

            U[i][j] = 0

    print("Матрица L:\n", L)

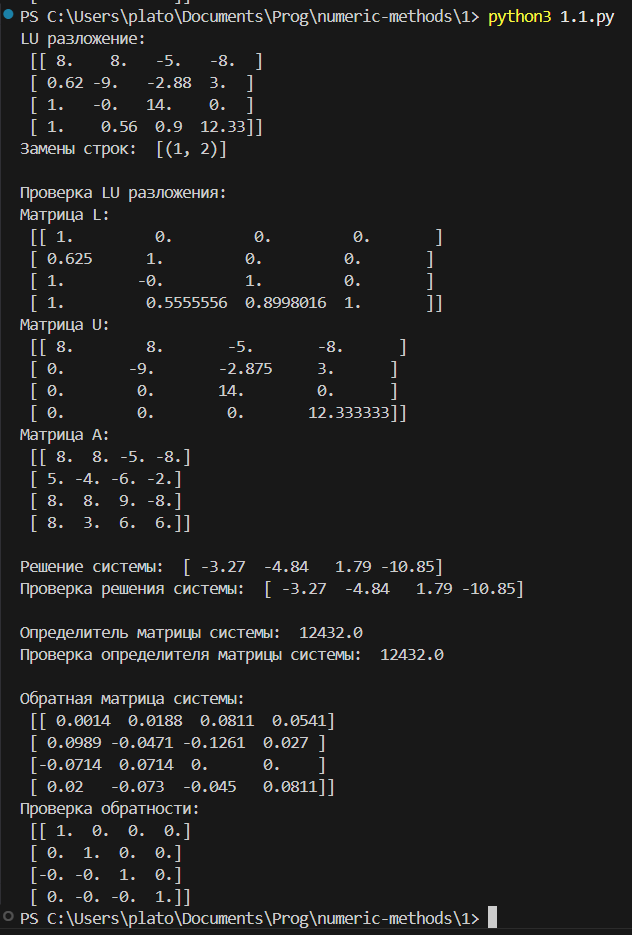
    print("Матрица U:\n", U)

    A = np.dot(L, U)

    print("Матрица A:\n", np.round(A, 2))

Для проверки воспользуемся определением матриц L и U, их произведение должно давать матрицу A.

### Результаты



Решение СЛАУ, нахождение определителя и обратной матрицы было проверено через функции numpy в python.

# **Задание 2**

## **Задание**

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.

## **Вариант**

**Вариант 17**



## **Ход лабораторной работы**

### Восстановление матрицы

def recoverMatrix(A):

    n = len(A)

    B = np.zeros((n, n))

    for i in range(n):

        if (i == 0):

            B[i][0] = A[i][1]

            B[i][1] = A[i][2]

        elif (i == n - 1):

            B[i][n - 2] = A[i][0]

            B[i][n - 1] = A[i][1]

        else:

            B[i][i - 1] = A[i][0]

            B[i][i] = A[i][1]

            B[i][i + 1] = A[i][2]

    return B

Реализуем функцию для восстановления матрицы nxn из компактного вида хранения матрицы, где мы храним только две диагонали.

### Решение СЛАУ

Реализованный алгоритм:

def tridiagonalMatrixAlgorithm(A, b):

    n = len(b)

    # Вычисляем прогоночные коэффициенты

    P = np.empty((n))

    P[0] = -A[0][2] / A[0][1]

    Q = np.empty((n))

    Q[0] = b[0] / A[0][1]

    for i in range(n):

        P[i] = (-A[i][2]) / (A[i][1] + A[i][0] \* P[i - 1])

        Q[i] = (b[i] - A[i][0] \* Q[i - 1]) / (A[i][1] + A[i][0] \* P[i - 1])

    # Обратный ход

    x = np.empty((n))

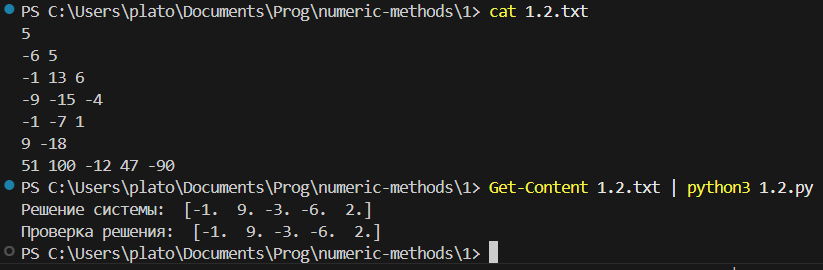
    x[n - 1] = Q[n - 1]

    for i in range(n - 2, -1, -1):

        x[i] = P[i] \* x[i + 1] + Q[i]

    return x

### Результаты



Проверка решения так же осуществляется через встроенную функцию np.linalg.solve.

# **Задание 3**

## **Задание**

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.

## **Вариант**

**Вариант 17**



## **Ход лабораторной работы**

### Построение матриц alpha и beta

def buildMatrixAlphaAndBeta(A, b):

    n = len(b)

    alpha = np.empty((n, n))

    beta = np.empty(n)

    for i in range(n):

        beta[i] = b[i] / A[i][i]

        for j in range(n):

            alpha[i][j] = 0 if i == j else -A[i][j] / A[i][i]

    return alpha, beta

Для работы метода простых итераций необходимо отдельно построить матрицы alpha и beta из исходной матрицы.

### Метод простых итераций

def FixedPointIteration(alpha, beta, eps):

    prevX = np.copy(beta)

    iter = 0

    norm = np.linalg.norm(alpha, ord=NORM\_ORD)

    while (True):

        iter += 1

        curX = np.dot(alpha, prevX) + beta

        if norm < 1:

            curEps = norm / (1 - norm) \* np.linalg.norm(curX - prevX)

        else:

            curEps = np.linalg.norm(curX - prevX)

        prevX = curX

        if (curEps < eps):

            break

    return prevX, iter

Метод принимает на вход матрицы alpha и beta, а также точность вычислений eps. Реализуется дополнительное сравнение нормы матрицы alpha с единицей, что повлияет на способ вычисления текущей погрешности.

В качестве нормы используется бесконечная норма:

NORM\_ORD = np.inf

### Метод Зейделя

Метод Зейделя заключается лишь в том, что на каждой итерации мы используем информацию о вычисленных уже на этой итерации значениях. Таким образом, метод Зейделя сведется к методу простых итераций с построенными по-другому матрицами alpha и beta.

def Seidel(A, b, eps):

    alpha, beta = buildMatrixAlphaAndBeta(A, b)

    B = np.zeros((n, n))

    C = np.zeros((n, n))

    for i in range(n):

        for j in range(n):

            if i > j:

                B[i][j] = alpha[i][j]

            else:

                C[i][j] = alpha[i][j]

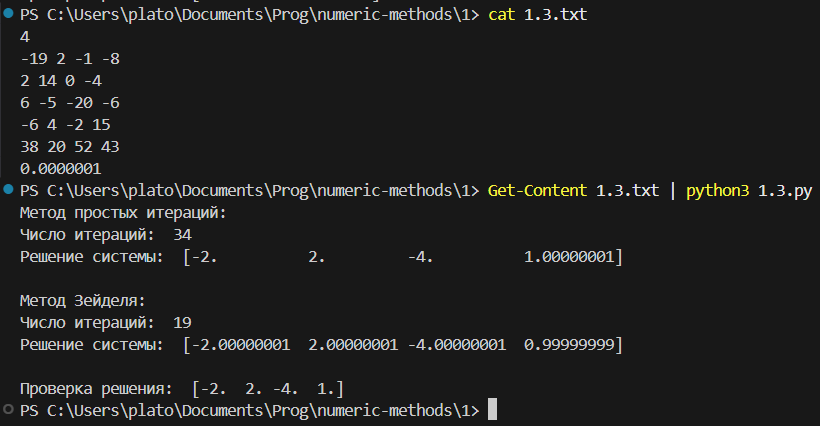
    tmp = np.linalg.inv((np.eye(n) - B))

    newAplha = np.dot(tmp, C)

    newBeta = np.dot(tmp, beta)

    return FixedPointIteration(newAplha, newBeta, eps)

### Результаты



Проверка решения так же осуществляется через встроенную функцию np.linalg.solve.

Заметно, что метод Зейделя сходится быстрее метода простых итераций.

# **Задание 4**

## **Задание**

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.

## **Вариант**

**Вариант 17**



## **Ход лабораторной работы**

### Метод вращения

def rotationMethod(A, eps):

    n = len(A)

    A = np.copy(A)

    U = np.eye(n)

    iter = 0

    while (True):

        iter += 1

        # Выбрали максимальный элемент

        I, J = -1, -1

        max\_ = 0

        for i in range(n):

            for j in range(i + 1, n):

                if abs(A[i][j]) > max\_:

                    I, J = i, j

                    max\_ = abs(A[i][j])

        # Строим Uk

        Uk = np.eye(n)

        phi = 0.5 \* np.atan((2 \* A[I][J]) / (A[I][I] - A[J][J]))

        Uk[I][I] = np.cos(phi)

        Uk[I][J] = -np.sin(phi)

        Uk[J][I] = np.sin(phi)

        Uk[J][J] = np.cos(phi)

        # Строим U

        U = np.dot(U, Uk)

        # Строим Ak

        A = np.dot(np.dot(np.transpose(Uk), A), Uk)

        # Проверяем критерий окончания

        t = 0

        for i in range(n):

            for j in range(i + 1, n):

                t += A[i][j] \*\* 2

        t = np.sqrt(t)

        if (t < eps):

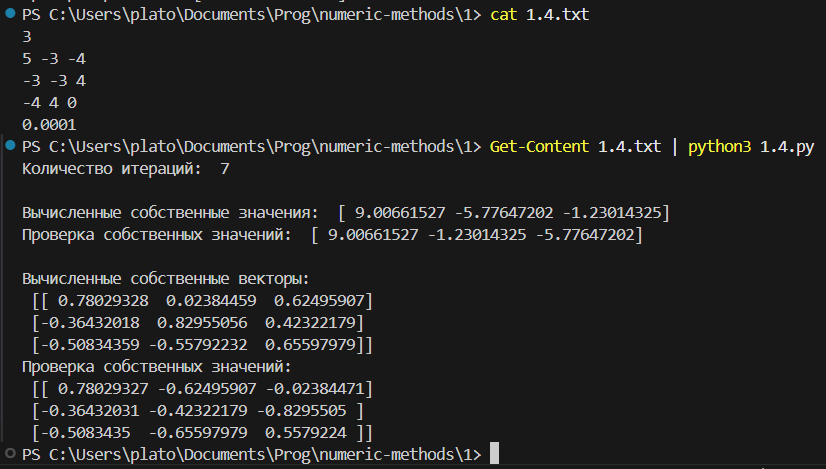
            break

    lambdas = np.array([A[i][i] for i in range(n)])

    return lambdas, U, iter

При вводе матрицы реализуется дополнительная проверка на ее симметричность, затем реализуется стандартный алгоритм метода вращения.

### Результаты



Проверка собственных значений и векторов осуществляется через встроенную функцию np.linalg.eig.

Собственные векторы определяются с точностью до константы, а набор собственных значений – с точностью до перестановок.

# **Задание 5**

## **Задание**

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.

## **Вариант**

**Вариант 17**



## **Ход лабораторной работы**

### QR-разложение

def decompositionQR(A):

    n = len(A)

    A = np.copy(A)

    Q = np.eye(n)

    for i in range(n):

        # Строим вектор v

        v = np.zeros((n, 1))

        v[i] = A[i][i] + np.sign(A[i][i]) \* np.sqrt(sum( [A[j][i] \*\* 2 for j in range(i, n)] ))

        for j in range(i + 1, n):

            v[j] = A[j][i]

        # Строим матрицу Хаусхолдера

        vTrans = np.transpose(v)

        H = np.eye(n) - 2 / np.dot(vTrans, v) \* np.dot(v, vTrans)

        # Строим Q

        Q = np.dot(Q, H)

        # Строим Ak

        A = np.dot(H, A)

    return Q, A

### QR-алгоритм

Реализация алгоритма:

def alrorithmQR(A, eps):

    n = len(A)

    A = np.copy(A)

    iter = 0

    lambdas = np.empty((n, 2))

    while (True):

        iter += 1

        Q, R = decompositionQR(A)

        A = np.dot(R, Q)

        flg = True

        skip = False

        # print(f"iter #{iter}")

        for i in range(n):

            if skip:

                skip = False

                continue

            if i < n - 1:

                D = A[i][i] \*\* 2 + A[i + 1][i + 1] \*\* 2 - 2 \* A[i][i] \* A[i + 1][i + 1] + 4 \* A[i][i + 1] \* A[i + 1][i]

                if D < 0:

                    re = (A[i][i] + A[i + 1][i + 1]) / 2

                    im = np.sqrt(-D) / 2

                    # Критерий остановки для пары комплексно-сопряженных

                    lambda\_ = np.sqrt(re \*\* 2 + im \*\* 2)

                    lambdaPrev = np.sqrt(lambdas[i][0] \*\* 2 + lambdas[i][1] \*\* 2)

                    # print(f"coord #{i}: abs(lambda\_ - lambdaPrev) = {abs(lambda\_ - lambdaPrev)}")

                    if iter > 1 and abs(lambda\_ - lambdaPrev) > eps:

                        flg = False

                    lambdas[i][0] = re

                    lambdas[i][1] = im

                    lambdas[i + 1][0] = re

                    lambdas[i + 1][1] = -im

                    skip = True

                    continue

            lambdas[i][0] = A[i][i]

            lambdas[i][1] = 0

            # Критерий остановки для действительного значения

            sum\_ = np.sqrt(sum([A[j][i] \*\* 2 for j in range(i + 1, n)]))

            # print(f"coord #{i}: sum = {sum\_}")

            if sum\_ > eps:

                flg = False

        if flg:

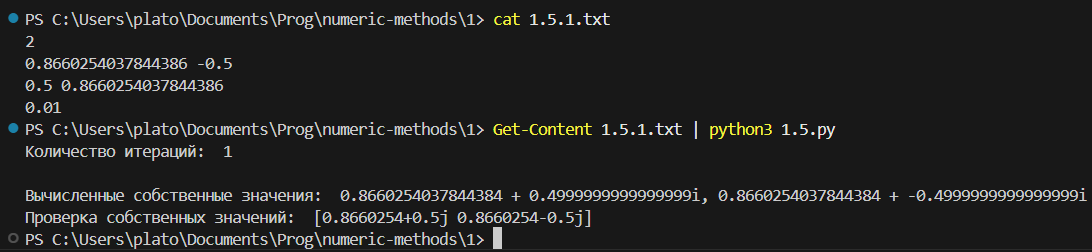
            break

    return lambdas, iter

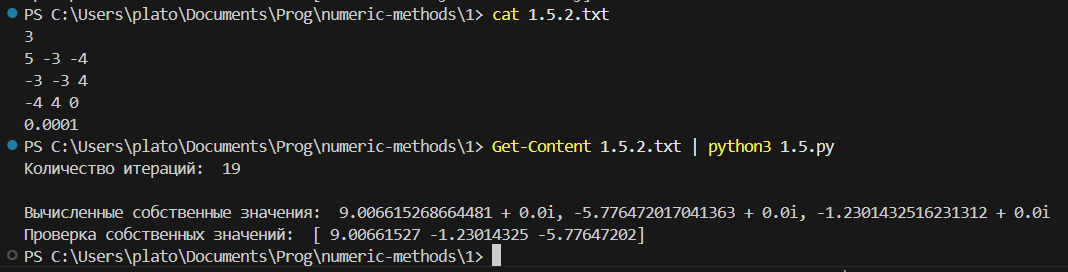
В ходе работы поддерживается массив текущих собственных значений. На каждой итерации происходит проверка каждого диагонального элемента матрицы – является ли он единичным действительным собственным значением или, вместе со следующим диагональным элементом, образует пару комплексно-сопряженных корней. Этот вывод производится на основе знака дискриминанта квадратного уравнения, определяющего пару собственных значений. Каждое собственное значение хранит отдельно действительную и мнимую части.

### Результаты

Первый набор входных данных – матрица поворота на 30 градусов – ее собственные значения комплексные:



Второй набор – матрица из варианта:



Проверка собственных значений и векторов осуществляется через встроенную функцию np.linalg.eig.

Набор собственных значений определяется с точностью до перестановок.

# **Выводы**

В ходе лабораторной работы я реализовал стандартные алгоритмы для решения СЛАУ, такие как, LU-разложение, метод прогонки, метод простых итераций и Зейделя, а также алгоритмы для поиска собственных векторов и значений, такие как, метод вращения и QR-алгоритм, решающий полную проблему собственных значений.