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

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

$$\begin{cases} 8\cdot x_1 + 8\cdot x_2 - 5\cdot x_3 - 8\cdot x_4 = 13 \\ 8\cdot x_1 - 5\cdot x_2 + 9\cdot x_3 - 8\cdot x_4 = 38 \\ 5\cdot x_1 - 4\cdot x_2 - 6\cdot x_3 - 2\cdot x_4 = 14 \\ 8\cdot x_1 + 3\cdot x_2 + 6\cdot x_3 + 6\cdot x_4 = -95 \end{cases}$$

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

```
PS C:\Users\plato\Documents\Prog\numeric-methods\1> python3 1.1.py
LU разложение:
 [[ 8.    8.    -5.    -8.   ]
 [ 0.62 -9.    -2.88  3.   ]
 [ 1.   -0.    14.    0.   ]
 [ 1.    0.56  0.9   12.33]]
Замены строк:  [(1, 2)]

Проверка LU разложения:
Матрица L:
 [[ 1.         0.         0.         0.       ]
 [ 0.625      1.         0.         0.       ]
 [ 1.        -0.         1.         0.       ]
 [ 1.         0.5555556  0.8998016  1.       ]]
Матрица U:
 [[ 8.         8.        -5.        -8.      ]
 [ 0.        -9.        -2.875      3.      ]
 [ 0.         0.        14.         0.      ]
 [ 0.         0.         0.        12.333333]]
Матрица A:
 [[ 8.  8. -5. -8.]
 [ 5. -4. -6. -2.]
 [ 8.  8.  9. -8.]
 [ 8.  3.  6.  6.]]

Решение системы:  [ -3.27  -4.84   1.79 -10.85]
Проверка решения системы:  [ -3.27  -4.84   1.79 -10.85]

Определитель матрицы системы:  12432.0
Проверка определителя матрицы системы:  12432.0

Обратная матрица системы:
 [[ 0.0014  0.0188  0.0811  0.0541]
 [ 0.0989 -0.0471 -0.1261  0.027 ]
 [-0.0714  0.0714  0.      0.    ]
 [ 0.02   -0.073  -0.045   0.0811]]
Проверка обратности:
 [[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [-0. -0.  1.  0.]
 [ 0. -0. -0.  1.]]
PS C:\Users\plato\Documents\Prog\numeric-methods\1>
```

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

2 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

$$\begin{cases} -6 \cdot x_1 + 5 \cdot x_2 = 51 \\ -x_1 + 13 \cdot x_2 + 6 \cdot x_3 = 100 \\ -9 \cdot x_2 - 15 \cdot x_3 - 4 \cdot x_4 = -12 \\ -x_3 - 7 \cdot x_4 + x_5 = 47 \\ 9 \cdot x_4 - 18 \cdot x_5 = -90 \end{cases}$$

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 $n_x n$ из компактного вида хранения матрицы, где мы храним только две диагонали.

Решение СЛАУ

Реализованный алгоритм:

```
def tridiagonalMatrixAlgorithm(A, b):
    n = len(b)
```

```python
    # 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

```
PS C:\Users\plato\Documents\Prog\numeric-methods\1> cat 1.2.txt
5
-6 5
-1 13 6
-9 -15 -4
-1 -7 1
9 -18
51 100 -12 47 -90
PS C:\Users\plato\Documents\Prog\numeric-methods\1> Get-Content 1.2.txt | python3 1.2.py
Решение системы:  [-1.  9. -3. -6.  2.]
Проверка решения:  [-1.  9. -3. -6.  2.]
PS C:\Users\plato\Documents\Prog\numeric-methods\1>
```

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

3 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

$$\begin{cases} -19 \cdot x_1 + 2 \cdot x_2 - x_3 - 8 \cdot x_4 = 38 \\ 2 \cdot x_1 + 14 \cdot x_2 - 4 \cdot x_4 = 20 \\ 6 \cdot x_1 - 5 \cdot x_2 - 20 \cdot x_3 - 6 \cdot x_4 = 52 \\ -6 \cdot x_1 + 4 \cdot x_2 - 2 \cdot x_3 + 15 \cdot x_4 = 43 \end{cases}$$

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.

```python
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

```
PS C:\Users\plato\Documents\Prog\numeric-methods\1> cat 1.3.txt
4
-19 2 -1 -8
2 14 0 -4
6 -5 -20 -6
-6 4 -2 15
38 20 52 43
0.0000001
PS C:\Users\plato\Documents\Prog\numeric-methods\1> Get-Content 1.3.txt | python3 1.3.py
Метод простых итераций:
Число итераций:  34
Решение системы:  [-2.          2.         -4.          1.00000001]

Метод Зейделя:
Число итераций:  19
Решение системы:  [-2.00000001  2.00000001 -4.00000001  0.99999999]

Проверка решения:  [-2.  2. -4.  1.]
PS C:\Users\plato\Documents\Prog\numeric-methods\1>
```

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

Заметно, что метод Зейделя сходится быстрее метода простых итераций.

4 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

$$\begin{pmatrix} 5 & -3 & -4 \\ -3 & -3 & 4 \\ -4 & 4 & 0 \end{pmatrix}$$

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)
```

```python
        # 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

```
PS C:\Users\plato\Documents\Prog\numeric-methods\1> cat 1.4.txt
3
5 -3 -4
-3 -3 4
-4 4 0
0.0001
PS C:\Users\plato\Documents\Prog\numeric-methods\1> Get-Content 1.4.txt | python3 1.4.py
Количество итераций:  7

Вычисленные собственные значения:  [ 9.00661527 -5.77647202 -1.23014325]
Проверка собственных значений:  [ 9.00661527 -1.23014325 -5.77647202]

Вычисленные собственные векторы:
 [[ 0.78029328  0.02384459  0.62495907]
 [-0.36432018  0.82955056  0.42322179]
 [-0.50834359 -0.55792232  0.65597979]]
Проверка собственных значений:
 [[ 0.78029327 -0.62495907 -0.02384471]
 [-0.36432031 -0.42322179 -0.8295505 ]
 [-0.5083435  -0.65597979  0.5579224 ]]
PS C:\Users\plato\Documents\Prog\numeric-methods\1>
```

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.

5 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

$$\begin{pmatrix} -6 & 1 & -4 \\ -6 & 8 & -2 \\ 2 & -9 & 5 \end{pmatrix}$$

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:

```
PS C:\Users\plato\Documents\Prog\numeric-methods\1> cat 1.5.1.txt
2
0.8660254037844386 -0.5
0.5 0.8660254037844386
0.01
PS C:\Users\plato\Documents\Prog\numeric-methods\1> Get-Content 1.5.1.txt | python3 1.5.py
Количество итераций:  1

Вычисленные собственные значения:  0.8660254037844384 + 0.4999999999999999i, 0.8660254037844384 + -0.4999999999999999i
Проверка собственных значений:  [0.8660254+0.5j 0.8660254-0.5j]
PS C:\Users\plato\Documents\Prog\numeric-methods\1>
```

Второй набор – матрица из варианта:

```
PS C:\Users\plato\Documents\Prog\numeric-methods\1> cat 1.5.2.txt
3
5 -3 -4
-3 -3 4
-4 4 0
0.0001
PS C:\Users\plato\Documents\Prog\numeric-methods\1> Get-Content 1.5.2.txt | python3 1.5.py
Количество итераций:  19

Вычисленные собственные значения:  9.006615268664481 + 0.0i, -5.776472017041363 + 0.0i, -1.2301432516231312 + 0.0i
Проверка собственных значений:  [ 9.00661527 -1.23014325 -5.77647202]
PS C:\Users\plato\Documents\Prog\numeric-methods\1> 
```

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

Набор собственных значений определяется с точностью до перестановок.

6 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.