# Лабораторная работа №1

## Содержание

# Лабораторная работа 1.1

## LU-разложение матриц. Метод Гаусса

**Задача**: 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.

**СЛАУ**:

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

**Решение**: *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 $i>j$).

*LU* – разложение может быть построено с использованием метода Гаусса.

В методе Гаусса матрица СЛАУ с помощью равносильных преобразований преобразуется в верхнюю треугольную матрицу, получающуюся в результате прямого хода. В обратном ходе определяются неизвестные.

**Работа программы**:

```
Начальная матрица:
  8    8    −5    −8

  8    −5    9    −8

  5    −4    −6    −2

  8    3    6    6
Сопряженная матрица :

  330    234    1242    1166

  1230    −586    −1568    336
```

```
  -290    888    -1456    312

  -765    -907    584    1307

Обратная матрица :

  0.01646    0.01167    0.06195    0.05815

  0.06135    -0.02923    -0.0782    0.01676

  -0.01446    0.04429    -0.07262    0.01556

  -0.03815    -0.04524    0.02913    0.06519

Матрица A после LUP преобразования:
  8    8    -5    -8

  1.0    -13.0    14.0    0.0

  0.625    0.692    -12.567    3.0

  1.0    0.385    -0.447    15.34

матрица l:
  1    0    0    0

  1.0    1    0    0

  0.625    0.692    1    0

  1.0    0.385    -0.447    1

матрица u:
  8    8    -5    -8

  0    -13.0    14.0    0.0

  0    0    -12.567    3.0

  0    0    0    15.34
```

Проверяем, что разложение верное, перемножая L и U: A = LU

a:

8    8.0    -5.0    -8.0

8.0    -5.0    9.0    -8.0

5.0    -4.0    -6.0    -2.0

8.0    3.0    6.0    6.0

Результат решения слау:
[-4.0, -3.0, -1.0, -8.0]

**Листинг программы**:

```
def getCofactor(a, tmp, p, q, n):
  i = 0
  j = 0
  #копируем в новую матрицу все элементы исходной без строки и столбца
  for row in range(n):
    for col in range(n):
      if row != p and col != q:
        tmp[i][j] = a[row][col]
        j += 1
        #переход к первому столбцу и на следующую строку
        if j == n - 1:
          j = 0
          i += 1

def determinant(a, n):
    d = 0
    if n == 1:
        return a[0][0]
```

```
    tmp = [[0] * n for _ in range(n)]  # Cofactors
    sign = 1  # коэффициент при слагаемом (-1)^k

    for i in range(n):
        getCofactor(a, tmp, 0, i, n)
        d = d + sign * a[0][i] * determinant(tmp, n -
1)

        sign = -sign

    return d

def adjoin(a, n):  # Сопряженная матрица
    adj = [[0] * n for _ in range(n)]
    if n == 1:
        adj[0][0] = 1
        return

    tmp = [[0] * n for _ in range(n)]  # Cofactors
    for i in range(n):
        for j in range(n):
            getCofactor(a, tmp, i, j, n)  # Cofactor
a[i][j]

            # sign = (-1)**(i+j)
            if (i + j) % 2 == 0:
                sign = 1
            else:
                sign = -1

            adj[j][i] = sign * (determinant(tmp, n -
1))  # (-1^(i+j)*|алгебр дополнение|)

    return adj

def inverse(a, b, n):
    det = determinant(a, n)
    if det == 0:
        print("Матрица вырождена")
        return False
```

```
    adj = adjoin(a, n)

    for i in range(n):
        for j in range(n):
            b[i][j] = adj[i][j] / det

    return True

def transpose(a, n):
    b = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            b[i][j] = a[j][i]
    return b

def multi(M1, M2):
    sum = 0
    tmp = []
    ans = []

    row1 = len(M1)
    col1 = len(M1[0])
    row2 = col1
    col2 = len(M2[0])
    for k in range(0, row1):
        for j in range(0, col2):
            for i in range(0, col1):
                sum = round(sum + M1[k][i] * M2[i]
[j], 8)
            tmp.append(sum)
            sum = 0
        ans.append(tmp)
        tmp = []
    return ans

def lup_solve(l, u, pi, b, n):
    x = [0 for i in range(n)]
    y = [0 for i in range(n)]
```

```
    for i in range(n):
        summ = 0
        for j in range(i):
            summ += l[i][j] * y[j]

        y[i] = b[pi[i]] - summ

    for i in range(n - 1, -1, -1):
        sec_summ = 0
        for j in range(i + 1, n):
            sec_summ += u[i][j] * x[j]

        x[i] = (y[i] - sec_summ) / u[i][i]

    x = [round(x[i], 5) for i in range(len(x))]
    return x

def lupdecompose(a, n):
    pi = [i for i in range(n)]

    for k in range(n):
        p = 0
        for i in range(k, n):
            if abs(a[i][k]) > p:
                p = abs(a[i][k])
                tmp_k = i

        pi[k], pi[tmp_k] = pi[tmp_k], pi[k]

        for i in range(n):
            a[k][i], a[tmp_k][i] = a[tmp_k][i], a[k]
[i]
        for i in range(k + 1, n):
            a[i][k] = a[i][k] / a[k][k]
            for j in range(k + 1, n):
                a[i][j] = a[i][j] - a[i][k] * a[k][j]
    return pi

def get_lu(a):
    n = len(a)
```

```
    l = [[0] * n for i in range(0, n)]
    u = [[0] * n for i in range(0, n)]

    for i in range(n):
        l[i][i] = 1
        for j in range(n):
            if j < i:
                l[i][j] = a[i][j]
            else:
                u[i][j] = a[i][j]
    return l, u

def roundMatrix(a, after):
  retVal = [[0] * len(a) for _ in range(len(a[0]))]

  for i in range(0,len(a)):
    for j in range(0,len(a[0])):
      retVal[i][j] = round(a[i][j],after)

  return retVal

def show(a, n):
    for i in range(0, n):
        for j in range(0, n):
            print(" ", a[i][j], " ", end="")
        print("\n")
```

# Лабораторная работа 1.2

## Метод прогонки

**Задача**: 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.

**СЛАУ**:

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

**Решение**: 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.

Общее число операций в методе прогонки равно $8n + 1$, т.е. пропорционально числу уравнений. Такие методы решения СЛАУ называют экономичными. Для сравнения число операций в методе Гаусса пропорционально $n^3$

**Работа программы**:

```
Матрица A
[[-6, 5, 0, 0, 0], [-1, 13, 6, 0, 0], [0, -9, -15, -4, 0], [0, 0, -1, -7, 1], [0, 0, 0, 9, -18]]
Вектор b
[51, 100, -12, 47, -90]
Решение
[-1.  9. -3. -6.  2.]
```

**Листинг программы**:

```python
import numpy as np

def runmethod(c, f, size):
    cn = np.zeros(size)
    fn = np.zeros(size)
    yn = np.zeros(size)
```

```
    xn = np.zeros(size)

    for i in range(0, size):
        if i == 0:
            yn[i] = c[i][i]
            cn[i] = -c[i][i + 1] / yn[i]
            fn[i] = f[i] / yn[i]
        elif i == size - 1:
            yn[i] = c[i][i] + c[i][i-1] * cn[i-1]
            fn[i] = (f[i] - c[i][i-1] * fn[i-1]) /
yn[i]
        else:
            yn[i] = c[i][i] + c[i][i-1] * cn[i-1]
            cn[i] = -c[i][i+1] / yn[i]
            fn[i] = (f[i] - c[i][i-1] * fn[i-1]) /
yn[i]

    xn[size-1] = fn[size-1]
    i = size-2
    while i >= 0:
        xn[i] = cn[i] * xn[i+1] + fn[i]
        i = i - 1

    return xn
```

# Лабораторная работа 1.3

## Итерационные методы решения СЛАУ

**Задача**: 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.

**СЛАУ**:

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

**Решение**: При большом числе уравнений прямые методы решения СЛАУ (за исключением метода прогонки) становятся труднореализуемыми на ЭВМ прежде всего из-за сложности хранения и обработки матриц большой размерности. В то же время характерной особенностью ряда часто встречающихся в прикладных задачах СЛАУ является разреженность матриц. Число ненулевых элементов таких матриц мало по сравнению с

их размерностью. Для решения СЛАУ с разреженными матрицами предпочтительнее использовать итерационные методы.

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

СЛАУ $Ax = b$ приводят к эквивалентному виду $x = \beta + \alpha x$. Получают следующие выражения для компонентов вектора $\beta$ и матрицы $\alpha: \beta_i = \dfrac{b_i}{a_{ii}}; \alpha_{ij} = -\dfrac{a_{ij}}{a_{ii}}, i, j = \overline{1,n}, i \neq j.$

**Работа программы**:

```
Метод Зейделя:
Iteration 0 ans  [-2.0, 1.4285714285714286, -2.6,
2.8666666666666667]
Iteration 1 ans  [-2.919799498746867, 2.664733261725743,
-5.002123165055496, 0.3212015753669893]
Iteration 2 ans  [-1.5914748902331017,
1.7476968629952971, -3.610727155428852,
1.2825939263841666]
```

```
Iteration 3 ans  [-2.166033185771257, 2.1044601483627985,
-4.160703170737326, 0.8843035966964405]
Iteration 4 ans  [-1.9318318582162424,
1.9572055788013034, -3.934142031174131,
1.0474601648766046]
Iteration 5 ans  [-2.027954112117689, 2.0175534916958426,
-4.027012656022249, 0.9805357365643997]
Iteration 6 ans  [-1.9885350659526981,
1.9928009341544997, -3.988921474293754,
1.007982861271887]
Iteration 7 ans  [-2.004702081451176, 2.00295254342785,
-4.004543618673881, 0.9967260066822523]
Iteration 8 ans  [-1.9980715446278126, 1.998789079713188,
-3.9981365353213163, 1.0013427561825161]
Iteration 9 ans  [-2.000790913405918, 2.0004966322529927,
-4.000764258939778, 0.9994492981781978]
Iteration 10 ans  [-1.9996756243147271,
1.9997963172387319, -3.9996865560575605,
1.0002258582027725]

Метод простых итераций:
Решение найдено на итерации 16:
[-2.0000368098640267, 1.9999870625641294,
-3.9999302593776025, 1.0001197301739433]
```

**Листинг программы**:

```python
import copy
from math import sqrt
import numpy as np

def seidel(a, b, eps):
    n = len(a)
    beta = [0 for _ in range(n)]

    for i in range(n):
        beta[i] = b[i] / a[i][i]

    x = [beta[i] for i in range(n)]

    stop = False
    count = 0
    while not stop:
```

```
        x_new = copy.deepcopy(x)
        print("Iteration", count, "ans ", x)
        for i in range(n):
            tmp_1 = 0
            tmp_2 = 0
            for j in range(i):
                tmp_1 += a[i][j] * x_new[j] # Sum for
x_(k+1)
            for j in range(i+1, n):
                tmp_2 += x[j] * a[i][j] # Sum for x_k

            x_new[i] = (b[i] - tmp_1 - tmp_2) / a[i]
[i]

        stop = sqrt(sum((x_new[i] - x[i]) ** 2 for i
in range(n))) <= eps
        if not stop:
            x = x_new
        count = count + 1

    x = [round(x[i], 4) for i in range(n)]
    return x

def simpleIteration(a, b, eps):
    n = len(b)
    beta = np.zeros(n)
    alpha = np.zeros((n, n))
    x = np.zeros(n)
    dx = np.zeros(n)
    #приводим систему к эквивалентному виду
    for i in range(n):
        beta[i] = b[i] / a[i][i]
        for j in range(n):
            if i != j:
                alpha[i][j] = - a[i][j] / a[i][i]
            else:
                alpha[i][j] = 0
        x[i] = beta[i]
        dx[i] = beta[i]
```

```
    # запускаем процесс, пока не выполнено условие
останова
    norm = 1
    norma_a = m_norm(n, alpha)
    count = 0
    while abs(norm) > eps:
        count = count + 1
        x = mv_mult(alpha, x)
        x = v_sum(x, beta)
        dx = v_minus(dx, x)
        norm = norma_a * v_norm(dx)/(1-norma_a)
        dx = x

    print(f"Решение найдено на итерации {count-1}:")
    print(x)

def matrixsum(a, b):
    out = [[0] * len(a[0]) for _ in range(len(a))]
    for i in range(len(a)):
        for j in range(len(a[0])):
            out[i][j] = a[i][j] + b[i][j]
    return out

def mv_mult(matrix, vector):
    size = len(vector)
    r = np.zeros(size)
    for i in range(size):
        for j in range(size):
            r[i] = r[i] + matrix[i][j] * vector[j]
    return r

def norma(x, y):
    return sqrt(sum((x[i] - y[i]) ** 2 for i in
range(len(x))))


def m_norm(size, matrix):
    v = [abs(sum(matrix[i][j] for i in range(size)))
for j in range(size)]
    return max(v)
```

```python
def v_norm(vec):
    size = len(vec)
    return sum(abs(vec[i]) for i in range(size))

def v_sum(x, y):
    return [x[i] + y[i] for i in range(len(x))]


def v_minus(x, y):
    return [x[i] - y[i] for i in range(len(x))]

def show(a, n):
    for i in range(0, n):
        for j in range(0, n):
            print("\t", a[i][j], " ", end='')
        print("\n")


eps = 0.001
a = [[-19, 2, -1, -8],
     [2, 14, 0, -4],
     [6, -5, -20, -6],
     [-6, 4, -2, 15]]
b = [38, 20, 52, 43]
print("Метод Зейделя:")
seidel(a, b, eps)
print("\nМетод простых итераций:")
simpleIteration(a, b, eps)
```

# Лабораторная работа 1.4

## Метод вращений

**Задача**: 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.

**Матрица**:

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

**Решение**:

Метод вращений Якоби применим только для симметрических матриц $A_{nxn}(A = A^T)$ и решает полную проблему собственных значений и собственных векторов таких матриц. Он основан на отыскании с помощью итерационных процедур матрицы $U$ в преобразовании подобия $\Lambda = U^{-1}AU$, а поскольку для симметрических матриц $A$ матрица преобразования подобия $U$ является ортогональной ($U^{-1} = U^T$), то $\Lambda = U^T AU$, где $\Lambda$ - диагональная матрица с собственными значениями на главной диагонали.

**Работа программы**:

```
Симметрическая матрица:
     5   -3        -4

     -3        -3        4

     -4        4   0

x:
 [9.006611652402118, -5.776463605290808,
-1.2301480471132926]
u:

     0.7799249769051313   -0.024755149822928618
-0.6253832528589707

     -0.36463734680800086      -0.8300911992943908
-0.42188648493005576
```

```
     -0.5086812712416123      0.5570778970764281
-0.6564355115901949

Проверка с помощью linalg:
x:
 [ 9.00661527 -1.23014325 -5.77647202]
u:

     0.7802932741046164    -0.6249590674733193
-0.023844713669425758

     -0.36432031279313665      -0.42322179068094057
-0.8295504960996223

     -0.5083435020470007       -0.655979786179191
0.5579223996676363
```

**Листинг программы**:

```
import numpy as np
import math

"""Лабораторная работа 1, задача 4, вариант 17"""

def rotation(a, eps=0.01):
    n = len(a)
    ak = [row.copy() for row in a]

    u = [[0. if i != j else 1. for i in range(n)] for
j in range(n)]

    cov = False

    while not cov:
        ik, jk = 0, 1

        # Выбирается максимальный по модулю
недиагональный элемент
        for i in range(n - 1):
            for j in range(i + 1, n):
                if abs(ak[i][j]) > abs(ak[ik][jk]):
                    ik, jk = i, j
```

```
        if ak[ik][ik] == ak[jk][jk]:
            phi = math.pi / 4
        else:
            phi = math.atan(2 * a[ik][jk] / (a[ik][ik] - a[jk][jk])) * 0.5

        uk = [[0. if i != j else 1. for i in range(n)] for j in range(n)]

        uk[ik][jk] = math.sin(phi)
        uk[jk][ik] = -math.sin(phi)

        uk[ik][ik] = math.cos(phi)
        uk[jk][jk] = math.cos(phi)

        # домножаем матрицу а слева на u^T и справа на u^T
        tmp = multi(uk, ak)
        uk[ik][jk], uk[jk][ik] = uk[jk][ik], uk[ik][jk]

        ak = multi(tmp, uk)
        u = multi(u, uk)

        count = 0

        for i in range(n - 1):
            for j in range(i + 1, n):
                count += ak[i][j] ** 2

        average = math.sqrt(count)
        if average < eps:
            cov = True

    return [ak[i][i] for i in range(n)], u

def multi(m1, m2):
    sum = 0  # сумма
    tmp = []  # временная матрица
    ans = []  # конечная матрица
```

```
    row1 = len(m1)  # количество строк в первой
матрице
    col1 = len(m1[0])  # Количество столбцов в 1
    row2 = col1  # и строк во 2ой матрице
    col2 = len(m2[0])  # количество столбцов во 2ой
матрице
    for k in range(0, row1):
        for j in range(0, col2):
            for i in range(0, col1):
                sum = sum + m1[k][i] * m2[i][j]
            tmp.append(sum)
            sum = 0
        ans.append(tmp)
        tmp = []
    return ans

def matrixsum(a, b):
    out = [[0] * len(a[0]) for _ in range(len(a))]
    for i in range(len(a)):
        for j in range(len(a[0])):
            out[i][j] = a[i][j] + b[i][j]
    return out

def show(a, n):
    for i in range(0, n):
        for j in range(0, n):
            print("\t", a[i][j], " ", end='')
        print("\n")


eps = 0.01
n = 3

# вариант 17
a = [[5, -3, -4], [-3, -3, 4], [-4, 4, 0]]

print("Симметрическая матрица:")
show(a, n)

x, u = rotation(a, eps)
```

```
print('x:\n', x)
print('u:\n')
show(u, len(u))

print("Проверка с помощью linalg:")
x, u = np.linalg.eig(a)
print('x:\n', x)
print('u:\n')
show(u, len(u))
```

# Лабораторная работа 1.5

# QR алгоритм

**Задача**: 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.

**Матрица**:

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

**Решение**: В основе *QR*-алгоритма лежит представление матрицы в виде $A = QR$ , где $Q$ - ортогональная матрица *(Q−1 =QT* ), 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.

**Работа программы**:

```
matrix a---------

     8.321323756098895    3.9200195042042627
-10.717478904152857

     9.741763060109453e-05    -6.26787637003157
1.941650588671829

     2.960242502317698e-10    3.868927347302183e-05
4.946552613932668

SZ:

L1 = 8.321323756098895

L2 = -6.26787637003157

L3 = 4.946552613932668

Проверка с помощью linalg:
x:
```

```
 [-6.26790924  8.32134993  4.94655931]
u:

     -0.901557940001651   0.2730689443868811
-0.34810989894045224

     -0.4016670501729452      0.38588216292860433
-0.07180764618281674

     -0.16080068913205686     -0.8812084361517293
0.9346995026260321
```

**Листинг программы**:

```
import numpy as np
import math

"""Лабораторная работа 1, задача 5, вариант 17"""

def QR(a):
  h = np.zeros((n, n))
  hk = np.zeros((n, n))
  r = np.zeros((n, n))

  E = np.zeros((n, n))
  res_d = 0
  res_x1 = 0
  res_x2 = 0
  res_y1 = 0
  res_y2 = 0

  prev_d = 0
  prev_x1 = 0
  prev_x2 = 0
  prev_y1 = 0
  prev_y2 = 0
  flag = 0
  for i in range(n):
      for j in range(n):
          if i != j:
```

```python
                E[i][j] = 0
                h[i][j] = 0
            else:
                E[i][j] = 1
                h[i][j] = 1

    counter = 0
    v = np.zeros(n)
    while True:
        counter = counter + 1

        for i in range(n-1):
            norma = get_norm_of_row(a, i)
            signum = 0
            for j in range(0, n):
                if j >= i:
                    v[j] = a[j][i]
                    if i == j:
                        signum = sign(a[j][i])
                        v[j] += signum * norma
                else:
                    v[j] = 0

            hk = multiplyQR(v)

            for k in range(n):
                for j in range(n):
                    if k == j:
                        hk[k][j] = 1 - hk[k][j]
                    else:
                        hk[k][j] = -hk[k][j]

            # получаем матрицу Хаусхолдера
            h = multiply(h, hk)

            # изменяем матрицу A = A*h
            a = multiply(hk, a)
            # Получаем верхнетреугольную матрицу r
            if i == n - 2:
                r = multiply(E, a)
```

```
        a = multiply(r, h)

        # считаем корень из суммы квадратов
поддиагональных элементов
        sum = 0
        for i in range(n-1):
            for j in range(i+1, n):
                end[i] += a[j][i]**2
            sum += end[i]
        sum = math.sqrt(sum)
        # если сумма меньше точности, то выходим
        if sum < epsilon:
            break

        # проверяем, есть ли комплексные собственные
значения
        if precheckComplex(a) == 1:

            indexes = checkComplex(a)
            for i in range(n-1):
                flag = 1
                if indexes[i] != -1:
                    res_d, res_x1, res_x2, res_y1,
res_y2 = getRoots(a, i)
                    flag = 1
                    if (prev_x1 - res_x1 < epsilon and
                        prev_x2 - res_x2 < epsilon and
                        prev_y1 - res_y1 < epsilon and
                        prev_y2 - res_y2 < epsilon):
                        flag = 0

                    prev_x1 = res_x1
                    prev_x2 = res_x2
                    prev_y1 = res_y1
                    prev_y2 = res_y2

            summ = 0
            for i in range(n-1):
                if indexes[i] != -1:
```

```
                    summm += flag
            #  выходим, если разница между комплексными
корнями двух
            #  соседних решений меньше заданной
точности
            if summ == 0:
                break

        #  Обнуляем матрицу Хаусхолдера
        h = multiply(E, E)
    return a, res_x1, res_x2, res_y1, res_y2, flag

def sign(x):
    if x > 0:
        return 1
    elif x < 0:
        return -1
    else:
        return 0

def show(a, n):
    for i in range(0, n):
        for j in range(0, n):
            print("\t", a[i][j], " ", end='')
        print("\n")

def precheckComplex(a):
    for j in range(n-2):
        if abs(a[n - 1][j]) > epsilon:
            return 0
    return 1

def checkComplex(a):
    indexes_of_nonzero_subdiag = np.zeros(n-1)
    for i in range(n-1):
        indexes_of_nonzero_subdiag[i] = -1
        if abs(a[i + 1][i]) > epsilon:
            indexes_of_nonzero_subdiag[i] = i
    return indexes_of_nonzero_subdiag
```

```
def getRoots(a, i):
    res_d = 0
    a22 = a[i][i]
    a23 = a[i][i + 1]
    a32 = a[i + 1][i]
    a33 = a[i + 1][i + 1]
    d = (a22 + a33)**2 + 4 * (a23 * a32 - a22 * a33)
    if d < 0:
        d *= -1
        res_d = -1
    else:
        res_d = 1

    res_x1 = (a22 + a33) / 2
    res_x2 = (a22 + a33) / 2
    res_y1 = math.sqrt(d) / 2
    res_y2 = -math.sqrt(d) / 2
    res_flag = 0

    return res_d, res_x1, res_x2, res_y1, res_y2

def multiply(m1, m2):
    sum = 0  # сумма
    tmp = []  # временная матрица
    ans = []  # конечная матрица
    row1 = len(m1)  # количество строк в первой
матрице
    col1 = len(m1[0])  # Количество столбцов в 1
    row2 = col1  # и строк во 2ой матрице
    col2 = len(m2[0])  # количество столбцов во 2ой
матрице
    for k in range(0, row1):
        for j in range(0, col2):
            for i in range(0, col1):
                sum = sum + m1[k][i] * m2[i][j]
            tmp.append(sum)
            sum = 0
        ans.append(tmp)
        tmp = []
    return ans
```

```
#  функция для получения второго слагаемого формулы
#  для получения матрицы Хаусходера
def multiplyQR(a):
    pr = 0
    ab = np.zeros((n, n))
    for i in range(n):
        pr += a[i] * a[i]
        for j in range(n):
            ab[i][j] = 0
            ab[i][j] = a[i] * a[j]
    pr /= 2
    for i in range(n):
        for j in range(n):
            ab[i][j] /= pr
    return ab

#  норма столбца матрицы
def get_norm_of_row(a, row):
    sum = 0
    for i in range(row, n):
        sum += a[i][row]**2
    return math.sqrt(sum)


n = 3
epsilon = 0.0001

a = [ [ -6, 1, -4 ], [ -6, 8, -2 ], [ 2, -9, 5] ]

a, res_x1, res_x2, res_y1, res_y2, flag = QR(a)

print("matrix a---------\n")
show(a, n)
print("SZ: \n")
for i in range(n):
    if i < n - 1:
        if flag == 0:
            if res_d == -1:
```

```
                print(f"L{i + 1} = {res_x1} +
i({res_y1})\nL{i + 2} = {res_x2} + i({res_y2})\n")
            else:
                print(f"L{i + 1} = {res_x1 +
res_y1}\nL{i + 2} = {res_x2 + res_y2}\n")

            i = i + 1
        else:
            print(f"L{i + 1} = {a[i][i]}\n")
    else:
        print(f"L{i + 1} = {a[i][i]}\n")

end = np.zeros(n-1)
a = [ [ -6, 1, -4 ], [ -6, 8, -2 ], [ 2, -9, 5] ]
a_ = a
print("Проверка с помощью linalg:")
x, u = np.linalg.eig(a_)
print('x:\n', x)
print('u:\n')
show(u, len(u))
```

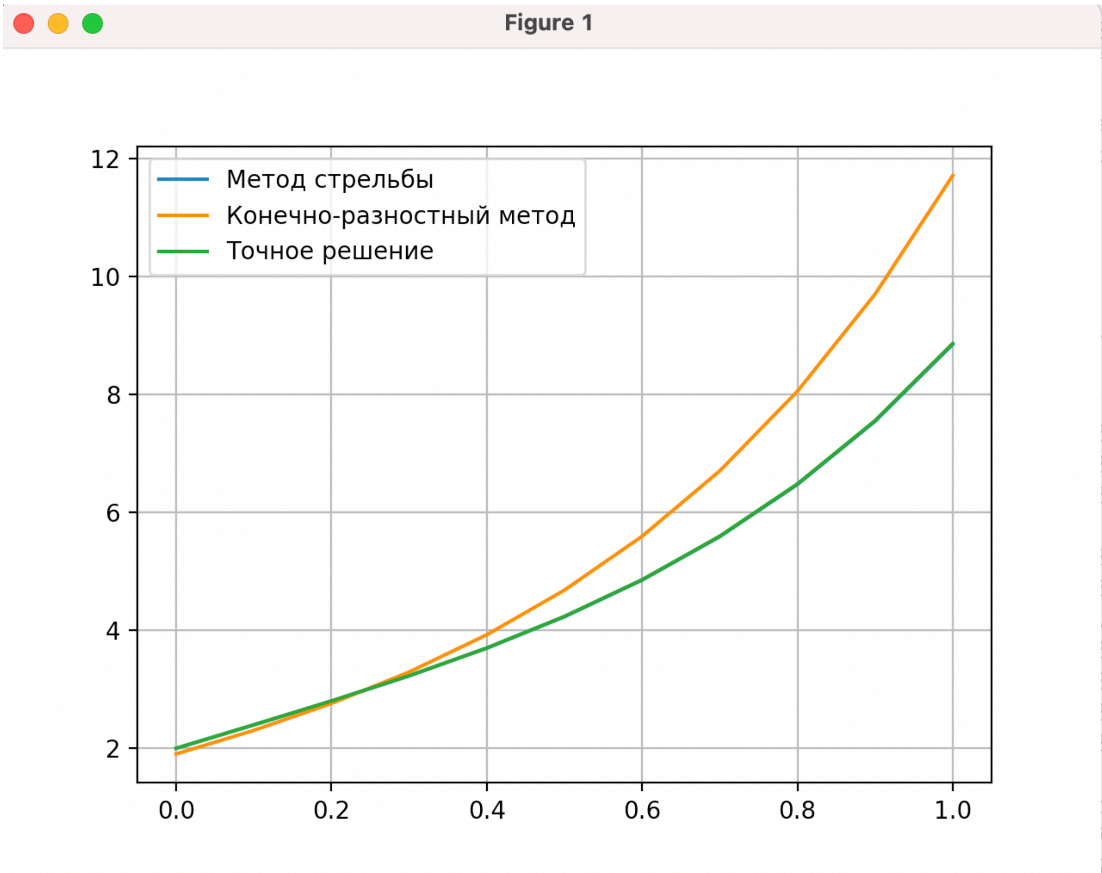
Figure 1
Метод стрельбы
Конечно-разностный метод
Точное решение
12
10
8
6
4
2
0.0
0.2
0.4
0.6
0.8
1.0