Московский авиационный институт
(Национальный исследовательский университет)

Отчет по лабораторным работам

по курсу «Численные методы»

Студент: Цыкин И.А.
Группа: М8О-301Б-19
Вариант: 14

Москва 2022

Численные методы линейной алгебры

- Лабораторная работа 1.1

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.

Условие:

$$14.\ \begin{cases} -x_1 - 3\cdot x_2 - 4\cdot x_3 = -3 \\ 3\cdot x_1 + 7\cdot x_2 - 8\cdot x_3 + 3\cdot x_4 = 30 \\ x_1 - 6\cdot x_2 + 2\cdot x_3 + 5\cdot x_4 = -90 \\ -8\cdot x_1 - 4\cdot x_2 - x_3 - x_4 = 12 \end{cases}$$

В данной лабораторной работе используется алгоритм LU разложения матриц. LU – разложение матрицы A представляет собой разложение матрицы A в произведение нижней и верхней треугольных матриц. После разложения с помощью полученных матриц L и U решается система алгебраических уравнений.

Фрагмент кода:

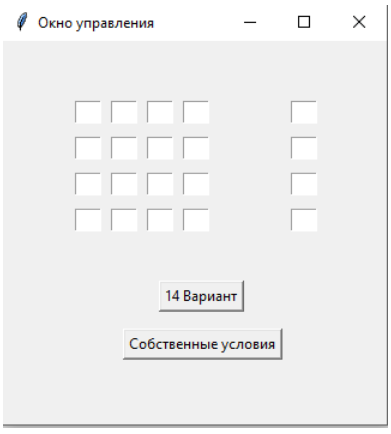
```
def funcLU(a):
    U = [[0 for i in range(len(a))] for i in range(len(a))]
    for i in range(len(a)):
        for j in range(len(a)):
            U[i][j] = a[i][j]
    L = [[0 for i in range(len(a))] for i in range(len(a))]

    for i in range(len(a)):
        for k in range(i+1,len(a)):
            L[k][i] = U[k][i]/U[i][i]
            for j in range(i,len(a)):
                U[k][j] = U[k][j] - L[k][i] * U[i][j]
        L[i][i] = 1

    return L, U

def solve(L, U, d):
    x, y = [0 for i in range(len(L))], [0 for i in range(len(U))]

    y[0] = d[0]
    for i in range(1, len(L)):
        y[i] = d[i]
        for j in range(i):
            y[i]-=L[i][j]*y[j]

    x[len(U)-1] = (y[len(U)-1]/U[len(U)-1][len(U)-1])
    for i in range(len(U)-2, -1, -1):
        x[i] = y[i]
        for j in range(len(L)-1, i, -1):
            x[i]-=U[i][j]*x[j]
        x[i] = (x[i]/U[i][i])
    return x

def det(a):
    L, U = funcLU(a)
    d = 1
```

```
    for i in range(len(U)):
        d*=U[i][i]
    return d

def obr(a):
    b = [[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]

    for k in range(len(a)):
        if abs(a[k][k]):
            for i in range(k+1, len(a)):
                if abs(a[i][k]) > abs(a[k][k]):
                    for j in range(k, len(a)):
                        a[k][j], a[i][j] = a[i][j], a[k][j]
                        b[k][j], b[i][j] = b[i][j], b[k][j]
                    b[k], b[i] = b[i], b[k]
                    break
        p = a[k][k]
        for j in range(k, len(a)):
            a[k][j] /= p
            b[k][j] /= p
        for i in range(len(a)):
            if i == k or a[i][k] == 0: continue
            f = a[i][k]
            for j in range(k, len(a)):
                a[i][j] -= f * a[k][j]
                b[i][j] -= f * b[k][j]

    return b
```

Результат выполнения:

Изначально выводится окно, в котором можно посмотреть результат 14 варианта или ввести свои данные.

```
Определитель: 2231.0
L-матрица
[[  1.           0.           0.           0.        ]
 [ -3.           1.           0.           0.        ]
 [ -1.           4.5          1.           0.        ]
 [  8.         -10.          -1.92045455   1.        ]]
U-матрица
[[ -1.          -3.          -4.           0.        ]
 [  0.          -2.         -20.           3.        ]
 [  0.           0.          88.          -8.5       ]
 [  0.           0.           0.          12.67613636]]
Обратная-матрица:
[[ 0.33333333  0.28       -0.09631728  0.12147019]
 [-0.33333333 -0.12       -0.03966006 -0.16405199]
 [ 2.66666667  1.76       -0.07082153 -0.15732855]
 [ 0.33333333 -0.08       -0.49858357 -0.15822501]]
Ответ: x = [-3.0000000000000107, 6.0000000000000036, -3.0,
-8.999999999999998]
```



Вывод:

В результате выполнения данной работы был изучен и реализован алгоритм LU-разложения. С помощью поиска корней систем уравнения результаты были высокой точность.

- Лабораторная работа 1.2

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.

Условие:

$$14.\begin{cases} -x_1 - x_2 = -4 \\ 7 \cdot x_1 - 17 \cdot x_2 - 8 \cdot x_3 = 132 \\ -9 \cdot x_2 + 19 \cdot x_3 + 8 \cdot x_4 = -59 \\ 7 \cdot x_3 - 20 \cdot x_4 + 4 \cdot x_5 = -193 \\ -4 \cdot x_4 + 12 \cdot x_5 = -40 \end{cases}$$

Метод прогонки является частным случаем метода Гаусса и используется для решения систем линейных уравнений вида Ax = B, где A — трёхдиагональная матрица. Трёхдиагональной матрицей называется матрица такого вида, где во всех остальных местах, кроме главной диагонали и двух соседних с ней, стоят нули.

Метод прогонки состоит из двух этапов: прямой прогонки и обратной прогонки. На первом этапе определяются прогоночные коэффициенты, а на втором – находят неизвестные x.

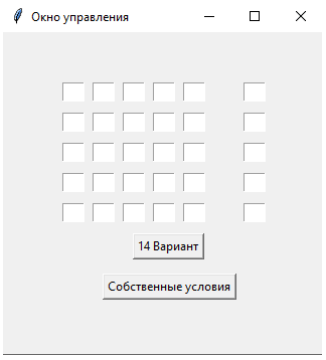
На вход программы подается матрица системы, вектор правых частей.

Фрагмент кода:

```
def func(a, d, P, Q, n):
    '''Прямой обход'''
    P[0] = -a[0][1]/a[0][0]
    Q[0] = d[0]/a[0][0]
    for i in range(1, n-1, 1):
        P[i] = -a[i][i+1]/(a[i][i]+a[i][i-1]*P[i-1])
        Q[i] = (d[i]-a[i][i-1]*Q[i-1])/(a[i][i]+a[i][i-1]*P[i-1])
    P[n-1] = 0
    Q[n-1] = (d[n-1] - a[n-1][n-2]*Q[n-2])/(a[n-1][n-1]+a[n-1][n-2]*P[n-2])

    '''Обратный обход'''
    d[n-1] = (Q[n-1])
    for i in range(n-2, -1, -1):
        d[i] = (P[i]*d[i+1] + Q[i])
    return d
```

Результат выполнения:

Изначально выводится окно, в котором можно посмотреть результат 14 варианта или ввести свои данные.



```
[6.0, -2.0, -7.0, 6.999999999999999, -1.0000000000000002]
```

Вывод:

В результате выполнения данной работы был изучен и реализован метод прогонки. С помощью поиска корней систем уравнения результаты были высокой точности.

- Лабораторная работа 1.3

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.

Условие:

14. $$\begin{cases} -22 \cdot x_1 - 2 \cdot x_2 - 6 \cdot x_3 + 6 \cdot x_4 = 96 \\ 3 \cdot x_1 - 17 \cdot x_2 - 3 \cdot x_3 + 7 \cdot x_4 = -26 \\ 2 \cdot x_1 + 6 \cdot x_2 - 17 \cdot x_3 + 5 \cdot x_4 = 35 \\ -x_1 - 8 \cdot x_2 + 8 \cdot x_3 + 23 \cdot x_4 = -234 \end{cases}$$

В программе реализован метод простых итераций и метод Зейделя. На вход программы подается матрица системы, вектор правых частей и точность, с которой нужно искать решение. Решение ищется в виде $x = \beta + \alpha x$. Начальное приближение $x = \beta$. Находим матрицу альфа и вектор бета, далее находим решение с помощью метода простых итераций и метода Зейделя. Метод Зейделя находит решение за меньшее число итераций.

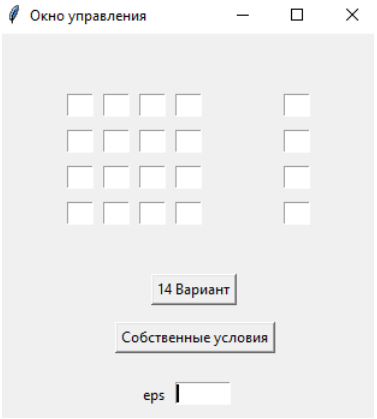
Фрагмент кода:

```
def qui(a, d):
	new_a = [[0 for i in range(len(a))] for i in range(len(a))]
	new_d = [[0] for i in range(len(a))]
	for i in range(len(a)):
		for j in range(len(a)):
			if i != j:
				new_a[i][j] = -1*a[i][j]/a[i][i]
		new_d[i][0] = d[i][0]/a[i][i]

	return np.array(new_a), np.array(new_d)

def norm(a):
	_max = 0
	for i in range(len(a)):
		delt = 0
		for j in range(len(a[0])):
			delt+=abs(a[i][j])
		if delt > _max:
			_max = delt
	return _max

def norm2(a):
	_max = 0
	for i in range(len(a)):
		if _max < abs(a[i]):
			_max = abs(a[i])
	return _max

def simple_iterations(ar, d, e):
	num = 0
	eps = 1
	a, b = qui(ar, d)
	koef = norm(a)/(1-norm(a))
	if 1 < norm(a):
		print("error")
		return 0, 0
	x = b.copy()
	while(eps > e):
		num+=1
```

```
            xk = b + np.dot(a,x)
            eps = koef * norm(x-xk)
            x = xk.copy()
      return x, num

def seidel(ar, d, e):
      num = 0
      eps = 1
      a, b = qui(ar, d)
      koef = norm(a)/(1-norm(a))
      if 1 < norm(a):
            print("error")
            return 0, 0
      xk = b.copy()
      x = b.copy()
      while(eps > e):
            num+=1
            xk = x.copy()
            for i in range(len(ar)):
                  xk[i] = b[i] + np.dot(a[i], xk)
            eps = koef * (norm(x-xk))
            x = xk.copy()
      return xk, num    return d
```

Результат выполнения:

Изначально выводится окно, в котором можно посмотреть результат 14 варианта или ввести свои данные.




```
 >>> Метод простых итераций
Ответы
[[-5.0000359 ]
 [-2.00008654]
 [-6.00009495]
 [-9.00006071]]
Итерации: 14
Метод Зейделя
Ответы
[[-5.00003918]
 [-2.00002555]
 [-6.00001321]
 [-9.000006  ]]
Итерации: 7
```

Вывод:

В результате выполнения данной работы был изучены и реализованы методы простых итераций и Зейделя. С помощью них удалось добить хорошей точности решения систем уравнения.

- Лабораторная работа 1.4

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.

Условие:

$$14.\begin{pmatrix} -7 & -5 & -9 \\ -5 & 5 & 2 \\ -9 & 2 & 9 \end{pmatrix}$$

Метод вращений Якоби применим только для симметрических матриц и решает полную проблему собственных значений и собственных векторов таких матриц. На вход программы подается матрица и точность вычислений. На каждой итерации выбираем максимальный по модулю внедиагональный элемент матрицы. Далее находим соответствующую этому элементу матрицу вращения U и получаем новую матрицу A: $A^{(i+1)} = U^{(i)T}A^{(i)}U^{(i)}$. В качестве критерия окончания итерационного процесса используется условие малости суммы квадратов вне диагональных элементов. Матрица собственных векторов V ищется одновременно с матрицей собственных значений. Изначально задает V единичной матрицей, далее на каждой итерации $V^{(i+1)} = V^{(i)}$

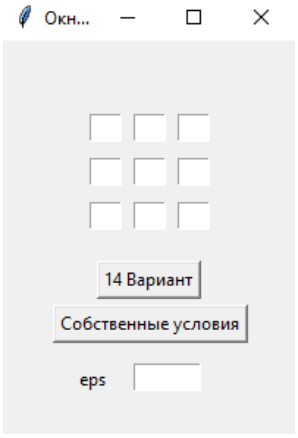
Фрагмент кода:

```
def maxij(a):
	_max = 0
	_i = 0
	_j = 0
	for i in range(len(a)):
		for j in range(len(a)):
			if i != j:
				if _max < abs(a[i][j]):
					_max = abs(a[i][j])
					_i = i
					_j = j
	return _i, _j

def norm(a):
	_max = (a[0][1]**2 + a[0][2]**2 + a[1][2]**2)**(0.5)
	return _max

def mat_rotation(a, i, j):
	if a[i][i] == a[j][j]:
		phi = math.pi/4
	else:
		phi = 0.5 * np.arctan((2*a[i][j])/(a[i][i] - a[j][j]))
	E = np.eye(len(a))
	E[i][i] = np.cos(phi)
	E[j][j] = np.cos(phi)
	E[i][j] = -np.sin(phi)
	E[j][i] = np.sin(phi)
	return E

def jacobi_rotation_method(a, e):
	_a = a.copy()
	E = np.eye(len(a))
	_eps = 1
	num = 0
	while(_eps > e):
```

```
            num+=1
            i, j = maxij(_a)
            U_1 = mat_rotation(_a, i, j)
            E = np.dot(E, U_1)
            _a = np.dot(np.transpose(U_1), _a)
            _a = np.dot(_a, U_1)
            _eps = norm(_a)
    return E, _a, num
```

Результат выполнения:

Изначально выводится окно, в котором можно посмотреть результат 14 варианта или ввести свои данные.




```
>>> Матрица:
[[-1.19006325e+01 -4.36433516e-18  4.72306819e-04]
 [-8.03142439e-16  4.21782203e+00  4.67150608e-07]
 [ 4.72306819e-04  4.67150607e-07  1.46828105e+01]]
Собственные значения:
Лямбда 0  =  -11.900632493807796
Лямбда 1  =  4.217822029602441
Лямбда 2  =  14.682810464205359
Собственные вектора:
x 0  =  [0.90279102 0.22362477 0.36736949]
x 1  =  [-0.03922165  0.8934393  -0.44746831]
x 2  =  [-0.42828734  0.38956153  0.8153599 ]
Количество итераций:  6
```

Вывод:

В результате выполнения данной работы был изучен и реализован метод вращений. Результаты были высокой точности.

- Лабораторная работа 1.5

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

Условие:

$$14.\begin{pmatrix} 2 & -4 & 5 \\ -5 & -2 & -3 \\ 1 & -8 & -3 \end{pmatrix}$$

На вход программы подается матрица A и точность вычислений. Для нахождения собственных значений матрицы необходимо найти ее QR разложение, где Q - ортогональная матрица, а R - верхняя треугольная. Такое разложение существует для любой квадратной матрицы. Матрицы Q и R находятся итерационно по формулам $Q^{(i+1)} = Q^{(i)}H^{(i)}, R^{(i+1)} = H^{(i)}R^{(i)}$, где H – соответствующая матрица Хаусхолдера. После QR разложения матрицы A получаем новую матрицу с помощью перемножения R и Q: $A = RQ$. Таким образом, каждая итерация реализуется в два этапа. На первом этапе осуществляется разложение матрицы $A^{(k)}$ в произведение матриц Q и R, а на втором – полученные матрицы перемножаются в обратном порядке. Последовательность $A^{(k)}$ сходится к верхней треугольной матрице или к верхней квазитреугольной матрице. Каждому вещественному собственному значению будет соответствовать столбец со стремящимися к нулю поддиагональными элементами. Каждой комплексно-сопряженной паре соответствует диагональный блок размерностью 2x2. Производится проверка, сходятся ли поддиагональные элементы к 0. Если сходятся, то считаем что диагональный элемент - вещественное собственное значение. Иначе проверяем наличие комплексно-сопряженной пары.

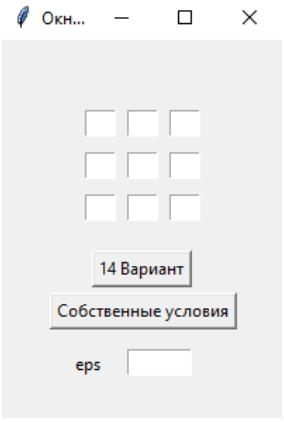
Фрагмент кода:

```
def find_v(a, n):
	v = [[a[i][n]] for i in range(len(a))]
	g = 0
	for i in range(n, len(a)):
		g += a[i][n] ** 2
	v[n][0] += np.sign(v[n][0]) * math.sqrt(g)
	for i in range(n):
		v[i][0] = 0

	return np.array(v)

def find_housholder(a, n):
	v = find_v(a, n)
	v_t = v.transpose()
	vv_t = np.dot(v, v_t)
	v_tv = np.dot(v_t, v)

	return np.eye(len(a)) - 2 / v_tv[0][0] * vv_t
```

```
def find_QR(a):
	R = a.copy()
	Q = np.eye(len(a))
	for i in range(len(R) - 1):
		H = find_housholder(R, i)
		Q = np.dot(Q, H)
		R = np.dot(H, R)
	return Q, R

def norm(a):
	s = 0
	for i in range(len(a)):
		for j in range(len(a)):
			if j == 0 and i > j:
				s += a[i][j] ** 2
	return math.sqrt(s)

def QRmethod(a, eps):
	it = 0
	A_ = a.copy()
	e = norm(A_)
	while(e > eps):
		it += 1
		Q, R = find_QR(A_)
		A_ = np.dot(R, Q)
		e = norm(A_)

	return A_, it

def solve_roots(a):
    res = [a[0][0]]
    b = a[2][2]+a[1][1]
    D = b ** 2 - 4 * (a[1][1]*a[2][2] - a[1][2] * a[2][1])
    b = -(a[2][2]+a[1][1])/2
    res.append(b + cmath.sqrt(D)/2)
    res.append(b - cmath.sqrt(D)/2)

    return res
```

Результат выполнения:

Изначально выводится окно, в котором можно посмотреть результат 14 варианта или ввести свои данные.




```
>>> Количество итераций:  32
Матрица A:
[[ 7.81324246e+00  1.05338873e+00  1.70072845e+00]
 [-8.50627265e-04 -4.13546933e+00  5.45463385e+00]
 [ 1.59382181e-05 -7.09468380e-01 -6.67777314e+00]]
Решения:
[7.813242461442525, (5.406621230721267+1.501353748634543j),
(5.406621230721267-1.501353748634543j)]
```

Вывод:

В результате выполнения данной работы был изучен и реализован метод вращений. Результаты были высокой точности.

Численные методы решения нелинейных уравнений и систем

- Лабораторная работа 2.1

Задание:

Реализовать методы простой итерации и Ньютона решения нелинейных уравнений в виде программ, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения найти положительный корень нелинейного уравнения (начальное приближение определить графически). Проанализировать зависимость погрешности вычислений от количества итераций.

Условие:

14. $x^3 - 2x^2 - 10x + 15 = 0.$

В программе реализованы методы простой итерации и Ньютона решения нелинейных уравнений. Для использования метода простых итераций исходное уравнение заменяется эквивалентным уравнением с выделенным линейным членом: $x = \varphi(x)$. Чтобы воспользоваться методом Ньютона, нужно найти начальную точку, в которой значение функции, умноженное на значение второй производной в этой точке больше 0. Далее находится решение согласно итерационной формуле метода.

Фрагмент кода:

```
def norm_1(x, x_1):
    return abs(x - x_1)

def prov(x_1):
    if (x_1 ** 3 - 2 * x_1 ** 2  - 10 * x_1 + 15) * (6 * x_1 - 4) > 0:
            return 1
    else:
            return 0

def Nfunc(x):
    k_1 = x ** 3 - 2 * x ** 2  - 10 * x + 15
    k_2 = 3 * x ** 2 - 4 * x - 10
    return -k_1/k_2

def newton(x_1, eps):
    num = x_1
    it = 0
    e = 1
    if not prov(x_1):
            print('error change x_1')
    else:
            while(e > eps):
                    it+=1
                    x = x_1
                    x_1 += Nfunc(x_1)
                    e = norm_1(x, x_1)
                    print('iter - ', it, 'x = ', x_1, ' norm = ', e)
    if x_1 < 0:
            print('your answer < 0')
            if num < 0:
                    x_1 = num + 10 ** (len(str(num)) - 1)
            else:
                    x_1 = num + 10 ** len(str(num))
            print('new x_1 = ', x_1)
```

```
        newton(x_1, eps)

def pfunk(x):
    return math.sqrt(10 + 5/(x-2))

def ppfunk(x):
    return -2.5 / math.sqrt((10*x-15) * (x-2)**3)

def norm_2(a, b):
    if abs(ppfunk(a)) > abs(ppfunk(b)):
        return abs(ppfunk(a))
    else: return abs(ppfunk(b))

def simple_it(a, b, eps):
    it = 0
    q = norm_2(a, b)
    q = q/(1-q)
    e = q * norm_1(a, b)
    x = (a+b)/2
    while(e > eps):
        it+=1
        x_1 = pfunk(x)
        e =  q * norm_1(x, x_1)
        print('iter - ', it, 'x = ', x_1, ' norm = ', e)
        x = x_1
```

Результат выполнения:

```
Enter eps:0.0001
Enter x for Newton method:4
Newton
iter -  1 x =  3.6818181818181817  norm =  0.31818181818181834
iter -  2 x =  3.6203264007541387  norm =  0.06149178106404296
iter -  3 x =  3.6180371260099813  norm =  0.00228927474415741
iter -  4 x =  3.618033988755784  norm =  3.1372541973162527e-06
Simple iteration
iter -  1 x =  3.6514837167011076  norm =  0.27582948678793173
iter -  2 x =  3.609373984850898  norm =  0.0766756056556133
iter -  3 x =  3.620331210840645  norm =  0.019951491072368644
iter -  4 x =  3.6174284872057814  norm =  0.005285431252465558
iter -  5 x =  3.618193856292121  norm =  0.001393624125984106
iter -  6 x =  3.6179917985086734  norm =  0.0003679173968237865 5
iter -  7 x =  3.6180451243788156  norm =  9.709853781108528e-05
```

Вывод:

В результате выполнения данной работы был изучены и реализованы методы простой итерации и Ньютона решения нелинейных уравнений. Результаты были высокой точности.

- Лабораторная работа 2.2

Задание:

Реализовать методы простой итерации и Ньютона решения систем нелинейных уравнений в виде программного кода, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения решить систему нелинейных уравнений (при наличии нескольких решений найти то из них, в котором значения неизвестных являются положительными); начальное приближение определить графически. Проанализировать зависимость погрешности вычислений от количества итераций.

Условие:

13	2	$\begin{cases} x_1^2/a^2 + x_2^2/(a/2)^2 - 1 = 0, \\ ax_2 - e^{x_1} - x_1 = 0. \end{cases}$
14	3	
15	4	

В программе реализованы методы простой итерации и Ньютона решения систем нелинейных уравнений. При использовании метода простой итерации система уравнений приводится к эквивалентной системе $x = \varphi(x)$ в векторной форме Методе Ньютона формулы записаны в разрешенном относительно x_1 и x_2 виде.

Фрагмент кода:

```
def mat(n, x_1, x_2):
    mat = [[0, 0], [0, 0]]
    mat[0][0] = 2/9 * x_1
    mat[0][1] = 8/9 * x_2
    mat[1][0] = -math.exp(x_1)
    mat[1][1] = 3

    if n == 1:
           mat[0][0] = x_1 ** 2 / 9 + x_2 ** 2 / 2.25 - 1
           mat[1][0] = 3 * x_2 - math.exp(x_1)
    if n == 2:
           mat[0][1] = x_1 ** 2 / 9 + x_2 ** 2 / 2.25 - 1
           mat[1][1] = 3 * x_2 - math.exp(x_1)

    return mat

def det(mat):
    return mat[0][0]*mat[1][1] - mat[0][1]*mat[1][0]

def norm(x, y, x_1, y_1):
    if abs(x-x_1) > abs(y-y_1):
           return abs(x-x_1)
    else: return abs(y-y_1)

def newton(x_1, x_2, eps):
    e = 1
    it = 0
    while(e > eps):
           detJ = det(mat(0, x_1, x_2))
           detA_1 = det(mat(1, x_1, x_2))
           detA_2 = det(mat(2, x_1, x_2))
           x = x_1
           y = x_2
```

```
        x_1 = x_1 - detA_1/detJ
        x_2 = x_2 - detA_2/detJ
        it += 1
        e = norm(x, y, x_1, x_2)
        print('iter - ', it, 'x_1 = ', x_1, 'x_2 = ', x_2, ' norm = ', e)

def qhi_p(x_1, x_2):
   mat = [[0, 0], [0, 0]]
   mat[0][0] = 0
   mat[0][1] = 1 / x_2
   mat[1][0] = - 3 * x_1 / (4 * (9 - x_1 ** 2))
   mat[1][1] = 0

   return mat

def qhi(x_1, x_2):
   return math.log(3*x_2), math.sqrt(2.25 * (1 - x_1 ** 2 / 9))

def find_q(mat):
   if abs(mat[0][0]) + abs(mat[0][1]) > abs(mat[1][0]) + abs(mat[1][1]):
         return abs(mat[0][0]) + abs(mat[0][1])
   else: return abs(mat[1][0]) + abs(mat[1][1])

def simple_it(x_1, x_2, eps):
   it = 0
   mat = qhi_p(x_1, x_2)
   q = find_q(mat)
   k = q/(1-q)
   e = 1
   while(e > eps):
         it+=1
         x = x_1
         y = x_2
         x_1 = math.log(3*x_2)
         x_2 = math.sqrt(2.25 * (1 - x_1 ** 2 / 9))
         e = norm(x, y, x_1, x_2)
         print('iter - ', it, 'x_1 = ', x_1, 'x_2 = ', x_2, ' norm = ', e)
```

Результат выполнения:

```
Enter eps:0.01
Enter x_1, x_1 lies (1, 1.5):1.3
Enter x_2, x_2 lies (1, 1.5):1.2
Newton
iter -  1 x_1 =  1.3926142712764449 x_2 =  1.336375301529296  norm =  0.1363753015292961
iter -  2 x_1 =  1.3843936128811112 x_2 =  1.3307561949202766  norm =  0.008220658395333702
Simple iteration
iter -  1 x_1 =  1.2809338454620642 x_2 =  1.3563930554553287  norm =  0.15639305545532878
iter -  2 x_1 =  1.4034413000889978 x_2 =  1.325740596535056  norm =  0.12250745462693358
iter -  3 x_1 =  1.3805835327382538 x_2 =  1.3317271782474782  norm =  0.022857767350743963
iter -  4 x_1 =  1.3850890186720668 x_2 =  1.3305570647621667  norm =  0.004505485933812947
```

Вывод:

В результате выполнения данной работы был изучены и реализованы методы простой итерации и Ньютона решения систем нелинейных уравнений. Результаты были высокой точности. Удалось добиться этого за малое число итераций.

Методы приближения функций. Численное дифференцирование и интегрирование

- Лабораторная работа 3.1

Задание:

Используя таблицу значений Y_i функции $y = f(x)$, вычисленных в точках X_i $i = 0, \dots, 3$ построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки $\{X_i, Y_i\}$. Вычислить значение погрешности интерполяции в точке X^*.

Условие:

14. $y = tg(x) + x$, а) $X_i = 0, \frac{\pi}{8}, \frac{2\pi}{8}, \frac{3\pi}{8}$; б) $X_i = 0, \frac{\pi}{8}, \frac{\pi}{3}, \frac{3\pi}{8}$; $X^* = \frac{3\pi}{16}$.

Программа строит интерполяционные многочлены Лагранжа и Ньютона по заданным точкам. На вход программы подается 2 набора точек и точка, в которой требуется вычислить абсолютную погрешность. Так как функция задана в 4 точках, то для построения многочлена Лагранжа 3 степени нужно вычислить $\omega_4(x), f(x_i), \omega_4'(x_i)$ $i = 0, \dots, 3$. Многочлен Ньютона строится через разделенные разности. В конце считается абсолютная погрешность в заданной точке.

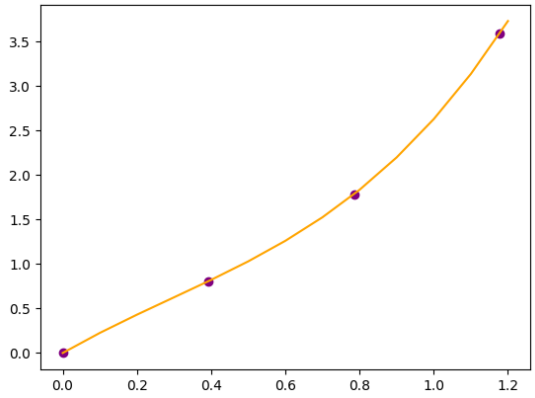
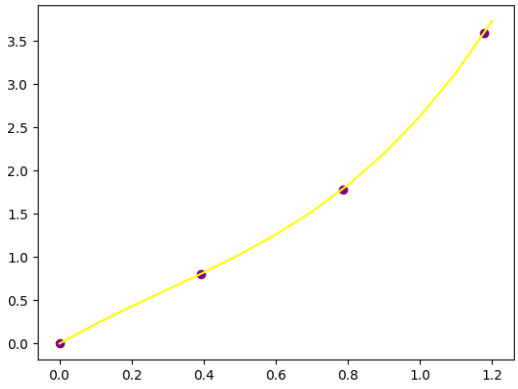
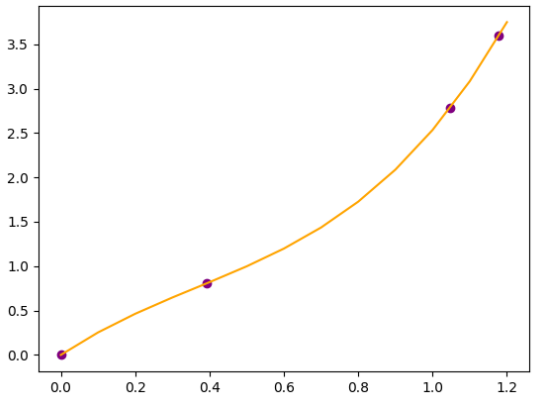
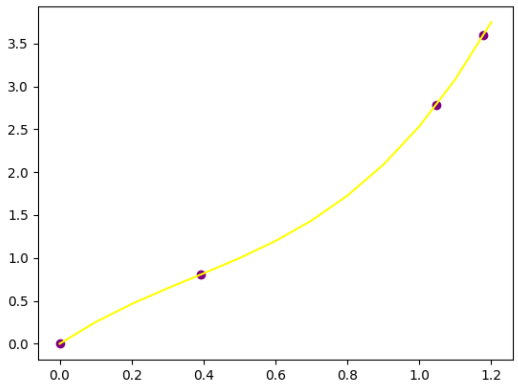
Фрагмент кода:

```
def f(x):
    return math.tan(x) + x

def f_omega(x):
    mas = [1 for i in range(len(x))]
    for i in range(len(x)):
            x_ = x[i]
            for j in range(len(x)):
                    if i != j:
                            mas[i]*=(x_ - x[j])
            mas[i] = f(x[i])/mas[i]
    return mas

def Lagrange(x, x_, mas):
    y = 0
    for i in range(len(mas)):
            lamb = mas[i]
            for j in range(len(mas)):
                    if i != j:
                            lamb *= (x - x_[j])
            y += lamb
    return y

def tabl(x):
    mas = [[] for i in range(len(x)+1)]
    mas[0] = x

    for i in range(len(x)):
            mas[1].append(f(x[i]))

    for i in range(2, len(mas)):
            for j in range(len(mas[i-1])-1):
```

```
                mas[i].append((mas[i-1][j]-mas[i-1][j+1])/(x[j]-x[j+i-1]))

    return mas


def newton(x, mas):
    y = 0
    for i in range(1, len(mas) - 1):
            lamb = mas[i+1][0]
            for j in range(i):
                    lamb *= (x - mas[0][j])
            y+=lamb

    return y
```

Результат выполнения:

```
1.2336554037346337 1.257227260467385
1.2336554037346337 1.257227260467385
1.1742994513762857 1.257227260467385
1.1742994513762854 1.257227260467385
```

Вывод:

В результате выполнения данной работы был изучены и реализованы интерполяционные многочлены Лагранжа и Ньютона.

- Лабораторная работа 3.2

Задание:

Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при $x = x_0$ и $x = x_4$. Вычислить значение функции в точке $x = X^*$.

Условие:

14. $X^* = 1.5$

i	0	1	2	3	4
x_i	0.0	0.9	1.8	2.7	3.6
f_i	0.0	0.72235	1.5609	2.8459	7.7275

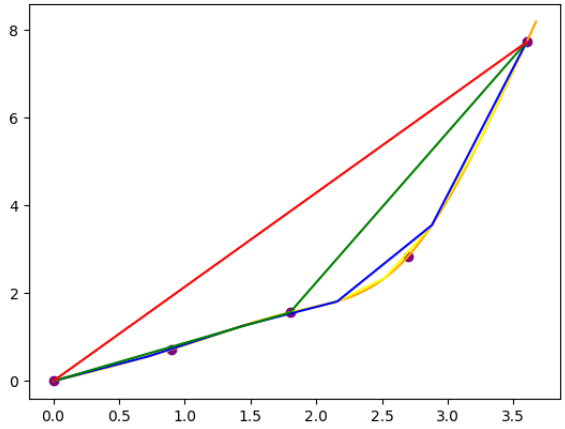
Программа строит кубический сплайн для функции, заданной в узлах интерполяции. На вход программы подается набор точек, значений в точках и точка, в которой необходимо найти значение функции. Для построения кубического сплайна необходимо построить n многочленов третьей степени, т.е. определить 4n неизвестных a_i, b_i, c_i, d_i. Для их нахождения нужно решить систему линейных алгебраических уравнений с трехдиагональной матрицей.

Фрагмент кода:

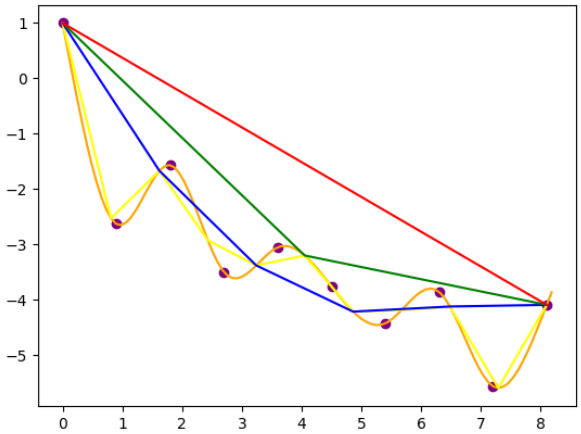
```
def h(i, x):
    return x[i] - x[i-1]

def system(x):
    n = len(x[0])-1
    arr = [[0 for i in range(len(x[0])-2)] for i in range(len(x[0])-2)]
    y = [[0] for i in range(len(x[0])-2)]
    arr[0][0] = 2*(h(1, x[0]) + h(2, x[0]))
    arr[0][1] = h(2, x[0])
    y[0][0] = 3*((x[1][2]-x[1][1])/h(2, x[0]) - (x[1][1]-x[1][0])/h(1, x[0]))
    for i in range(1, len(arr) - 1):
          arr[i][i-1] = h(i, x[0])
          arr[i][i] = 2*(h(i, x[0]) + h(i+1, x[0]))
          arr[i][i+1] = h(i+2, x[0])
          y[i][0] = 3*((x[1][i+2]-x[1][i+1])/h(i+2, x[0]) - (x[1][i+1]-x[1][i])/h(i+1,
x[0]))
    arr[-1][-2] = h(n-1, x[0])
    arr[-1][-1] = 2*(h(n-1, x[0])+h(n, x[0]))
    y[-1][0] = 3*((x[1][n]-x[1][n-1])/h(n, x[0]) - (x[1][n-1]-x[1][n-2])/h(n-1, x[0]))

    return np.array(arr), np.array(y)

def create_table(x, c):
    arr = [[0 for i in range(len(x[0])-1)] for i in range(len(x[0]) - 1)]
    arr[2][0] = 0
    for i in range(len(c)):
          arr[2][i+1] = c[i][0]
    for i in range(len(x[0])-1):
          arr[0][i] = x[1][i]
    for i in range(len(x[0])-2):
          arr[1][i] = (x[1][i+1]-x[1][i])/h(i+1, x[0]) - 1/3 * h(i+1, x[0]) * (arr[2][i+1]
+ 2*arr[2][i])
          arr[3][i] = (arr[2][i+1] - arr[2][i])/(3*h(i+1, x[0]))
    arr[1][-1] = (x[1][-1]-x[1][-2])/h(len(x[0]) - 1, x[0]) -2/3 * h(len(x[0]) - 1, x[0]) *
arr[2][-1]
    arr[3][-1] = - arr[2][-1]/(3* h(len(x[0]) - 1, x[0]))
```

```
    return arr

arr = create_table(C, c)
print(arr)

def func(x, c, A):
    summ = 0
    for i in range(len(c[0])-1):
           if x >= c[0][i] and x <= c[0][i+1]:
                  break
    x_ = x - c[0][i]
    for j in range(len(A)):
           summ += A[j][i]*(x_)**j
    return sum
```

Результат выполнения:




```
C = [[0.9*i for i in range(10)], [math.cos(0.9*i*math.pi) - (math.sqrt(0.9*i*math.pi)) for i in range(10)]]
```



Вывод:

В результате выполнения данной работы был изучены и реализованы кубический сплайн для функции, заданной в узлах интерполяции.

- Лабораторная работа 3.3

Задание:

Для таблично заданной функции путем решения нормальной системы МНК найти приближающие многочлены а) 1-ой и б) 2-ой степени. Для каждого из приближающих многочленов вычислить сумму квадратов ошибок. Построить графики приближаемой функции и приближающих многочленов.

Условие:

14.

i	0	1	2	3	4	5
x_i	-0.9	0.0	0.9	1.8	2.7	3.6
y_i	-1.2689	0.0	1.2689	2.6541	4.4856	9.9138

Программа находит приближающие многочлены разной степени для заданной табличной функции. На вход программе подается набор точек и значений. Для нахождения неизвестных коэффициентов нужно записать нормальную систему МНК и решить ее.

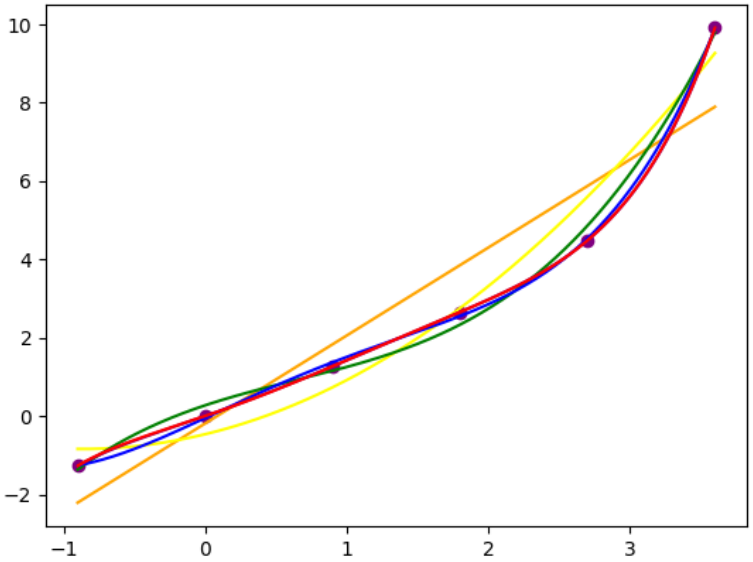
Фрагмент кода:

```
def create_table(n, C):
    arr = [[0 for i in range(n)] for i in range(n)]
    arr_y = [0 for i in range(n)]
    y = 0
    for i in range(n):
          l = i
          for j in range(n):
                for k in C[0]:
                      arr[i][j] += k**l
                l+=1
    for i in range(n):
          for j in range(len(C[0])):
                arr_y[i] += C[1][j]*(C[0][j]**i)
    return np.array(arr), np.array(arr_y)

def func(x, A):
    f = 0
    for i in range(len(A)):
          f += A[i]*(x**i)
    return f

def error(C, A):
    err = 0
    for i in range(len(C[0])):
          err += (func(C[0][i], A) - C[1][i])**2
    return err
```

Результат выполнения:

```
orange цвет - коэффициенты многочлен 1 степень: [-0.19012857  2.24620635]
Ошибка - 8.679034651428573
yellow цвет - коэффициенты многочлен 2 степень: [-0.46449643  0.87436706  0.50808862]
Ошибка - 2.3557060846428586
green цвет - коэффициенты многочлен 3 степень: [ 0.2733619    1.20620811 -0.46791446
0.24098842]
Ошибка - 0.35574107214285694
blue цвет - коэффициенты многочлен 4 степень: [-0.05172024   1.68781129   0.05605195 -
0.27926193  0.09634266]
```

```
Ошибка - 0.02696382892857171
purple цвет - коэффициенты многочлен 5 степень: [-3.02948900e-13    1.26332222e+00
1.23096708e-01  1.51148834e-01
 -1.51971244e-01  3.67872445e-02]
Ошибка - 1.9939483503314543e-25
red цвет - коэффициенты многочлен 6 степень: [ 8.52363292e-13  1.23276903e+00  1.59873702e-
01  1.74723831e-01
 -1.95628645e-01  5.42502049e-02 -2.15592104e-03]
Ошибка - 3.733723666318883e-24
```



Вывод:

В результате выполнения данной работы был изучены и реализованы метод интерполяции метод интерполяции путем решения нормальной системы МНК.

- Лабораторная работа 3.4

Задание:

Вычислить первую и вторую производную от таблично заданной функции $y_i = f(x_i),\ i = 0,1,2,3,4$ в точке $x = X^*$.

Условие:

14. $X^* = 3.0$

i	0	1	2	3	4
x_i	1.0	2.0	3.0	4.0	5.0
y_i	1.0	2.6931	4.0986	5.3863	6.6094

Программа с помощью разложения Ньютона считает 1-ую и 2-ую производную для таблично заданной функции. На вход программы поступает набор точек, набор значений и точка, в которой нужно вычислить производные. Для вычисления производных нужно определить интервал, которому принадлежит заданная точка.

Фрагмент кода:

```
def tabl(x, y):
    mas = [[] for i in range(len(x)+1)]
    mas[0] = x
    mas[1] = y

    for i in range(2, len(mas)):
           for j in range(len(mas[i-1])-1):
                  mas[i].append((mas[i-1][j]-mas[i-1][j+1])/(x[j]-x[j+i-1]))

    return mas

def proiz_1(x, mas):
    y = mas[2][0]
    if len(mas) > 3:
           y+=(mas[3][0]*((x-mas[0][1]) + (x-mas[0][0])))
    if len(mas) > 4:
           y+=(mas[4][0]*((x-mas[0][1])*(x-mas[0][2]) + (x-mas[0][0])*(x-mas[0][2]) + (x-
mas[0][0])*(x-mas[0][1])))
    if len(mas) > 5:
           y+=(mas[5][0]*((x-mas[0][1])*(x-mas[0][2])*(x-mas[0][3]) + (x-mas[0][0])*(x-
mas[0][2])*(x-mas[0][3]) + (x-mas[0][1])*(x-mas[0][0])*(x-mas[0][3]) + (x-mas[0][1])*(x-
mas[0][2])*(x-mas[0][0])))
    return y

def proiz_2(x, mas):
    if len(mas) < 4:
           return 'нет'
    y = 2*mas[3][0]
    if len(mas) > 4:
           y+=(2*mas[4][0]*((x-mas[0][1]) + (x-mas[0][0]) + (x-mas[0][2])))
    if len(mas) > 5:
           y+=(2*mas[5][0]*((x-mas[0][0])*(x-mas[0][1]) + (x-mas[0][0])*(x-mas[0][2]) + (x-
mas[0][0])*(x-mas[0][3]) + (x-mas[0][1])*(x-mas[0][2]) + (x-mas[0][1])*(x-mas[0][3]) + (x-
mas[0][2])*(x-mas[0][3])))
    return y
```

Результат выполнения:

```
Внесите диапазон точек
1 - число: 2
2 - число: 5
[1, 2, 3, 4]
[2.6931, 4.0986, 5.3863, 6.6094]
[1.4055000000000004, 1.2877, 1.2230999999999996]
[-0.058900000000000174, -0.03230000000000022]
[0.008866666666666653]
Введите x в пределах ( 1 , 4 ):
1.45
первая производная -  1.4105698333333336
вторая производная -  -0.1470600000000003
```

Вывод:

В результате выполнения данной удалось посчитать с помощью разложения Ньютона на интервале удалось посчитать 1 и 2 производную.

- Лабораторная работа 3.5

Задание:

Вычислить определенный интеграл $F = \int_{x_0}^{x_1} ydx$, методами прямоугольников, трапеций, Симпсона с шагами h_1, h_2. Оценить погрешность вычислений, используя Метод Рунге-Ромберга.

Условие:

14. $y = \frac{1}{x^4 + 16}$, $X_0 = 0,\ X_k = 2,\ h_1 = 0.5,\ h_2 = 0.25;$

Программа находит значение определенного интеграла методами прямоугольников, трапеций и Симпсона. В зависимости от метода используется своя формула нахождения решения. Решение находится для двух значений шагов, далее оценивается погрешность вычислений по методу Рунге-Ромберга.

Фрагмент кода:

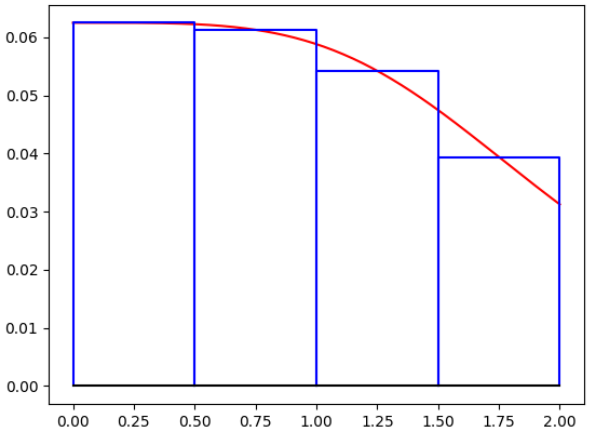
```
def func(x):
    return 1/(x**4+16)#x/((3*x+4)**2)

def rungerombergerror(f1, f2, h1, h2, p):
    k = max(h1/h2, h2/h1)
    print('Ошибка Рунге-Ромберга:', (f2-f1)/(k**p - 1))
    print('Точность_1:', f1+(f2-f1)/(k**p - 1))
    print('Точность_2:', f2+(f2-f1)/(k**p - 1))

def create_table(n, C):
    arr = [[0 for i in range(n)] for i in range(n)]
    arr_y = [0 for i in range(n)]
    y = 0
    for i in range(n):
          l = i
          for j in range(n):
                for k in C[0]:
                      arr[i][j] += k**l
                l+=1
    for i in range(n):
          for j in range(len(C[0])):
                arr_y[i] += C[1][j]*(C[0][j]**i)
    return np.array(arr), np.array(arr_y)

def func_2(x, A):
    f = 0
    for i in range(len(A)):
          f += A[i]*(x**i)
    return f

def rectangle_plot(h):
    summ = 0
    Y = []
    float_range_array = np.arange(X0, X1+0.01, 0.01)
    x = list(float_range_array)
    for i in x:
          Y.append(func(i))
    plt.plot(x, Y, color='red')
    Y = []
    float_range_array = np.arange(X0, X1+h, h)
```

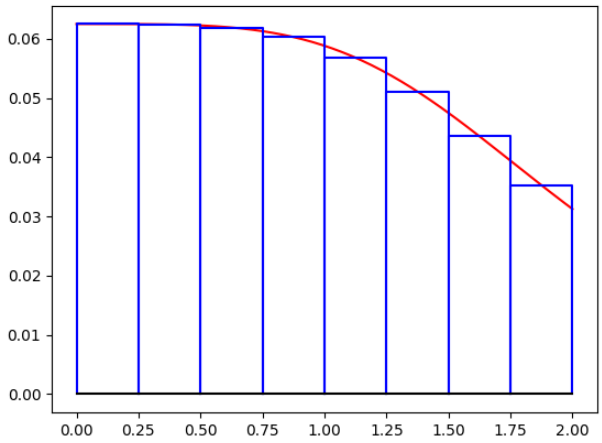
```
    x = list(float_range_array)
    for i in range(len(x)-1):
          y = func((x[i]+x[i+1])/2)
          plt.plot([x[i], x[i], x[i+1], x[i+1]], [0, y, y, 0], color = 'blue')
          plt.plot([x[i], x[i+1]], [0, 0],  color='black')
          summ += h*y
    print('шаг - ', h, ':ответ методом прямоугольников:', summ)
    plt.show()
    return summ

def trapezoid_plot(h):
    summ = 0
    Y = []
    float_range_array = np.arange(X0, X1+0.01, 0.01)
    x = list(float_range_array)
    for i in x:
          Y.append(func(i))
    plt.plot(x, Y, color='red')
    Y = []
    float_range_array = np.arange(X0, X1+h, h)
    x = list(float_range_array)
    for i in range(len(x)-1):
          y_1 = func((x[i]))
          y_2 = func((x[i+1]))
          plt.plot([x[i], x[i], x[i+1], x[i+1]], [0, y_1, y_2, 0], color = 'blue')
          plt.plot([x[i], x[i+1]], [0, 0],  color='black')
          summ += (y_1+y_2)*h
    summ/=2
    print('шаг - ', h, ':ответ методом трапеций:', summ)
    plt.show()
    return summ

def simpson_plot(h):
    summ = 0
    Y = []
    float_range_array = np.arange(X0, X1+0.01, 0.01)
    x = list(float_range_array)
    for i in x:
          Y.append(func(i))
    plt.plot(x, Y, color='red')
    Y = []
    float_range_array = np.arange(X0, X1+h, h)
    x = list(float_range_array)
    for i in range(0, len(x)-1, 1):
          y_1 = func((x[i]))
          y_2 = func((x[i+1]))
          y_12 = func((x[i]+x[i+1])/2)
          plt.plot([x[i], x[i]], [0, y_1], color = 'blue')
          arr, y = create_table(3, [[x[i], (x[i]+x[i+1])/2 ,x[i+1]],[y_1, y_12, y_2]])
          a = np.linalg.solve(arr, y)
          float_range_array = np.arange(x[i], x[i+1]+0.01, 0.01)
          x_2 = list(float_range_array)
          Y_2 = []
          for j in x_2:
                Y_2.append(func_2(j, a))
          plt.plot(x_2, Y_2, color = 'blue')
          plt.plot([x[i+1], x[i+1]], [y_2, 0], color = 'blue')
          plt.plot([x[i], x[i+1]], [0, 0],  color='black')
          summ += (y_1+4*y_12+y_2)*h
    summ/=6
    print('шаг - ', h, ':ответ методом Симпсона:', summ)
    plt.show()
```

```
    return summ
```

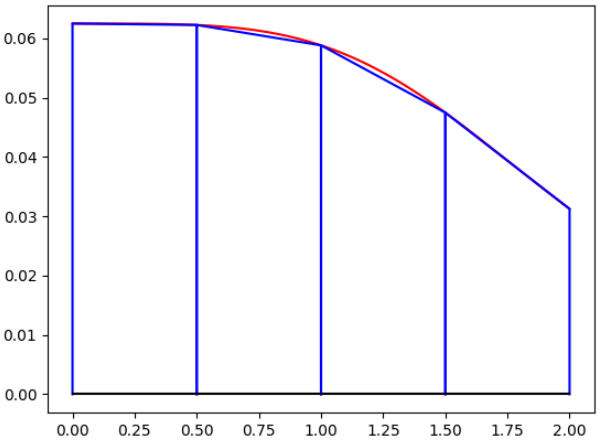
Результат выполнения:

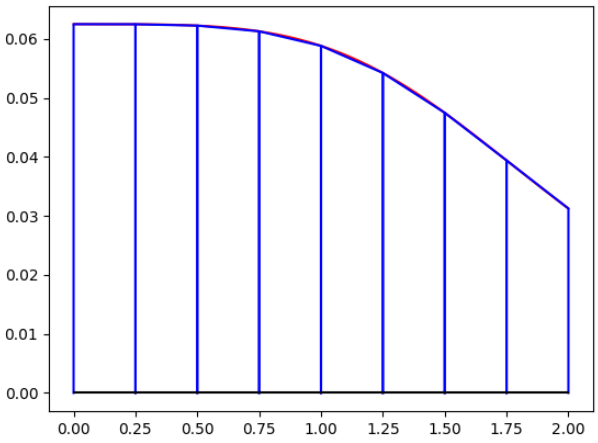
```
шаг -  0.5 :ответ методом прямоугольников: 0.10870067579634916
шаг -  0.5 :ответ методом прямоугольников: 0.10870067579634916
шаг -  0.25 :ответ методом прямоугольников: 0.10845322605071427
Ошибка Рунге-Ромберга: -8.248324854496596e-05
Точность_1: 0.1086181925478042
Точность_2: 0.1083707428021693
```



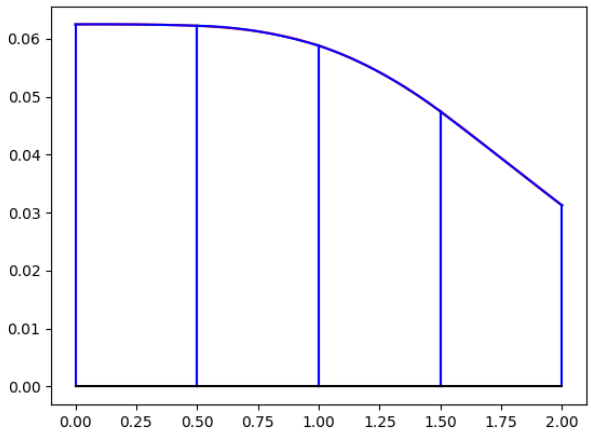


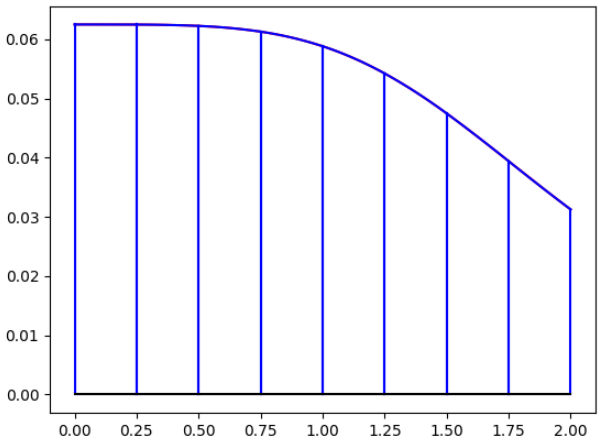

```
шаг -  0.5 :ответ методом трапеций: 0.10771654177870388
шаг -  0.25 :ответ методом трапеций: 0.10820860878752653
Ошибка Рунге-Ромберга: 0.0001640223362742165 8
Точность_1: 0.1078805641149781
Точность_2: 0.10837263112380074
```






```
шаг -  0.5 :ответ методом Симпсона: 0.10837263112380074
шаг -  0.25 :ответ методом Симпсона: 0.10837168696298505
Ошибка Рунге-Ромберга: -6.294405437978768e-08
Точность_1: 0.10837256817974636
Точность_2: 0.10837162401893066
```


0.06
0.05
0.04
0.03
0.02
0.01
0.00
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


0.06
0.05
0.04
0.03
0.02
0.01
0.00
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

Численные методы решения начальных и краевых задач для ОДУ

- Лабораторная работа 4.1

Задание:

Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки h. С использованием разработанного программного обеспечения решить задачу Коши для ОДУ 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

Условие:

14	$y''+2y'ctgx+3y=0,$ $y(1)=1,$ $y'(1)=1,$ $x\in[1,2], h=0.1$	$y=\frac{-0.9783\cos 2x+0.4776\sin 2x}{\sin x}$

Программа решает задачу Коши для ОДУ 2-го порядка методами Эйлера, Рунге-Кутты и Адамса 4-го порядка. Метод Адамса использует метод Рунге-Кутты для получения первых четырех точек. На вход программы подаются коэффициенты уравнения, начальные условия, интервал, шаг сетки и точное решение. В конце программа выводит результаты и строит графики.

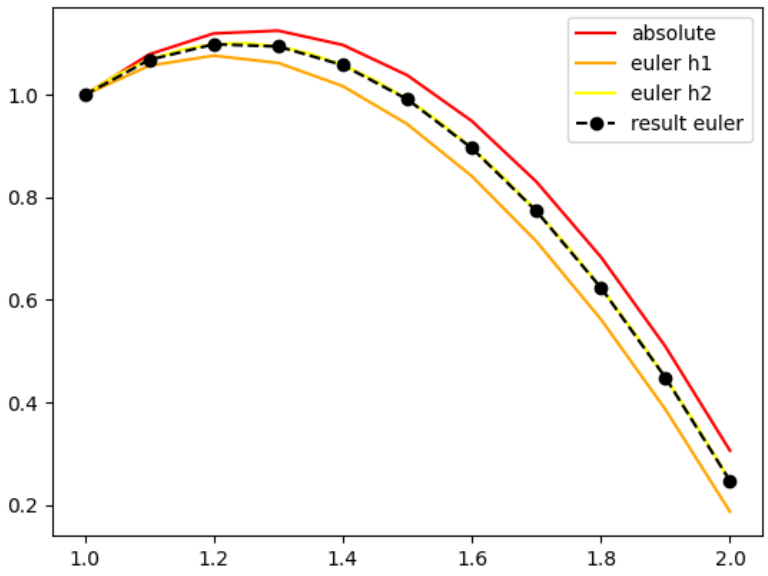
Фрагмент кода:

```
def y__(x, y, dy):
    return -2*dy/math.tan(x) - 3*y

def f(x):
    return (-0.9783*math.cos(2*x) + 0.4776*math.sin(2*x))/math.sin(x)

def error(yi, y):
    yi = np.array(yi)
    y = np.array(y)
    return (abs(yi - y)).tolist()

def euler(x, h, y0, dy0):
    y_ = []
    y = y0
    z = dy0
    for x in x:
        y_.append(y)
        z += h*y__(x, y, z)
        y += h*z

    return y_

def runge_kutta(x, h, y0, dy0):
    n = len(x)
    y_ = [y0]
    z = [dy0]
    for i in range(n-1):
        K1 = h*z[i]
        L1 = h*y__(x[i], y_[i], z[i])
        K2 = h * (z[i] + 0.5 * L1)
```

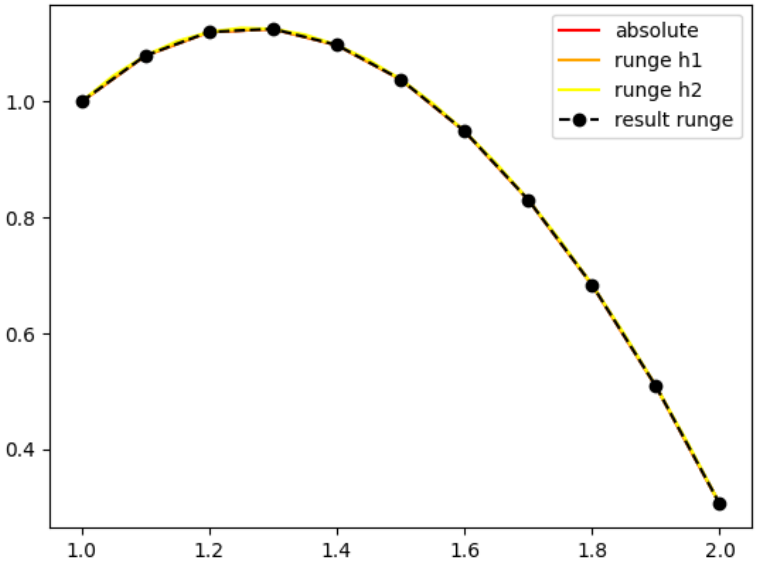
```
        L2 = h * y__(x[i] + 0.5 * h, y_[i] + 0.5 * K1, z[i] + 0.5 * L1)
        K3 = h * (z[i] + 0.5 * L2)
        L3 = h * y__(x[i] + 0.5 * h, y_[i] + 0.5 * K2, z[i] + 0.5 * L2)
        K4 = h * (z[i] + L3)
        L4 = h * y__(x[i] + h, y_[i] + K3, z[i] + L3)
        y_.append(y_[i] + (K1 + 2*K2 + 2*K3 + K4) / 6)
        z.append(z[i] + (L1 + 2*L2 + 2*L3 + L4) / 6)

    return y_, z

def adams(x, h, y_, z):
    n = len(x)
    for i in range(3, n - 1):
        z[i+1] = z[i] + h*(55*y__(x[i], y_[i], z[i]) - 59*y__(x[i - 1], y_[i - 1], z[i -
1]) + 37*y__(x[i - 2], y_[i - 2], z[i - 2]) - 9*y__(x[i - 3], y_[i - 3], z[i - 3]))/24
        y_[i+1] = y_[i] + h*(55 * z[i] - 59*z[i - 1] + 37*z[i - 2] - 9*z[i-3])/24
    return y_

def rungerombergerror(y1, y2, h1, h2, p):
    k = h1/h2
    return y1+(y2-y1)/(k**p - 1)
```

Результат:

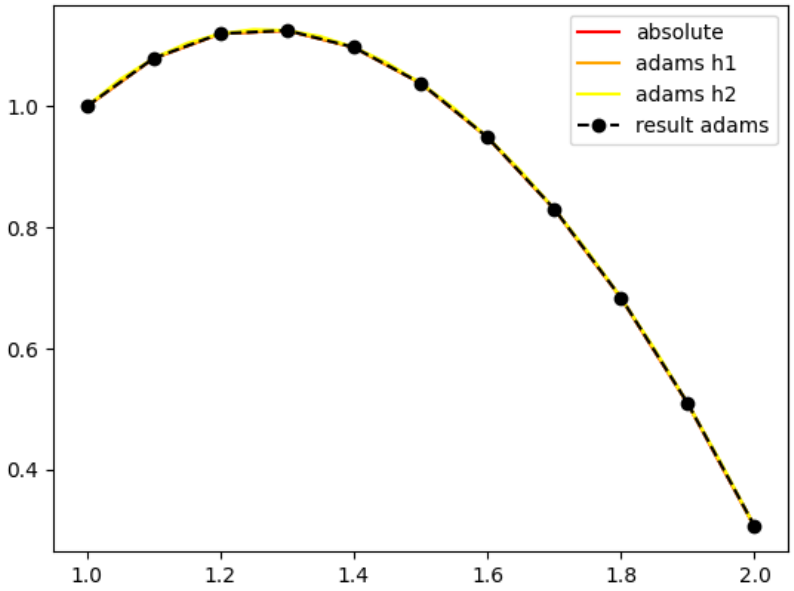
```
Method: euler, runge, adams
Enter the method: euler
Euler   error:      [8.804054235866943e-05,   0.02212886068544262,   0.04333449425868041,
0.06293465308601554,    0.0804531157010473,    0.0954735764925454,    0.10760780206446563,
0.11647372980386439, 0.12167437896928102, 0.12277035967851446, 0.11923728515292367]
runge   error:      [8.804054235866943e-05,   0.01075633127960507,   0.02115466647778841,
0.03080450900054354 5,   0.03946184117232754,   0.04691246145408767,   0.0529559560000048,
0.05739444017647255, 0.06002153877362082, 0.06060787630707215, 0.05887842714891711]
```




```
Enter the method: runge
Runge_kutta error:  [8.804054235866943e-05, 9.902184065357922e-05, 0.0001064147596225861 7,
0.00011046767860678486,          0.00011136818046875163,          0.00010925140645801079,
0.00010421112647285291, 9.630666319182524e-05, 8.556459218322932e-05, 7.197556582583253e-
05, 5.548761375667732e-05]
runge   error:     [8.804054235866943e-05,   9.89378769438165e-05,   0.0001062382376435522 2,
0.00011019251579336142,          0.00011098936091236133,          0.00010876484423105737,
0.00010361384421464503, 9.559699766392704e-05, 8.474232775279589e-05, 7.104189375184422e-
05, 5.4444653459162495e-05]
```




```
Enter the method: adams
Runge_kutta error:  [8.804054235866943e-05, 9.902184065357922e-05, 0.00010641475962258617,
0.00011046767860678486,            2.81273925961667e-05,            6.925922620237834e-05,
0.00012405210587795867,          0.00025884002748033375,          0.00046225788465203976,
0.0006830657352226988, 0.0009691340844471541]
runge  error:    [8.804054235866943e-05,  9.89378769438165e-05,  0.00010575287402314615,
0.00010958145174355671,          1.9456892913938262e-05,          7.185843893320332e-05,
0.000123427644942109,          0.00024962070041230436,          0.00043982347604598626,
0.0006462463016577713, 0.0009136614692096812]
```



Вывод:

В результате выполнения данной удалось реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка и посчитать ошибку Рунге-Ромберга.

Численные методы решения начальных и краевых задач для ОДУ

- Лабораторная работа 4.2

Задание:

Реализовать метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ в виде программ. С использованием разработанного программного обеспечения решить краевую задачу для обыкновенного дифференциального уравнения 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

Условие:

14	$(e^x+1)\,y''-2y'-e^xy=0,$ $y'(0)=1,$ $y'(1)-y(1)=1$	$y(x)=e^x-1$

В программе реализован метод стрельбы и конечно разностный метод для решения краевой задачи для ОДУ. Суть метода заключена в многократном решении задачи Коши для приближенного нахождения решения краевой задачи. Для решения задачи Коши используется метод Рунге-Кутты из предыдущей лабораторной. Следующее значение параметра η вычисляется методом секущих. В конечно-разностном методе нужно решить систему с трехдиагональной матрицей. Для решения системы используется метод прогонки. В конце программа выводит результаты и рисует графики.

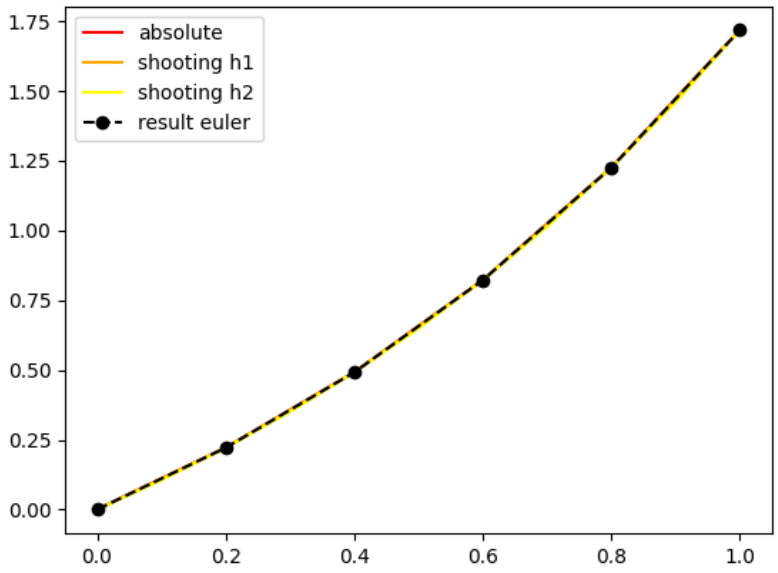
Фрагмент кода:

```
def y__(x, y, dy):
    return (2*dy + math.exp(x)*y)/(math.exp(x) + 1)

def error(yi, y):
    yi = np.array(yi)
    y = np.array(y)
    return (abs(yi - y)).tolist()

def rungerombergerror(y1, y2, h1, h2, p):
    k = h1/h2
    return y1+(y2-y1)/(k**p - 1)

def f(x):
    return math.exp(x) - 1

def runge_kutta(a, b, h, y0, dy0):
    n = int((b-a)/h)
    x = list(np.arange(a, b+h, h))
    y_ = [y0]
    z = [dy0]
    for i in range(n):
        K1 = h*z[i]
        L1 = h*y__(x[i], y_[i], z[i])
        K2 = h * (z[i] + 0.5 * L1)
        L2 = h * y__(x[i] + 0.5 * h, y_[i] + 0.5 * K1, z[i] + 0.5 * L1)
        K3 = h * (z[i] + 0.5 * L2)
        L3 = h * y__(x[i] + 0.5 * h, y_[i] + 0.5 * K2, z[i] + 0.5 * L2)
        K4 = h * (z[i] + L3)
        L4 = h * y__(x[i] + h, y_[i] + K3, z[i] + L3)
        y_.append(y_[i] + (K1 + 2*K2 + 2*K3 + K4) / 6)
```

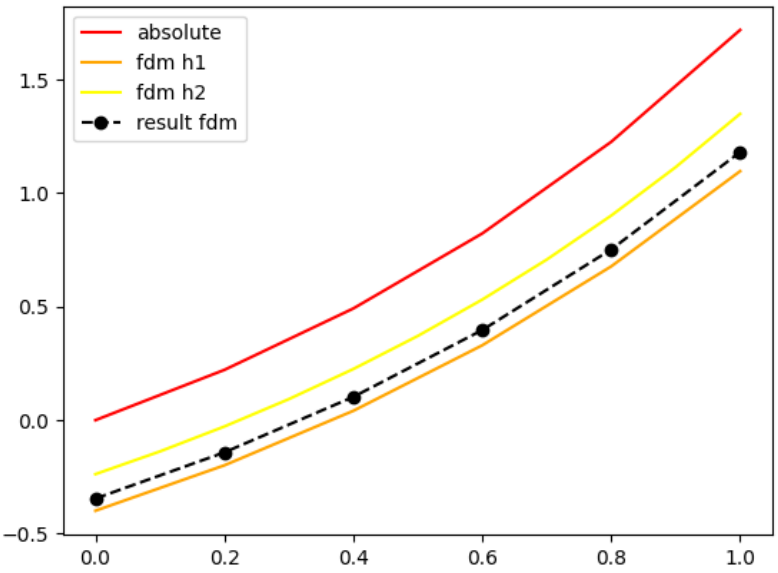
```
        z.append(z[i] + (L1 + 2*L2 + 2*L3 + L4) / 6)

    return y_, z

def shooting(a, b, h, y, c, d, eps):
    dy0 = y
    nlast = c
    n = d
    y_1, z_1 = runge_kutta(a, b, h, nlast, dy0)
    y_2, z_2 = runge_kutta(a, b, h, n, dy0)
    e = abs(y_2[-1] - (z_2[-1] - 1))
    while(e > eps):
        philast = y_1[-1] - (z_1[-1] - 1)
        phi = y_2[-1] - (z_2[-1] - 1)
        n, nlast = n - (n - nlast)/(phi - philast) * phi, n
        y_1 = y_2.copy()
        z_1 = z_2.copy()
        y_2, z_2 = runge_kutta(a, b, h, n, dy0)
        e = abs(y_2[-1] - (z_2[-1] - 1))
    print(n)
    return y_2

def p(x):
    return -2/(math.exp(x)+1)

def q(x):
    return -math.exp(x)/(math.exp(x)+1)

def f_(x):
    return 0

def fdm(a, b, h):
    x = list(np.arange(a, b+h, h))
    dy0 = 1
    arr = [[0 for i in range(len(x))] for i in range(len(x))]
    arr_y = [0 for i in range(len(x))]
    arr[0][0] = -1
    arr[0][1] = 1
    arr_y[0] = h*dy0
    for i in range(1, len(x) - 1):
        arr[i][i-1] = (2-h*p(x[i-1]))/(2*h**2)
        arr[i][i] = (q(x[i-1]) - 2/(h**2))
        arr[i][i+1] =  (2+h*p(x[i-1]))/(2*h**2)
        arr_y[i] = f_(x[i-1])*h**2
    arr[-1][-1] = (1-h)
    arr[-1][-2] = -1
    arr_y[-1] = h
    return np.linalg.solve(np.array(arr), np.array(arr_y))
```

Результат:

```
Enter h1:0.2
Method: shoot, fdm
Enter the method:shoot
enter eps:0.001
Enter n1:4
Enter n2:3
shooting error:    [3.499511042711252e-05,  3.910493043021712e-05,  4.59734449230198e-05,
5.64204919683009e-05, 7.14335235192376e-05, 9.22059156414079e-05]
```




```
Enter the method:fdm
fdm    error:        [0.3451634321256718,    0.36329907556334645,    0.38933714125979774,
0.42536892351706335, 0.47384861667489153, 0.5376595843396943]
```



Вывод:

В результате выполнения данной удалось реализовать метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ.