Московский авиационный институт
(Национальный исследовательский университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»

Лабораторная работа № 1

по курсу «Численные методы»

Методы решения задач линейной алгебры

Студент: Лукашкин К. В.
Группа: М80-308Б
Вариант: № 7
Преподаватель: Сластушенский Ю. В.

Москва, 2019

Задача

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

Вариант

$$\begin{cases} x_1 - 5 \cdot x_2 - 7 \cdot x_3 + x_4 = -75 \\ x_1 - 3 \cdot x_2 - 9 \cdot x_3 - 4 \cdot x_4 = -41 \\ -2 \cdot x_1 + 4 \cdot x_2 + 2 \cdot x_3 + x_4 = 18 \\ -9 \cdot x_1 + 9 \cdot x_2 + 5 \cdot x_3 + 3 \cdot x_4 = 29 \end{cases}$$

Алгоритм

```
def decompose_LU(A):
    # приводим значения к типу float
    A = [[float(x) for x in line] for line in A]
    # копируем матрицу, чтобы не изменять входную
    U = [x.copy() for x in A]

    n = len(A[0])
    # единичная матрица
    L = [[0.] * n for x in range(n)]
    for k in range(n):
        L[k][k] = 1

    for k in range(0, n - 1):
        # находим максимальный элемент в столбце ниже k и меняем
эту строку с k-той
        max_elem = U[k][k]
        max_row = k
        for j in range(k + 1, n):
            if max_elem < U[j][k]:
                max_elem = U[j][k]
                max_row = j
        # меняем местами строки
        U[k], U[max_row] = U[max_row], U[k]
        for i in range(k + 1, n):
            mu = U[i][k] / U[k][k]
            L[i][k] = mu
            for j in range(k, n):
```

```
            U[i][j] -= mu * U[k][j]
    return L, U
```

Результаты

```
konstanze@G5-5587:~/PycharmProjects/num_met$ python3 lab1_1.py
Значения x: [2.0, 3.9999999999999982, 7.000000000000002, -
8.000000000000002]
Определитель:  551.9999999999998
Обратная матрица:
 [[ 0.02898551 -0.0326087   0.52536232 -0.22826087]
 [ 0.00724638  0.05434783  0.56884058 -0.11956522]
 [-0.10869565 -0.06521739 -0.2826087   0.04347826]
 [ 0.24637681 -0.15217391  0.34057971 -0.06521739]]

Произведение A на x:
 [-75. -41.  18.  29.]
Произведение A на A^-1:
 [[ 1.00000000e+00 -2.77555756e-17  2.22044605e-16 -6.93889390e-
17]
 [ 0.00000000e+00  1.00000000e+00  4.44089210e-16 -5.55111512e-17]
 [-1.38777878e-16  8.32667268e-17  1.00000000e+00  2.77555756e-17]
 [-1.85962357e-15  2.49800181e-16 -6.60582700e-15
1.00000000e+00]]
```

Вывод

Проделав лабораторную работу, я изучил алгоритм LU - разложения матриц, и применил его для нахождения решения СЛАУ, нахождения обратной матрицы.

Задача

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

Вариант

$$\begin{cases} 15 \cdot x_1 + 8 \cdot x_2 = 92 \\ 2 \cdot x_1 - 15 \cdot x_2 + 4 \cdot x_3 = -84 \\ 4 \cdot x_2 + 11 \cdot x_3 + 5 \cdot x_4 = -77 \\ -3 \cdot x_3 + 16 \cdot x_4 + -7 \cdot x_5 = 15 \\ 3 \cdot x_4 + 8 \cdot x_5 = -11 \end{cases}$$

Алгоритм

```
def tridiagonal(a, b, c, d, n):
    P = [0.] * n
    Q = [0.] * n

    P[0] = -c[0] / b[0]
    Q[0] = d[0] / b[0]

    for i in range(1, n):
        P[i] = -c[i] / (a[i] * P[i - 1] + b[i])
        Q[i] = (d[i] - a[i] * Q[i - 1]) / (a[i] * P[i - 1] + b[i])

    x = [0.] * n
    # последний элемент
    x[n - 1] = Q[n - 1]
    for i in range(n - 1, 0, -1):
        x[i - 1] = P[i - 1] * x[i] + Q[i - 1]
    return x
```

Результаты

```
konstanze@G5-5587:~/PycharmProjects/num_met$ python3 lab1_2.py
Метод прогонки.
Значения x: [4.0, 4.0, -8.0, -1.0, -1.0]
```

Вывод

Проделав лабораторную работу, я изучил и реализовал алгоритм прогонки для нахождения решения СЛАУ, где матрица трёхдиагональна. Применил полученный алгоритм для решения заданного вариантом задания.

Задача

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

Вариант

$$\begin{cases} 29 \cdot x_1 + 8 \cdot x_2 + 9 \cdot x_3 - 9 \cdot x_4 = 197 \\ -7 \cdot x_1 - 25 \cdot x_2 + 9 \cdot x_4 = -226 \\ x_1 + 6 \cdot x_2 + 16 \cdot x_3 - 2 \cdot x_4 = -95 \\ -7 \cdot x_1 + 4 \cdot x_2 - 2 \cdot x_3 + 17 \cdot x_4 = -58 \end{cases}$$

Алгоритм

```
def seidel(A, b, eps):
    beta = mt.zeroes(n, 1)
    for i in range(n):
        beta[i][0] = b[i][0] / A[i][i]
    # print("Столбец бета:\n", mt.format(beta))
    alpha = mt.zeroes(n, n)  # пустая матрица
    for i in range(n):
        for j in range(n):
            if i != j:
                alpha[i][j] = -A[i][j] / A[i][i]
    # print("Матрица альфа:\n", mt.format(alpha))
    iter = 0
    norm = mt.norm(alpha)

    x = mt.copy(beta)
    while iter < MAX_ITER:
        x_prev = mt.copy(x)
        for i in range(n):
            summ = beta[i][0]
            for j in range(0, i):
                summ += alpha[i][j] * x[j][0]
            for j in range(i, n):
                summ += alpha[i][j] * x_prev[j][0]
            x[i][0] = summ
        iter += 1
        # проверка условия завершения
```

```
        if norm / (1 - norm) * mt.norm2(mt.subtract(x, x_prev)) <
eps:
            break

    print("Метод Зейделя выполнялся {} итераций".format(iter))
    return x


def simple_iter(A, b, eps):
    beta = mt.zeroes(n, 1)
    for i in range(n):
        beta[i][0] = b[i][0] / A[i][i]
    print("Столбец бета:\n", mt.format(beta))

    alpha = mt.zeroes(n, n)  # пустая матрица
    for i in range(n):
        for j in range(n):
            if i != j:
                alpha[i][j] = -A[i][j] / A[i][i]
    print("Матрица альфа:\n", mt.format(alpha))

    iter = 0
    norm = mt.norm(alpha)
    print("Норма  матрицы", norm)
    pred = (log(eps) - log(mt.norm2(beta)) + log(1 - norm)) /
log(norm)
    print("Верхняя оценка количества итераций: ", pred)

    x = mt.copy(beta)
    x_prev = mt.zeroes(1, n)
    while iter < MAX_ITER:
        x_prev = mt.copy(x)
        x = mt.add(beta, mt.mul(alpha, x_prev))
        iter += 1
        # проверка условия завершения
        if norm / (1 - norm) * mt.norm2(mt.subtract(x, x_prev)) <
eps:
            break

    print("Метод простых итераций выполнялся {}
итераций".format(iter))
    return x
```

Результаты

```
konstanze@G5-5587:~/PycharmProjects/num_met$ python3 lab1_3.py <
tests/1.3.txt
Столбец бета:
 [[ 6.79310345]
 [ 9.04      ]
 [-5.9375    ]
 [-3.41176471]]
Матрица альфа:
 [[ 0.         -0.27586207 -0.31034483  0.31034483]
 [-0.28        0.          0.          0.36      ]
 [-0.0625     -0.375       0.          0.125     ]
 [ 0.41176471 -0.23529412  0.11764706  0.        ]]
Норма  матрицы 0.896551724137931
Верхняя оценка количества итераций:  128.7621549168596
Метод простых итераций выполнялся 26 итераций
Метод Зейделя выполнялся 10 итераций
Значения x методом простых итераций:
 [[ 7.00000242]
 [ 6.0000019 ]
 [-8.99999829]
 [-3.00000126]]
Значения x методом Зейделя:
 [[ 7.00000041]
 [ 6.00000076]
 [-9.00000001]
 [-3.00000001]]
```

Вывод

Проделав лабораторную работу, я изучил и реализовал метод простых итераций и метод Зейделя в виде программы. Решил заданную задачу и проанализировал количество итераций для заданной точности.

Задача

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

Вариант

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

Алгоритм

```
U = mc.ident_matrix(n)
iter_count = 0
summ = 0
Ak = A.copy()
while iter_count < MAX_ITER:
    iter_count += 1
    # находим максимальный по модулю недиагональный элемент
    max_elem = abs(Ak[0][1])
    i, j = 0, 1
    for l in range(n):
        for m in range(n):
            if l < m and max_elem < abs(Ak[l][m]):  #
                max_elem = abs(Ak[l][m])
                i, j = l, m

    # угол вращения
    if Ak[i][i] == Ak[j][j]:
        phi = math.pi / 4
    else:
        phi = 0.5 * math.atan(2 * Ak[i][j] / (Ak[i][i] - Ak[j]
[j]))
    Uk = mc.ident_matrix(n)
    Uk[i][i] = Uk[j][j] = math.cos(phi)
    Uk[j][i] = math.sin(phi)
    Uk[i][j] = -math.sin(phi)
    # новое A
    Ak = Uk.transpose() * Ak * Uk
    # сохраняем произведение U
```

```
    U *= Uk

    # проверка условия завершения
    # корень суммы квадратов поддиагональны элементов < eps
    summ = 0
    for l in range(n):
        for m in range(n):
            if l < m:
                summ += Ak[l][m] ** 2
    if summ ** 0.5 < eps:
        break
```

Результаты

```
konstanze@G5-5587:~/PycharmProjects/num_met$ python3 lab1_4.py <
tests/1.4.txt
A last [[ 7.70696754e-01  5.82557166e-07  8.67361738e-19]
 [ 5.82557166e-07 -1.93441423e+01 -2.36239224e-04]
 [ 6.41847686e-16 -2.36239224e-04  6.57344552e+00]]
Количество итераций : 6
Точность последней итерации:  0.00023623994212690835
Найденная матрица U:
 [[ 0.76204662 -0.59591222 -0.25332505]
 [ 0.62257015  0.56672685  0.5396546 ]
 [-0.17802067 -0.56895458  0.80286944]]
Собственные значения:
0.7706967538253665 -19.3441422755157 6.573445521690334
Собственные векторы:
[0.7620466200609889, 0.6225701495810244, -0.17802066650893114]
[-0.595912217020221, 0.5667268468751345, -0.5689545769540947]
[-0.2533250450699027, 0.5396546023908521, 0.8028694362464567]
```

Вывод

Проделав лабораторную работу, я изучил и реализовал метод вращений. Нашел собственные значения и собственные векторы симметрической матрицы.

Задача

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

Вариант

$$\begin{pmatrix} 9 & 0 & 2 \\ -6 & 4 & 4 \\ -2 & -7 & 5 \end{pmatrix}$$

Алгоритм

```
def QR_decomp(A):
    # возвращает R, Q разложение
    v = Matrix(rows=n, cols=1)
    Q = mc.ident_matrix(n)
    Hk = Matrix(rows=n, cols=n)
    Ak = A.copy()
    for k in range(n - 1):

        for i in range(n):
            if i < k:
                v[i][0] = 0
            elif i == k:
                v[i][0] = Ak[i][i] + sign(Ak[i][i]) * \
                          mt.vector_norm(Ak.transpose()[k][i:])
                # евклидова норма столбца
            else:
                v[i][0] = Ak[i][k]

        v_vt = v * v.transpose()
        vt_v = v.transpose() * v
        vt_v = vt_v[0][0]
        coef = 2 / vt_v
        Hk = mc.ident_matrix(n) - (v_vt * coef)
        Ak = Hk * Ak
        Q *= Hk
    R = Ak
    return Q, R
```

Результаты

```
konstanze@G5-5587:~/PycharmProjects/num_met$ python3 lab1_5.py <
tests/1.5.txt
QR разложение исходной матрицы.
Q:  [[-0.81818182 -0.0928494   0.56741299]
 [ 0.54545455 -0.43742383  0.71494037]
 [ 0.18181818  0.8944492   0.40853735]]
R:  [[-1.10000000e+01  9.09090909e-01  1.45454545e+00]
 [ 7.32480137e-16 -8.01083976e+00  2.53685190e+00]
 [-9.16269499e-16  8.88178420e-16  6.03727422e+00]]
Количество итераций:  24
Полученная матрица A^k
 [[ 1.00266683e+01 -2.34881689e+00 -1.63570395e+00]
 [-3.20124189e-03  4.32117144e+00  8.40039764e+00]
 [-6.87038700e-04 -4.43801374e+00  3.65216025e+00]]
Собственные значения:
10.026668312830278
(-3.9866658435848583+6.096653688631042j)
(-3.986665843584859-6.096653688631042j)
```

Вывод

Проделав лабораторную работу, я изучил и реализовал алгоритм QR . Нашел собственные значения матрицы, в том числе комплексные.

Московский авиационный институт
(Национальный исследовательский университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»

Лабораторная работа № 2
по курсу «Численные методы»

Методы решения нелинейных уравнений и систем нелинейных уравнений

Студент: Лукашкин К. В.
Группа: М80-308Б
Вариант: № 7
Преподаватель: Сластушенский Ю. В.

Москва, 2019

Задача

2.1. 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.

Вариант

$2^x + x^2 - 2 = 0$.

Алгоритм

```
def newton(x0):
    x_prev = x0
    x = x0
    x = x - f(x) / f_diff(x)
    iter_count = 0
    while abs(x_prev - x) > eps:
        x_prev = x
        x = x - f(x) / f_diff(x)
        iter_count += 1
    return x, iter_count


def simple_iter(x0):
    x_prev = x0
    x = phi(x0)
    iter_count = 0
    while abs(x_prev - x) > eps:
        x_prev = x
        x = phi(x)
        iter_count += 1
    return x, iter_count
```

Результаты

```
konstanze@G5-5587:~/PycharmProjects/num_met$ python3 lab2_1.py < tests/2.1.txt
Корень на отрезке (0.0001 ; 3)

Метод Ньютона 0.6534825247841737
Количество итераций:  5
```

```
Метод простых итераций 0.6534430113565527
Количество итераций:  57
```

Вывод

Проделав лабораторную работу, я изучил и реализовал методы простой итерации и Ньютона решения нелинейных уравнений . Нашел положительный корень нелинейного уравнения.

Задача

2.2. 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.

Вариант

$\begin{cases} x_1^2 + x_2^2 - a^2 = 0, \\ x_1 - e^{x_2} + a = 0. \end{cases}$, где a = 2

Алгоритм

```
def newton_system(x0):
    x = x0.copy()
    # матрица функций
    F = Matrix([
        [f1],
        [f2],
    ])
    # Матрица Якоби из функций
    J = Matrix([
        [f1_diff_x1, f1_diff_x2],
        [f2_diff_x1, f2_diff_x2],
    ])

    iter_count = 0
    while True:
        iter_count += 1
        x_prev = x.copy()

        # делаем список аргмуентов ф-ии из столбца значений x
        args = x.transpose()[0]  # [x1, x2]

        # Найдем обратную матрицу
        [[a, b],
         [c, d]] = J(*args).matrix
        J_inv = Matrix([
            [d, -b],
```

```
            [-c, a],
        ])
        J_inv = J_inv * (1 / (a * d - b * c))

        x = x - J_inv * F(*args)
        # условие завершения
        if mc.norm2(x_prev - x) < eps:
            break
    return x, iter_count



def norm(x, x_prev):
    return max(x[0] - x_prev[0], x[1] - x_prev[1])



def simple_iter_system(x0, phi1, phi2, phi1_diff_x1, phi1_diff_x2,
phi2_diff_x1, phi2_diff_x2):
    x1, x2 = x0[0][0], x0[1][0]
    # q = max(
    #     abs(phi1_diff_x1(x1, x2)) + abs(phi1_diff_x2(x1, x2)),
    #     abs(phi2_diff_x1(x1, x2)) + abs(phi2_diff_x2(x1, x2)))
    #
    # if q >= 1:
    #     print("В заданной области не выполняется условие
сходимости итерационного процесса")
    #     print('q = ', q)
    #     return mc.zeroes(2, 1), 0

    # Матрица функций
    phi = Matrix([
        [phi1],
        [phi2]
    ])
    x = x0.copy()
    iter_count = 0
    while True:
        iter_count += 1
        x_prev = x
        args = x.transpose()[0]  # [x1, x2]

        x = phi(*args)

        if mc.norm2(x - x_prev) < eps:
```

```
            break

    return x, iter_count
```

Результаты

```
konstanze@G5-5587:~/PycharmProjects/num_met$ python3 lab2_2.py <
tests/2.2.txt
Найденное значение Методом Ньютона:
 [[1.54799144]
 [1.26638165]]
Количество итераций:  2

Метод простой итерации
В заданной области не выполняется условие сходимости итерационного
процесса

Для задачи из следующего варианта
Значение x
 [[1.03896011]
 [2.8622095 ]]
Количество итераций:  8
```

Вывод

Проделав лабораторную работу, я изучил и реализовал методы простой итерации и Ньютона решения систем нелинейных уравнений. Решил заданное уравнение. К сожалению, метод простой итерации, каким бы я образом не выражал переменную, не сходился. Поэтому он был применен к задаче из следующего варианта, где успешно нашел решение.

Московский авиационный институт

(Национальный исследовательский университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»

Лабораторная работа № 3

по курсу «Численные методы»

Методы приближения функций. Численное дифференцирование и интегрирование

Студент: Лукашкин К. В.

Группа: М80-308Б

Вариант: № 7

Преподаватель: Сластушенский Ю. В.

Москва, 2019

Задача

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

Вариант

7. $y = \sqrt{x}$, а) $X_i = 0, 1.7, 3.4, 5.1$; б) $X_i = 0, 1.7, 4.0, 5.1$; $X^* = 3.0$.

Алгоритм

```
def lagrange(x_0, x):
    print("Многочлен Лагранжа: ")
    polynom = ""
    summ = 0
    n = len(x)
    for i in range(n):
        mul = 1
        w = 1
        braces = ""
        for j in range(n):
            if j != i:
                w *= x[i] - x[j]
                mul *= x_0 - x[j]
                braces += '(x - {:3.1f})'.format(x[j])
        w = f(x[i]) / w
        mul = mul * w
        summ += mul
        # вывод полинома
        if polynom != '' and w >= 0:
            polynom += '+'
        if w != 0:
            polynom += "{:3.3f}".format(w) + braces
    print(polynom)
    return summ



def newton(x_0, x, show=True):
    if show:
        print("Многочлен Ньютона: ")
    polynom = ""
```

```
    summm = 0
    n = len(x)
    fi = []
    for i in range(n):
        mul = 1
        braces = ""
        for j in range(i):
            mul *= x_0 - x[j]
            braces += '(x - {:3.1f})'.format(x[j])
        fi_prev = fi.copy()
        fi = []
        for j in range(n - i):
            if i > 0:
                fi.append((fi_prev[j] - fi_prev[j + 1]) / \
                          (x[j] - x[j + i]))
            else:
                fi.append(f(x[j]))
        w = fi[0]
        mul = mul * w
        summ += mul
        # вывод полинома
        if polynom != '' and w >= 0:
            polynom += '+'
        if w != 0 and braces != ' ':
            polynom += "{:3.3f}".format(w) + braces
    if show:
        print(polynom)
    return summ
```

Результаты

Первый подпункт лабораторной

```
konstanze@G5-5587:~/PycharmProjects/num_met$ python3 lab3_1.py < tests/3.1a.txt Многочлен Лагранжа:
0.218(x - 0.0)(x - 3.4)(x - 5.1)-0.242(x - 0.0)(x - 2.7)(x - 5.1)+0.022(x - 0.0)(x - 2.7)(x - 3.4)
Значение полученное многочленом Лагранжа:  0.9977813973714987
Точное значение:  1.0
Погрешность интерполяции:  0.002218602628501265

Многочлен Ньютона:
0.366(x - 0.0)-0.112(x - 0.0)(x - 2.7)-0.002(x - 0.0)(x - 2.7)(x - 3.4)
```

```
Значение полученное многочленом Ньютона:  0.9977813973714986
Точное значение:  1.0
Погрешность интерполяции:  0.002218602628501376
```

Второй подпункт лабораторной

```
konstanze@G5-5587:~/PycharmProjects/num_met$ python3 lab3_1.py <
tests/3.1b.txt
Многочлен Лагранжа:
0.117(x - 0.0)(x - 4.0)(x - 5.1)-0.151(x - 0.0)(x - 2.7)(x -
5.1)+0.034(x - 0.0)(x - 2.7)(x - 4.0)
Значение полученное многочленом Лагранжа:  0.9944606022865535
Точное значение:  1.0
Погрешность интерполяции:  0.00553939771344647

Многочлен Ньютона:
0.366(x - 0.0)-0.115(x - 0.0)(x - 2.7)-0.000(x - 0.0)(x - 2.7)(x -
4.0)
Значение полученное многочленом Ньютона:  0.9944606022865535
Точное значение:  1.0
Погрешность интерполяции:  0.00553939771344647
```

Вывод

Проделав лабораторную работу, я изучил и реализовал алгоритм построения интерполяционных многочленов Лагранжа и Ньютона. Применил полученный алгоритм для решения заданного вариантом задания, вычислил значение погрешности интерполяции.

Задача

3.2. 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 $x = X^*$.

Вариант

7. $X^* = 3.0$

i	0	1	2	3	4
x_i	0.0	1.7	3.4	5.1	6.8
f_i	0.0	1.3038	1.8439	2.2583	2.6077

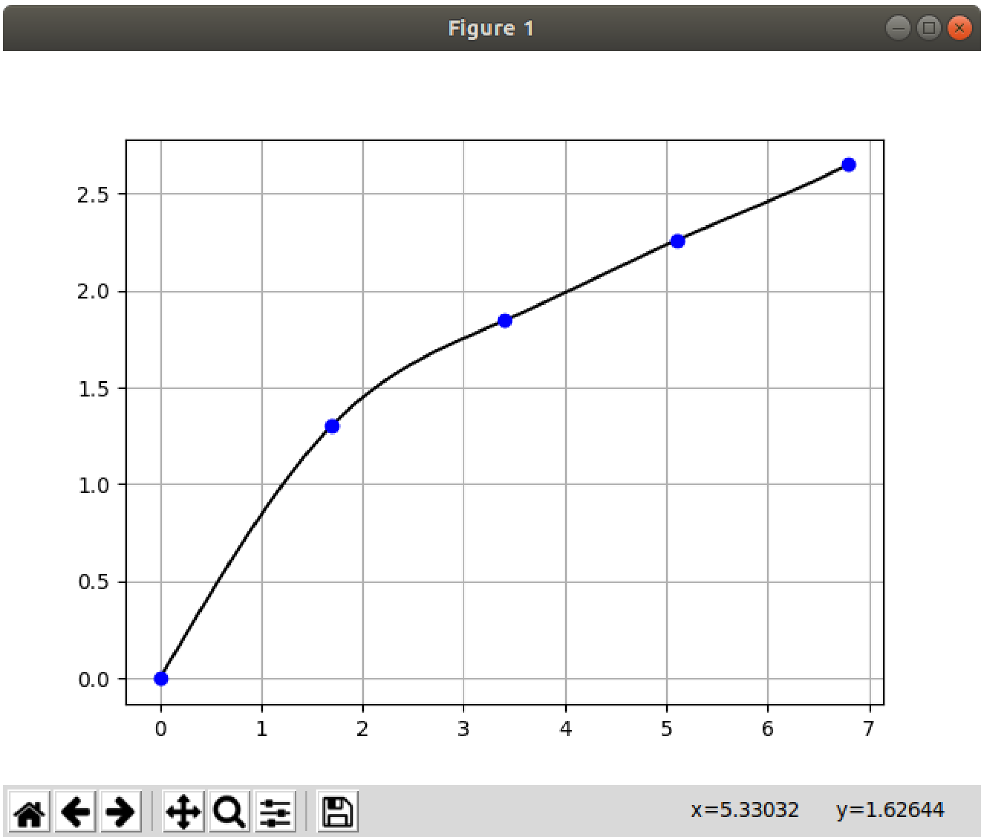
Алгоритм

```
def cubic_spline(x, f):
    n = len(x)

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

    a, b, c, d = [], [], [], []
    # b- главная диагональ
    # a - над ней
    # b - под ней
    # d - столбец свободных членов
    a = [0.] + [h(i - 1) for i in range(3, n)]
    b = [2 * (h(i - 1) + h(i)) for i in range(2, n)]
    c = [h(i) for i in range(2, n - 1)] + [0.]
    d = [3 * ((f[i] - f[i - 1]) / h(i) - ((f[i - 1] - f[i - 2]) /
h(i - 1)))
         for i in range(2, n)]

    C = [0, 0] + tridiagonal(a, b, c, d, n - 2)
    A = [0] + [f[i] for i in range(n - 1)]
    B = [0]
    for i in range(1, n - 1):
        B.append((f[i] - f[i - 1]) / h(i) - 1 / 3 * h(i) * (C[i +
1] + 2 * C[i]))
    B.append((f[n - 1] - f[n - 2]) / h(n - 1) - 2 / 3 * h(n - 1) *
C[n - 1])
```

```
    D = [0] + [(C[i + 1] - C[i]) / (3 * h(i)) for i in range(1, n
- 1)]
    D.append(-C[n - 1] / 3 * h(n - 1))

    S = []
    for i in range(1, n):
        S.append(
            [x[i - 1], A[i], B[i], C[i], D[i]])

    return S
```

Результаты

```
konstanze@G5-5587:~/PycharmProjects/num_met$ python3 lab3_2.py <
tests/3.2.txt
[0.0,   1.7] :  0.00 +  0.88(x -  0.00) +  0.00(x -  0.00)^2 + -
0.04(x -  0.00)^3
[1.7,   3.4] :  1.30 +  0.54(x -  1.70) + -0.20(x -  1.70)^2 +
0.04(x -  1.70)^3
[3.4,   5.1] :  1.84 +  0.23(x -  3.40) +  0.02(x -  3.40)^2 + -
0.01(x -  3.40)^3
[5.1,   6.8] :  2.26 +  0.23(x -  5.10) + -0.02(x -  5.10)^2 +
0.01(x -  5.10)^3
Значение в точке 3.0 согласно построенному сплайну:
1.7531560510017157
```



Вывод

Проделав лабораторную работу, я изучил и реализовал алгоритм построения кубического сплайна для функции, заданной в узлах интерполяции. Полученный результат отобразил графически.

Задача

3.3. 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.

Вариант

7.

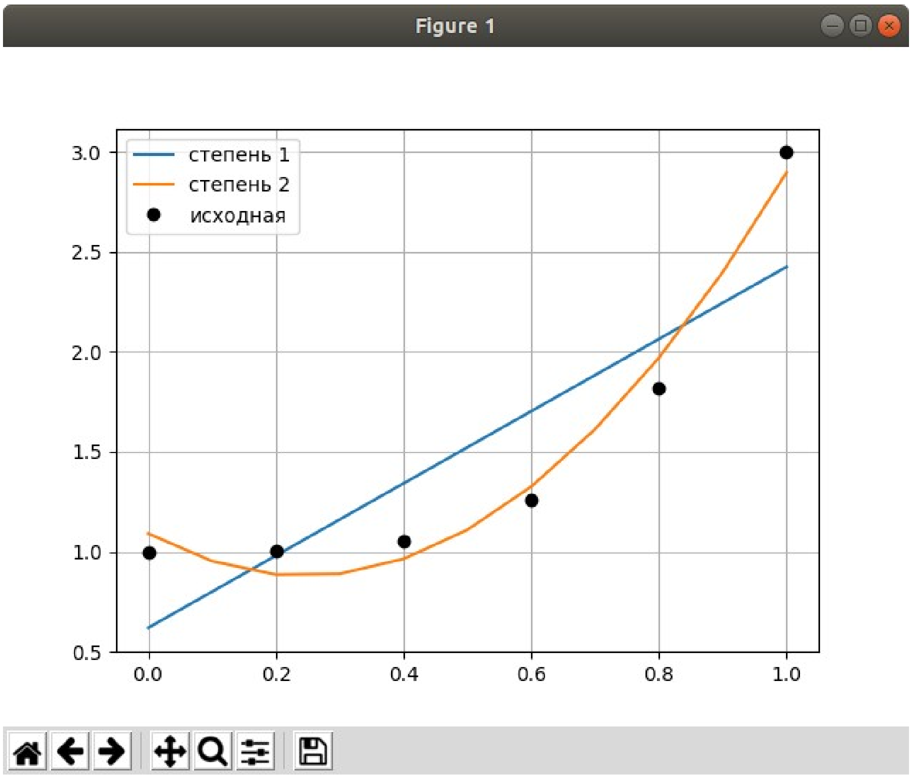
i	0	1	2	3	4	5
x_i	0.0	0.2	0.4	0.6	0.8	1.0
y_i	1.0	1.0032	1.0512	1.2592	1.8192	3.0

Алгоритм

```
ef n_degree_polynom(x, y, n):
    N = len(x)
    max_pow = 2 * n

    xpow = [N] + [0 for _ in range(max_pow)]

    b = [0 for _ in range(n + 1)]

    for i in range(N):
        for k in range(1, len(xpow)):
            xpow[k] += x[i] ** (k)
        for k in range(len(b)):
            b[k] += y[i] * (x[i] ** k)

    system = []
    for shift in range(n + 1):
        system.append([xpow[i + shift] for i in range(n + 1)])

    a = np.linalg.solve(system, b)

    def f(x):
        return a[0] + sum(a[i] * (x ** i) for i in range(1,
len(a)))

    return a, f
```

Результаты

```
konstanze@G5-5587:~/PycharmProjects/num_met$ python3 lab3_3.py <
tests/3.3.txt
Степень 1
 0.0000    0.2000    0.4000    0.6000    0.8000    1.0000
 0.6181    0.9797    1.3413    1.7029    2.0645    2.4261
Сумма квадратов ошибок: 0.816961

Степень 2
 0.0000    0.2000    0.4000    0.6000    0.8000    1.0000
 1.0905    0.8853    0.9634    1.3250    1.9701    2.8985
Сумма квадратов ошибок: 0.067198
```



Вывод

Проделав лабораторную работу, я изучил и реализовал алгоритм, который путем решения нормальной системы МНК находит приближающие многочлены а) 1-ой и б) 2-ой степени. Вычислил сумму квадратов ошибок и отобразил графически полученный результат.

Задача

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

Вариант

7. $X^* =$ 0.2

i	0	1	2	3	4
x_i	-0.2	0.0	0.2	0.4	0.6
y_i	1.7722	1.5708	1.3694	1.1593	0.9273

Алгоритм

```
def newtons_polynom(x, y, n):
    # пришлось переписывать полином, чтобы он хранил коэффициенты
    diff = mt.zeroes(n, n)
    for i in range(0, n):
        diff[i][i] = y[i]
        for j in range(i - 1, -1, -1):
            diff[j][i] = (diff[j + 1][i] - diff[j][i - 1])
            diff[j][i] /= (x[i] - x[j])
    coeff = [diff[0][i] for i in range(n)]
    return x, coeff, n


def newtons_derivative(polynom, point, order):
    # производная от заданного полинома заданной степени
    x, coeff, n = polynom

    def mul_deriv(operands, cur_order=order):
        res = 0.0
        if cur_order != 0:
            for i in range(n):
                if operands[i] != 0:
                    operands[i] = 0
                    res += mul_deriv(operands, cur_order - 1)
                    operands[i] = 1
        else:
            res = 1.0
            for i in range(n):
```

```
                if operands[i] != 0:
                    res *= (point - x[i])
        return res

    operands = [0.] * n
    for i in range(order):
        operands[i] = 1
    result = 0.0
    for i in range(order, n):
        result += coeff[i] * mul_deriv(operands)
        operands[i] = 1

    return result
    return x
```

Результаты

```
konstanze@G5-5587:~/PycharmProjects/num_met$ python3 lab3_4.py <
tests/3.4.txt
Первая
м-м 2ой степ -1.02875
м-м Ньютона -1.019625

Вторая
м-м 2ой степ -0.2081249999999978
м-м Ньютона  -0.21749999999999825
konstanze@G5-5587:~/PycharmProjects/n
```

Вывод

Проделав лабораторную работу, я изучил и реализовал алгоритм нахождения производных от таблично заданных функций. В качестве *дополнительного задания*, я также реализовал вычисление приближения многочленом второй степени. Результаты совпали.

Задача

3.5. Вычислить определенный интеграл $F = \int_{X_0}^{X_1} y\, dx$, методами прямоугольников, трапеций, Симпсона с шагами h_1, h_2. Оценить погрешность вычислений, используя Метод Рунге-Ромберга.

Вариант

7. $y = \frac{1}{3x^2 + 4x + 2}$, $X_0 = -2$, $X_k = 2$, $h_1 = 1.0$, $h_2 = 0.5$;

Алгоритм

```
def runge_romberg(h1, h2, y1, y2, n=2):
    return abs((y1 - y2) / ((h2 / h1) ** n — 1.0))

def rectangle(h, x, f):
    return h * sum(
        f((x[i - 1] + x[i]) / 2) for i in range(1, len(x)))


def trapezium(h, x, f):
    return h * (
            f(x[0]) / 2 + sum(
        f(x[i]) for i in range(1, len(x) - 1)) + f(x[len(x) - 1]))


def simpson(h, x, f):
    # https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC
%D1%83%D0%BB%D0%B0_%D0%A1%D0%B8%D0%BC%D0%BF%D1%81%D0%BE%D0%BD
%D0%B0
    n = len(x)
    return (h / 3) * (f(x[0]) +
                      sum(4 * f(x[i]) for i in range(1, n - 1, 2))
+
                      sum(2 * f(x[i]) for i in range(2, n - 1, 2))
+
                      f(x[n - 1]))
```

Результаты

```
konstanze@G5-5587:~/PycharmProjects/num_met$ python3 lab3_5.py <
tests/3.5.txt
```

```
Метод прямоугольников
для h1: 1.8573277545230926
для h2: 1.8574186978554517
Погрешность:
Рунге-Ромберга         0.00018188666471807835
Разница с точным реш.  9.224547690744878e-05

Метод трапеций
для h1: 1.857373147579959
для h2: 1.8574413932181812
Погрешность:
Рунге-Ромберга         9.099418429621882e-05
Разница с точным реш.  4.685242004098811e-05

Метод Симпсона
для h1: 1.857297342723147
для h2: 1.8574186872187857
Погрешность:
Рунге-Ромберга         0.00012943412868130129
Разница с точным реш.  0.000122657276853122
```

Вывод

Проделав лабораторную работу, я изучил и реализовал методы прямоугольников, трапеций, Симпсона. Применил полученный алгоритм для вычисления определённого интеграла. Стоит отметить, что при оценке погрешности методом Рунге-Ромберга, следует использовать её с разными параметрами, для метода прямоугольников степень будет 1, для трапеций – 2, и для метода Симпсона – 4.

Московский авиационный институт

(Национальный исследовательский университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»

Лабораторная работа № 4

по курсу «Численные методы»

Методы решения начальных и краевых задач для обыкновенных дифференциальных уравнений (ОДУ) и систем ОДУ

Студент: Лукашкин К. В.

Группа: М80-308Б

Вариант: № 7

Преподаватель: Сластушенский Ю. В.

Москва, 2019

Задача

4.1. 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.

Вариант

$y'' - 4xy' + (4x^2 - 2)y = 0$, $y(0) = 1$, $y'(0) = 1$, $x \in [0,1], h = 0.1$	$y = (1 + x)e^{x^2}$

Алгоритм

```
def euler(x0, y0, z0, h, n):
    x, y, z = [x0], [y0], [z0]
    for k in range(n - 1):
        y.append(y[k] + h * f(x[k], y[k], z[k]))
        z.append(z[k] + h * g(x[k], y[k], z[k]))
        x.append(x[k] + h)
    return x, y, z



def adams(h, n, x, y, z):
    # метод Адамса
    # https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE
%D0%B4_%D0%90%D0%B4%D0%B0%D0%BC%D1%81%D0%B0
    for k in range(3, n - 1):
        del_y = 55 * f(x[k], y[k], z[k]) \
                - 59 * f(x[k - 1], y[k - 1], z[k - 1]) \
                + 37 * f(x[k - 2], y[k - 2], z[k - 2]) \
                - 9 * f(x[k - 3], y[k - 3], z[k - 3])
        del_z = 55 * g(x[k], y[k], z[k]) \
                - 59 * g(x[k - 1], y[k - 1], z[k - 1]) \
                + 37 * g(x[k - 2], y[k - 2], z[k - 2]) \
                - 9 * g(x[k - 3], y[k - 3], z[k - 3])
        y.append(y[k] + h / 24 * del_y)
        z.append(z[k] + h / 24 * del_z)
        x.append(x[k] + h)
```

```python
    return x, y, z


def runge_kutt(x0, y0, z0, f, g, h, n):
    # Метод Рунге-Кутта произвольного порядка для системы двух ДУ
    p = 4
    a = [0, 1 / 2, 1 / 2, 1]
    c = [1 / 6, 1 / 3, 1 / 3, 1 / 6]
    b = [
        [],
        [1 / 2, ],
        [0, 1 / 2],
        [0, 0, 1 / 2],
    ]

    x, y, z = [x0], [y0], [z0]
    fault = [0.]  # погрешность
    K = [0.] * p
    L = [0.] * p
    for k in range(n - 1):
        xk = x[k]
        yk = y[k]
        zk = z[k]
        del_y, del_z = 0, 0
        for i in range(p):
            # вычисляем значения K и L
            summ_K = 0
            summ_L = 0
            for j in range(i):
                summ_K += b[i][j] * K[j]
                summ_L += b[i][j] * L[j]
            K[i] = h * f(xk + a[i] * h,
                         yk + summ_K,
                         zk + summ_L)

            L[i] = h * g(xk + a[i] * h,
                         yk + summ_K,
                         zk + summ_L)
            # вычисляем значение дельта
            del_y += c[i] * K[i]
            del_z += c[i] * L[i]
        y.append(y[k] + del_y)
        z.append(z[k] + del_z)
```

```
        x.append(xk + h)
        # контроль точности решения методом Рунге - Ромберга - Ричардсона.
        # if abs(K[0] - K[1]) > 0 and abs((K[1] - K[2]) / (K[0] - K[1])) > 0.1:
        #     # шаг следует уменьшить
        #     h /= 2
        if abs(K[0] - K[1]) > 0:
            fault.append(((K[1] - K[2]) / (K[0] - K[1])))
        else:
            fault.append(0.)
    return x, y, z, fault
```

Результаты

```
konstanze@G5-5587:~/PycharmProjects/num_met$ python3 lab4_1.py < tests/4.1.txt
Решение методом Эйлера:
x 0.000000 0.100000 0.200000 0.300000 0.400000 0.500000 0.600000 0.700000 0.800000 0.900000 1.000000
y 1.000000 1.100000 1.220000 1.366360 1.546877 1.771464 2.053023 2.408607 2.861030 3.441094 4.190756
Промежуточные значения z = y'
z 1.000000 1.200000 1.463600 1.805168 2.245871 2.815586 3.555849 4.524223 5.800639 7.496626 9.768716
Сравнение с точным решением
  0.000000 0.011055 0.028973 0.056067 0.096038 0.154574 0.240305 0.366330 0.552636 0.829932 1.245807

Решение методом Рунге-Кутта:
x 0.000000 0.100000 0.200000 0.300000 0.400000 0.500000 0.600000 0.700000 0.800000 0.900000 1.000000
y 1.000000 1.109118 1.244182 1.413432 1.627701 1.901562 2.254967 2.715612 3.322438 4.130851 5.220630
Промежуточные значения z = y'
z 1.000000 1.229664 1.533060 1.932130 2.458758 3.158762 4.098002 5.371695 7.118666 9.543351 12.950032
Погрешность методом Рунге – Ромберга
  0.000000 0.154750 0.145506 0.143516 0.145175 0.148776 0.153459 0.158778 0.164500 0.170504 0.176728
Сравнение с точным решением
  0.000000 0.001937 0.004791 0.008995 0.015215 0.024476 0.038360 0.059326 0.091228 0.140174 0.215934

Решение методом Адамса:
```

```
x 0.000000 0.100000 0.200000 0.300000 0.400000 0.500000 0.600000
0.700000 0.800000 0.900000 1.000000
y 1.000000 1.109118 1.244182 1.413432 1.631407 1.911868 2.274998
2.750597 3.380637 4.225063 5.370907
Промежуточные значения z = y'
z 1.000000 1.229664 1.533060 1.932130 2.466361 3.181367 4.145061
5.459178 7.273071 9.807810 13.395226
Сравнение с точным решением
  0.000000 0.001937 0.004791 0.008995 0.011508 0.014170 0.018330
0.024341 0.033029 0.045962 0.065657
```

Вывод

Проделав лабораторную работу, я изучил и реализовал методы Эйлера, Рунге-Кутты и Адамса 4-го порядка. Применил полученный алгоритм для решения задачи Коши для ОДУ 2-го порядка на указанном отрезке. Стоит отметить, что в методе Рунге-Кутты, есть очень удобный способ определять погрешность Рунге-Ромберга: $\theta^k = \left|\frac{K_2^k - K_3^k}{K_1^k - K_2^k}\right|$

Задача

4.2. 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.

Вариант

$$\begin{cases} 15 \cdot x_1 + 8 \cdot x_2 = 92 \\ 2 \cdot x_1 - 15 \cdot x_2 + 4 \cdot x_3 = -84 \\ 4 \cdot x_2 + 11 \cdot x_3 + 5 \cdot x_4 = -77 \\ -3 \cdot x_3 + 16 \cdot x_4 + -7 \cdot x_5 = 15 \\ 3 \cdot x_4 + 8 \cdot x_5 = -11 \end{cases}$$

Алгоритм

```
def shooting(xa, xb, ya, yb, h, f, g):
    n = int(math.ceil((xb - xa) / h) + 1)
    eta = [1, 0.8]  # некоторое значение тангенса угла наклона касательной
    # к решению в точке a из [a,b]

    eps = 0.000001
    F = []
    for et in eta:
        # решаем задачу коши методом Рунге - Кутта
        x, y, z, _ = runge_kutt(xa, ya, et, f, g, h, n)
        F.append(y[-1] - yb)
    k = 2
    while True:
        # вычисляем новую 'эта'
        eta.append(
            eta[k - 1] - (eta[k - 1] - eta[k - 2]) /
            (F[k - 1] - F[k - 2]) * F[k - 1]
        )

        x, y, z, _ = runge_kutt(xa, ya, eta[k], f, g, h, n)
        F.append(y[-1] - yb)
        # проверяем удовлетворение условия
        if abs(F[k]) < eps:
```

```
            break
        k += 1
    return x, y, eta, F


def finite_diff(x, za, fb, h, n, second_approx=False):
    # Вслучае использования граничных условий второго и третьего
рода аппроксимация
    # производных проводится с помощью односторонних разностей
первого и второго порядков.
    # в случае с первым порядком -- ситстема будет трёхдиагональна

    if not second_approx:
        # составляем СЛАУ с неизвестными y[k]
        last = n - 1  # последний элемент
        a = [0.]
        b = [-1 / h]
        c = [1 / h]
        d = [-1]
        for k in range(1, last):
            a += [1 - p(x[k]) * h / 2]
            b += [-2 + (h ** 2) * q(x[k])]
            c += [1 + p(x[k]) * h / 2]
            d += [(h ** 2) * f(x[k])]
        a += [-1 / h]
        b += [2 + 1 / h]  # из условия y'(1) + 2y(1) = 3
        c += [0.]
        d += [fb]  # fb= 3
        # print("Полученная матрица ")
        # print('a ', a)
        # print('b ', b)
        # print('c ', c)
        # print('d ', d)
        y = tridiagonal(a, b, c, d, n)
    else:
        # если используем апроксимацию второго порядка система не
трёхдиагональна
        A = Matrix(rows=n, cols=n)
        d = [0.] * n
        A[0][0] = -3 / (2 * h)
        A[0][1] = 4 / (2 * h)
        A[0][2] = -1 / (2 * h)
        d[0] = -1
```

```
        last = n - 1
        A[last][last - 2] = 1 / (2 * h)
        A[last][last - 1] = -4 / (2 * h)
        A[last][last - 0] = 2 + 3 / (2 * h) # из условия y'(1) +
2y(1) = 3
        d[last] = fb
        for k in range(1, last):
            A[k][k - 1] = 1 - p(x[k]) * h / 2
            A[k][k - 0] = -2 + (h ** 2) * q(x[k])
            A[k][k + 1] = 1 + p(x[k]) * h / 2
            d[k] = (h ** 2) * f(x[k])
        y = np.linalg.solve(A.matrix, np.transpose([d]))
        y = np.transpose(y)[0]
    return y
```

Результаты

```
konstanze@G5-5587:~/PycharmProjects/num_met$ python3 lab4_2.py
Конечно-разностный метод
x        0.00000 0.10000 0.20000 0.30000 0.40000 0.50000 0.60000 0.70000 0.80000 0.90000 1.00000
y        1.01159 0.92817 0.87792 0.85485 0.85404 0.87148 0.90385 0.94846 1.00309 1.06592 1.13547
Точн.реш.1.00000 1.08020 1.12312 1.13527 1.12615 1.10653 1.08675 1.07531 1.07804 1.09790 1.13534
Погрешн. 0.01159 0.15203 0.24520 0.28042 0.27211 0.23505 0.18290 0.12685 0.07495 0.03198 0.00013
Погр. РР 0.27410 0.26595 0.26216 0.25871 0.25325 0.24441 0.23143 0.21393 0.19180 0.16509 0.13392

Метод стрельбы
x        0.00000 0.10000 0.20000 0.30000 0.40000 0.50000 0.60000 0.70000 0.80000 0.90000 1.00000
y        1.00000 0.91699 0.86767 0.84583 0.84641 0.86527 0.89904 0.94497 1.00083 1.06478 1.13534
Погрешн. 0.00000 0.16321 0.25544 0.28944 0.27974 0.24127 0.18772 0.13034 0.07721 0.03312 0.00000
Погр. РР 0.00000 0.00125 0.00190 0.00214 0.00210 0.00188 0.00156 0.00118 0.00077 0.00038 0.00000
  эта     f
+1.00000 3.10673
+0.80000 2.90831
-0.98708 1.13534
```

Вывод

Проделав лабораторную работу, я изучил и реализовал метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ. Применил полученный алгоритм для решения заданного вариантом задания. В качестве дополнительного задания в конечно разностном методе (поскольку условия краевой задачи не первого рода) я реализовал приближение первого и последнего элемента СЛАУдвумя способами: многочленом первой и второй степени.

Весь код, а также тестовые данные и отчёты, можно найти на моём Github по ссылке: https://github.com/memosiki/numerical_methods