МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)
ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И ПРИКЛАДНОЙ МАТЕМАТИКИ
КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И ПРОГРАММИРОВАНИЯ

ЛАБОРАТОРНЫЙ РАБОТЫ
ПО ДИСЦИПЛИНЕ «ЧИСЛЕННЫЕ МЕТОДЫ»
IV курс, VII семестр

Выполнил:
Студент группы М8О-305Б-19
Данилова Татьяна Михайловна
Номер по списку: 9
Преподаватель:
Демидова Ольга Львовна

Оценка: ____

Дата: 18.05.2022

Москва
2022

Оглавление

Лабораторная работа 1

Методы решения задач линейной алгебры

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;

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;

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;

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;

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.

1.1. LU-разложение матриц

Матрицу коэффициентов A представляют в виде произведения нижней треугольной матрицы L и верхней треугольной матрицы U: A = LU.

Используем модифицированный метод, а именно LUP-разложение.

LUP-разложение матрицы A — это представление матрицы A в виде произведения PA = LU, где L-нижняя треугольная матрица, U - верхняя треугольная или ступенчатая матрица, P-матрица перестановок (матрица перестановок - эта матрица, полученная из единичной матрицы перестановкой некоторых строк или столбцов).

Для получения LUP-разложения используют одну из модификаций метода Гаусса исключения неизвестных. Ядром её, как и для LU-разложения, остаются декомпозиционные преобразования, но каждому рекурсивному шагу предшествует операция перестановки местами двух строк соответствующей матрицы, обеспечивающая получение ненулевых ведущих элементов.

```
def LU_decomposition(A):
    LU = deepcopy(A)
    size = len(LU)
    P = [i for i in range(size)]
    n = 0
    for k in range(size):   #поиск главного элемента
```

```
        flag = 0
        for i in range(k, size):
            if abs(LU[i][k]) > flag:
                flag = abs(LU[i][k])
                n = i
        if flag == 0:
            print('Вырожденная матрица\n')
            return -1

        Swap(P, k ,n)
        Swap(LU, k, n)

        for i in range(k + 1, size):
            LU[i][k] = LU[i][k] / LU[k][k]
            for j in range(k + 1, size):
                LU[i][j] = LU[i][j] - LU[i][k] * LU[k][j]
    return LU, P
```

Решая СЛАУ с помощью матриц L и U, мы повышаем эффективность вычислений. Процесс решения СЛАУ сводится к двум простым этапам.

На первом этапе решается СЛАУ $Lz = b$. Поскольку матрица системы – нижняя треугольная, решение можно записать в явном виде:

$$z_1 = b_1,\ z_i = b_i - \sum_{j=1}^{i-1} l_{ij} z_j\ , i = 2, n$$

На втором этапе решается СЛАУ $Ux = c$ верхней треугольной матрицей. Здесь, как и на предыдущем этапе, решение представляется в явном виде:

$$x_n = \frac{z_n}{u_{nn}},\ x_i = \frac{(z_i - \sum_{j=i+1}^{n} u_{ij} x_j)}{u_{ii}}, i = n - 1,1$$

```
def LU_solve(LU, P, B):
    size = len(LU)

    X = [0 for i in range(size)]
    Y = [0 for i in range(size)]

    for i in range(size):
        if i == 0:
            Y[i] = B[P[i]]
        else:
            suma_y = sum(map(lambda u, y: u * y, LU[i][:i], Y[:i]))
            Y[i] = B[P[i]] - suma_y

    for i in range(size - 1, -1, -1):
        if i == size - 1:
            X[i] = Y[i] / LU[i][i]
        else:
            suma_x = sum(map(lambda l, x: l * x, LU[i][i + 1 :], X[i + 1 :]))
            X[i] = (Y[i] - suma_x) / LU[i][i]
    return X


def LU_solution(A, B):
    A_ = deepcopy(A)
    B_ = B
    A_, P = LU_decomposition(A_)
    X = LU_solve(A_, P, B_)
    return X
```

Обратная матрица:

```
def Inverse(A):
    A_ = deepcopy(A)
    E = [[1 if i == j else 0 for i in range (len(A))] for j in range(len(A))]
    inv =[]
    for i in range(len(E)):
        column = Get_column(E, i)
        column = Column_to_vec(column)
        inv.append(LU_solution(A_, column))
    inv = Transpose(inv)
    return inv
```

Определитель:

```
def Determinant(A):
    A_ = deepcopy(A)
    A_ = LU_decomposition(A_)[0]
    return reduce(lambda x, y: x * y, [A_[i][i] for i in range(len(A_))])
    #reduce(lambda x, y: x*y, [A[i][i]]) эквивалентно ((A[0][0]*A[1][1])*A[2][2])...
```

Вспомогательный код:

```
#Вывод матрицы
def Print_matrix(matrix):
    for i in range(len(matrix)):
        for j in range(len(matrix[i])):
            print('{:6.2f}'.format(float(matrix[i][j])), end=' ')
        print()
    print()

#Вывод вектора
def Print_vector(vec):
    for i in range(len(vec)):
        print('{:6.2f}'.format(float(vec[i])), end=' ')
    print('\n')

def Get_column(A, k):
    column = [[0] for i in range(len(A))]
    for i in range(len(A)):
        column[i][0] = A[i][k]
    return column

#Вектор из столбца
def Column_to_vec(column):
    vec = []
    for i in range (len(column)):
        vec.append(column[i][0])
    return vec

#Транспонирование
def Transpose(A):
    C = [[A[j][i] for j in range(len(A))]
    for i in range(len(A[0]))]
    return  C

def Swap(a, i, j):
    t = a[i]
    a[i] = a[j]
    a[j]= t

print('Решение СЛАУ:')
Print_vector(LU_solution(A1, B1))
#print(np.linalg.solve(A1, B1))

print('Обратная матрица:')
Print_matrix(Inverse(A1))
#print(np.linalg.inv(np.array(A1)))
```

```
print('Определитель:')
print(Determinant(A1))
#print(np.linalg.det(A1))
```

Результаты выполнения программы:

Решение СЛАУ:

-4.00 2.00 -8.00 -3.00

Обратная матрица:

0.06 -0.18 -0.15 0.05

-0.10 0.06 0.10 0.02

0.07 -0.06 0.03 -0.11

-0.05 0.07 0.02 -0.06

Определитель:

6284.99999999999

1.2. Метод прогонки

Дано:

$$\begin{cases} 8 \cdot x_1 - 4 \cdot x_2 = 32 \\ -2 \cdot x_1 + 12 \cdot x_2 - 7 \cdot x_3 = 15 \\ 2 \cdot x_2 - 9 \cdot x_3 + x_4 = -10 \\ -8 \cdot x_3 + 17 \cdot x_4 - 4 \cdot x_5 = 133 \\ -7 \cdot x_4 + 13 \cdot x_5 = -76 \end{cases}$$

Для трехдиагональных матриц можно применить метод прогонки.

$$\begin{cases} b_1 x_1 + c_1 x_2 = d_1, \\ a_2 x_1 + b_2 x_2 + c_2 x_3 = d_2, \\ \dots \\ a_{n-1} x_{n-2} + b_{n-1} x_{n-1} + c_{n-1} x_n = d_{n-1}, \\ a_n x_{n-1} + b_n x_n = d_n,\ c_n = 0 \end{cases}$$

$A_i, B_i, i = \overline{1,n}$ – прогоночные коэффициенты, подлежащие определению.

Прямой ход метода осуществляется по формулам:

$$A_1 = \frac{-c_1}{b_1}, \qquad B_1 = \frac{d_1}{b_1}$$

$$A_i = \frac{-c_i}{b_i + a_i A_{i-1}}, \qquad B_i = \frac{d_1 - a_i B_{i-1}}{b_i + a_i A_{i-1}}$$

$$A_n = 0\ (\text{т.к}\ c_i = 0), \qquad B_n = \frac{d_n - a_n B_{n-1}}{b_n + a_i A_{n-1}} = x_n$$

Обратный ход метода осуществляется по формулам:

$$\begin{cases} x_n = A_n x_{n+1} + B_n = 0 \cdot x_{n+1} + B_n = B_n, \\ x_{n-1} = A_{n-1} x_n + B_{n-1}, \\ x_{n-2} = A_{n-2} x_{n-1} + B_{n-2}, \\ \dots \\ x_1 = A_1 x_2 + B_1. \end{cases}$$

```
def Progon(A, B):
    if (not Chek(A)):
        print('Ошибка в исходных данных')
        return 0

    n = len(A)
    x = [0 for k in range(0, n)]

    #Прогоночные коэффициенты
    a = [0 for k in range(0, n)]
    b = [0 for k in range(0, n)]
    a[0] = -A[0][1] / A[0][0]
    b[0] = B[0] / A[0][0]
    for i in range(1, n - 1):
        a[i] = -A[i][i+1] / ( A[i][i] + A[i][i-1]*a[i-1] )
        b[i] = (B[i] - A[i][i-1]*b[i-1] ) / ( A[i][i] + A[i][i-1]*a[i-1] )
    a[n-1] = 0
    b[n-1] = (B[n-1] - A[n-1][n-2]*b[n-2]) / (A[n-1][n-1] + A[n-1][n-2]*a[n-2])

    #Обратный ход
    x[n-1] = b[n-1]
    for i in range(n-1, 0, -1):
        x[i-1] = a[i-1] * x[i] + b[i-1]
    return x
```

Достаточное условие:

Пусть коэффициенты СЛАУ удовлетворяют условию $|b_i| \geq |a_i| + |c_i|$, причем строгое неравенство должно выполняться хотя бы раз.

```
def Chek(a):
    n = len(a)
    k = 0
    for str in range(0, n):
        if( len(a[str]) != n ):
            print('Не соответствует размерность')
            return False
    for i in range(1, n - 1):
        if (abs(a[i][i]) >= abs(a[i][i-1]) + abs(a[i][i+1])):
            if (abs(a[i][i]) > abs(a[i][i-1]) + abs(a[i][i+1])):
                k += 1
        else:
            print('Не выполнены условия достаточности')
            return False
        if k == 0:
            print('Не выполнены условия достаточности')
            return False
    for stroka in range(0, len(a)):
        if( a[str][str] == 0 ):
            print('Нулевые элементы на главной диагонали')
            return False
    return True
```

Вызов функции:

```
print("Ответ:")
print("\n".join("X{0} =\t{1:10.2f}".format(i + 1, j) for i, j in enumerate(Progon(A2, B2))))
```

Результаты выполнения программы:

Ответ:

X1 = 6.00

X2 = 4.00

X3 = 3.00

X4 = 9.00

X5 = -1.00

1.3. Метод простых итераций и метод Зейделя

Дано:

$$\begin{cases} 12\cdot x_1 - 3\cdot x_2 - x_3 + 3\cdot x_4 = -31 \\ 5\cdot x_1 + 20\cdot x_2 + 9\cdot x_3 + x_4 = 90 \\ 6\cdot x_1 - 3\cdot x_2 - 21\cdot x_3 - 7\cdot x_4 = 119 \\ 8\cdot x_1 - 7\cdot x_2 + 3\cdot x_3 - 27\cdot x_4 = 71 \end{cases}$$

Метод простых итераций

Исходную СЛАУ можно привести к эквивалентному виду с помощью простейших преобразований. В общем случае эквивалентный вид выглядит:

$$\begin{cases} x_1 = \beta_1 + \alpha_{11}x_1 + \alpha_{12}x_2 + \cdots + \alpha_{1n}x_n, \\ x_2 = \beta_2 + \alpha_{21}x_1 + \alpha_{22}x_2 + \cdots + \alpha_{2n}x_n, \\ \cdots \\ x_n = \beta_n + \alpha_{n1}x_1 + \alpha_{n2}x_2 + \cdots + \alpha_{nn}x_n. \end{cases}$$

Эту систему можно разрешить относительно неизвестных при ненулевых диагональных элементах исходной матрицы. Тогда:

$$\beta_1 = \frac{b_i}{a_{ii}}, \qquad a_{ij} = \begin{cases} -\dfrac{a_{ij}}{a_{ii}}, i,j = \overline{1,n}, i \neq j \\ 0, \qquad i = j, i = \overline{1,n} \end{cases}$$

Тогда метод простых итераций имеет вид:

$$\begin{cases} x^{(0)} = \beta, \\ x^{(1)} = \beta + \alpha x^{(0)}, \\ \cdots \\ x^{(k)} = \beta + \alpha x^{(k-1)}. \end{cases}$$

Критерий остановки: $\left\| x^{(k)} - x^{(k-1)} \right\| \leq \varepsilon$

Достаточное условие сходимости: какая-либо норма эквивалентной матрицы меньше единицы.

```
def Iteration(A, B, eps):
    flag = False
    #Приведение СЛАУ к эквивалентному виду
    a = [[0 for j in range(len(A[0]))] for i in range(len(A))]
    b = [0 for i in range(len(B))]
    for i in range(len(A)):
        for j in range(len(A[0])):
            if i == j:
```

```
                a[i][j] = 0
            else:
                a[i][j] = -A[i][j]/A[i][i]
            if numpy.linalg.norm(a) < 1:    #Достаточное условие сходимости
                flag = True
        b[i] = B[i]/A[i][i]
    if flag == False:
        print('Достаточное условие сходимости не выполнено')
        return

    x1 = b
    x2 = list(numpy.array(list(numpy.dot(numpy.array(a), numpy.array(x1)))) + numpy.array(b))
    k = 0
    while numpy.linalg.norm(list(numpy.array(x1) - numpy.array(x2))) > eps:     #Условие выхода
        x1 = x2
        x2 = list(numpy.array(list(numpy.dot(numpy.array(a), numpy.array(x1)))) +
numpy.array(b))
        k += 1
    print("Ответ:")
    print("\n".join("X{0} =\t{1:10.2f}".format(i + 1, j) for i, j in enumerate(x1)))
    print("Колличество итераций: ", k)
```

Метод Зейделя

Для ускорения сходимости метода простых итераций существует метод Зейделя, заключающийся в использовании на текущей итерации результат уже вычисленных на этой итерации значений. Метод Зейделя имеет вид:

$$\begin{cases} x_1^{k+1} = \beta_1 + \alpha_{11}x_1^k + \alpha_{12}x_2^k + \cdots + \alpha_{1n}x_n^k, \\ x_2^{k+1} = \beta_2 + \alpha_{21}x_1^k + \alpha_{22}x_2^k + \cdots + \alpha_{2n}x_n^k, \\ \cdots \\ x_n^{k+1} = \beta_n + \alpha_{n1}x_1^k + \alpha_{n2}x_2^k + \cdots + \alpha_{nn}x_n^k. \end{cases}$$

```
def Seidel(A, B, eps):
    n = len(A)
    x = numpy.zeros(n)
    iter = 0
    converge = False
    k = 0
    while not converge:
        x_new = numpy.copy(x)
        for i in range(n):
            s1 = sum(A[i][j] * x_new[j] for j in range(i))
            s2 = sum(A[i][j] * x[j] for j in range(i + 1, n))
            x_new[i] = (B[i] - s1 - s2) / A[i][i]
        converge = numpy.linalg.norm(list(numpy.array(x_new) - numpy.array(x))) <= eps
        x = x_new
        k += 1
    print("Ответ:")
    print("\n".join("X{0} =\t{1:10.2f}".format(i + 1, j) for i, j in enumerate(x)))
    print("Колличество итераций: ", k)
```

Вспомогательный код:

```
print("Введите точность вычислений:")
eps = float(input())
print("Метод простых итераций:")
Iteration(A3, B3, eps)
print("Метод Зейделя:")
Seidel(A3, B3, eps)
```

Результаты выполнения программы:

Введите точность вычислений:

0.00001

Метод простых итераций:

Ответ:

X1 = 0.00

X2 = 7.00

X3 = -5.00

X4 = -5.00

Колличество итераций: 19

Метод Зейделя:

Ответ:

X1 = 0.00

X2 = 7.00

X3 = -5.00

X4 = -5.00

Колличество итераций: 14

1.4. Метод вращений

Дано:

$$\begin{pmatrix} 4 & 7 & -1 \\ 7 & -9 & -6 \\ -1 & -6 & -4 \end{pmatrix}$$

Метод вращений применим только для симметричных матриц и решает полную проблему собственных векторов и собственных значений для таких матриц.

Он основан на отыскании матрицы U:

$$A = U^{-1}AU$$

В начале алгоритма выбирается максимальный по модулю недиагональный элемент матрицы. Далее ставится задача найти такую ортогональную матрицу U, чтобы в результате преобразования подобия произошло обнуление максимального по модулю недиагонального элемента матрицы.

Угол вращения определяется по формуле:

$$\varphi^{(k)} = \frac{1}{2} arctg \frac{2a_{ij}^{(k)}}{a_{ii}^{(k)} - a_{jj}^{(k)}}$$

```
def find_id_max(A):
    i_max = j_max = 0
    elem_max = A[0][0]
    for i in range(len(A)):
        for j in range(i+1, len(A)):
            if abs(A[i][j]) > elem_max:
```

```
                elem_max = abs(A[i][j])
                i_max = i
                j_max = j
    return i_max, j_max


def find_phi(a_ii, a_jj, a_ij):
     return math.pi / 4 if a_ii == a_jj else \
            0.5 * math.atan(2 * a_ij / (a_ii - a_jj))
```

Критерий окончания:

$$t(A^{(k+1)})\sqrt{\left(\sum_{l,m;l<m}^{n}(a_{lm}^{(k+1)})^2\right)}$$

```
def find_t(A):
    C = math.sqrt(sum([A[i][j] ** 2 for i in range(len(A))
        for j in range(i + 1, len(A))]))
    return C

def Rotation(A, eps):
    size = len(A)
    eigen_vectors = [[1 if i == j else 0 for j in range(size)] for i in range(size)]
    while True:
        matrix_U = [[1 if i == j else 0 for j in range(size)] for i in range(size)]
        i, j = find_id_max(A)
        phi = find_phi(A[i][i], A[j][j], A[i][j])

        matrix_U[i][j] = -math.sin(phi)
        matrix_U[j][i] = math.sin(phi)
        matrix_U[i][i] = matrix_U[j][j] = math.cos(phi)
        matrix_UT = Transpose(matrix_U)
        A = Mul(Mul(matrix_UT,A), matrix_U)
        eigen_vectors = Mul(eigen_vectors, matrix_U)
        if find_t(A) < eps:
            break
    eigen_values = np.diagonal(A)
    return eigen_values, eigen_vectors
```

Вспомогательный код:

```
def Print_matrix(matrix):
    for i in range(len(matrix)):
        for j in range(len(matrix[i])):
            print('{:6.2f}'.format(float(matrix[i][j])), end=' ')
        print()
    print()

def Mul(A,B):
    C = []
    for i in range(0,len(A)):
        c_ = []
        for j in range(0,len(B[0])):
            elem = 0
            for k in range(0,len(B)):
                elem += A[i][k] * B[k][j]
            c_.append(elem)
        C.append(c_)
    return C
```

```
def Transpose(A):
    C = [[A[j][i] for j in range(len(A))]
    for i in range(len(A[0]))]
    return  C

val, vec = Rotation(A4, 0.001)
print('Собственные значения:')
print(val)
print('\nСобственные векторы:')
Print_matrix(vec)
```

Результаты выполнения программы:

Собственные значения:

[0.02021121 -14.52858454 -1.92222488]

Собственные векторы:

 0.03 -0.35 0.28

 0.03 0.82 -0.34

-0.04 0.45 0.84

1.5. Алгоритм QR – разложения матриц

Дано:

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

Данный метод используется для решения полной проблемы собственных значений для несимметричных матриц. Метод позволяет найти как вещественные, так и комплексные собственные значения.

QR-алгоритм основан по многократном построении QR-разложения матрицы.

Исходную матрицу можно представить в виде:

$$A = QR$$

где Q - ортогональная матрица, а R - верхняя треугольная.

Для построения данного разложения используется преобразование Хаусхолдера:

$$H = E - \frac{2}{v^T v} v v^T$$

```
def Find_housholder_matrix(column, size, k):
    v = np.zeros(size)
    column = np.array(Column_to_vec(column))
    v[k] = column[k] + Sign(column[k]) * norm(column[k:])

    for i in range(k + 1, size):
        v[i] = column[i]

    v = v[:, np.newaxis]
    H = np.eye(size) - (2 / (v.T.dot(v))) * (v.dot(v.T))
    H = H.tolist()
    return H
```

Данное преобразование позволит обнулять поддиагональные элементы матрицы:

$$A_0 = \begin{pmatrix} a_{11}^0 & a_{12}^0 & \cdots & a_{1n}^0 \\ a_{21}^0 & a_{22}^0 & \cdots & a_{2n}^0 \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1}^0 & a_{n2}^0 & \cdots & a_{nn}^0 \end{pmatrix} \xrightarrow{H_1} A_1 = \begin{pmatrix} a_{11}^0 & a_{12}^0 & \cdots & a_{1n}^0 \\ 0 & a_{22}^0 & \cdots & a_{2n}^0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & a_{n2}^0 & \cdots & a_{nn}^0 \end{pmatrix}$$

Компоненты вектора вычисляются так:

$$v_1^1 = a_{11}^0 + sign(a_{11}^0)\sqrt{\left(\sum_{j=1}^{n}(a_{j1}^0)^2\right)}$$

$$v_i^1 = a_{i1}^0, i = \overline{2,n}$$

На следующем шаге строится новая матрица, обнуляющая следующие поддиагональные элементы:

$$v_1^2 = 0$$

$$v_2^2 = a_{22}^0 + sign(a_{22}^1)\sqrt{\left(\sum_{j=2}^{n}(a_{j2}^1)^2\right)}$$

$$v_i^2 = a_{i1}^1, i = \overline{3,n}$$

Повторяя процесс n-1 раз, получим искомое разложение, где:

$$Q = (H_{n-1}H_{n-2}\cdots H_0)^T = H_1H_2\cdots H_{n-1}, R = A_{n-1}$$

Итерационный процесс для QR-алгоритма выглядит так:

$A^{(k)} = Q^{(k)}R^{(k)}$ – разложение,

$A^{(k+1)} = R^{(k)}RQ^{(k)}$ – перемножение.

```
def Find_QR(A):
    size = len(A)
    Q =  [[1 if i == j else 0 for j in range(size)] for i in range(size)]
    A_i = A
    for i in range(size - 1):
        column = Get_column(A_i, i)
        H = Find_housholder_matrix(column, len(A_i), i)
        Q = Mul(Q, H)
        A_i = Mul(H, A_i)
    return Q, A_i
```

В случае отсутствия комплексных собственных значений алгоритм сводит исходную матрицу к треугольной матрице, иначе к квазитреугольной, где каждой комплексно-сопряженной паре соответствует блок 2 * 2. Для вещественных значений критерий сходимости

$$\sqrt{\left(\sum_{l=m+1}^{n}(a_{lm}^{(k)})^2\right)} \leq \varepsilon$$

Для комплексной пары

$$\left|\lambda^{(k)} - \lambda^{(k-1)}\right| \leq \varepsilon$$

значения которых находятся из

$$(a_{jj}^{(k)} - \lambda^{(k)})(a_{j+1j+1}^{(k)} - \lambda^{(k)}) = a_{jj+1}^{(k)} a_{j+1j}^{(k)}$$

```
def Get_roots_complex(A, i):
    sz = len(A)
    A11 = A[i][i]
    A12 = A[i][i + 1] if i + 1 < sz else 0
    A21 = A[i + 1][i] if i + 1 < sz else 0
    A22 = A[i + 1][i + 1] if i + 1 < sz else 0
    return np.roots((1, -A11 - A22, A11 * A22 - A12 * A21))

def Finish_iter_complex(A, eps, i):
    Q, R = Find_QR(A)
    A_next = Mul(R, Q)
    l1 = Get_roots_complex(A, i)
    l2 = Get_roots_complex(A_next, i)
    return True if abs(l1[0] - l2[0]) <= eps and \
            abs(l1[1] - l2[1]) <= eps else False

def Find_eigenval (A, eps, i):
    A_i = A
    while True:
        Q,R = Find_QR(A_i)
        A_i = Mul(R, Q)
        a = np.array(A_i)
        if norm(a[i + 1:, i]) <= eps:
            res = (a[i][i], False, A_i)
            break
        elif norm(a[i + 2:, i]) <= eps and Finish_iter_complex(A_i, eps, i):
            res = (Get_roots_complex(A_i, i), True, A_i)
            break
    return res
```

Нахождение собственных значений:

```
def QR (A, eps):
    res = []
    i = 0
    A_i = A
    while i < len(A):
        eigenval = Find_eigenval(A_i, eps, i)
        if eigenval[1]:
            res.extend(eigenval[0])
            i+=2
        else:
            res.append(eigenval[0])
            i+=1
        A_i = eigenval[2]
    return res

print('\n Собственные значения матрицы:\n')
print(QR(A5, 0.01))
```

Вспомогательный код:

```
def Mul(A,B):
    C = []

    for i in range(0,len(A)):
        c_ = []
        for j in range(0,len(B[0])):
```

```
            elem = 0
            for k in range(0,len(B)):
                elem += A[i][k] * B[k][j]
            c_.append(elem)
        C.append(c_)
    return C

def Transpose(A):
    C = [[A[j][i] for j in range(len(A))]
    for i in range(len(A[0]))]
    return  C

def Column_to_vec(column):
    vec = []
    for i in range (len(column)):
        vec.append(column[i][0])
    return vec

def Vec_to_column(vec):
    column = [[0] for i in range(len(vec))]
    for i in range(0, len(vec)):
        column[i][0] = vec[i]
    return column

def Get_column(A, k):
    column = [[0] for i in range(len(A))]
    for i in range(len(A)):
        column[i][0] = A[i][k]
    return column

def Sign(x):

    return -1 if x < 0 else 1 if x > 0 else 0
```

Результаты выполнения программы:

Собственные значения матрицы:

[(-6.21681507253468+4.710162688932027j), (-6.21681507253468-4.710162688932027j), 3.4519573240337187]

Лабораторная работа 2

Методы решения нелинейных уравнений и систем нелинейных уравнений

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;

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.

2.1. Методы простой итерации и Ньютона решения нелинейных уравнений

Метод простой итерации:

Метод простой итерации уточнения корней уравнения состоит в замене этого уравнения эквивалентным ему уравнением

$$x = \varphi(x),\ a < x < b$$

и построением последовательности

$$x_{k+1} = \varphi(x_k),\ k = 0, 1, 2, \dots,$$

где $a < x_0 < b$.

Достаточные условия сходимости метода простых итераций:

Пусть функция $\varphi(x)$ в эквивалентном уравнении определена и дифференцируема на отрезке $a \le x \le b$. Тогда, если существует число q такое, что

$$\left|\varphi'(x_k)\right| \le q < 1$$

на отрезке $a \le x \le b$, то последовательность сходится к единственному корню уравнения при любом начальном приближении $a \le x \le b$.

Метод Ньютона:

Пусть имеется значение корня на k-ой итерации - x_k. Тогда значение корня на (k + 1)-й итерации вычисляется следующим образом:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)},\ k = 0, 1, 2, \dots$$

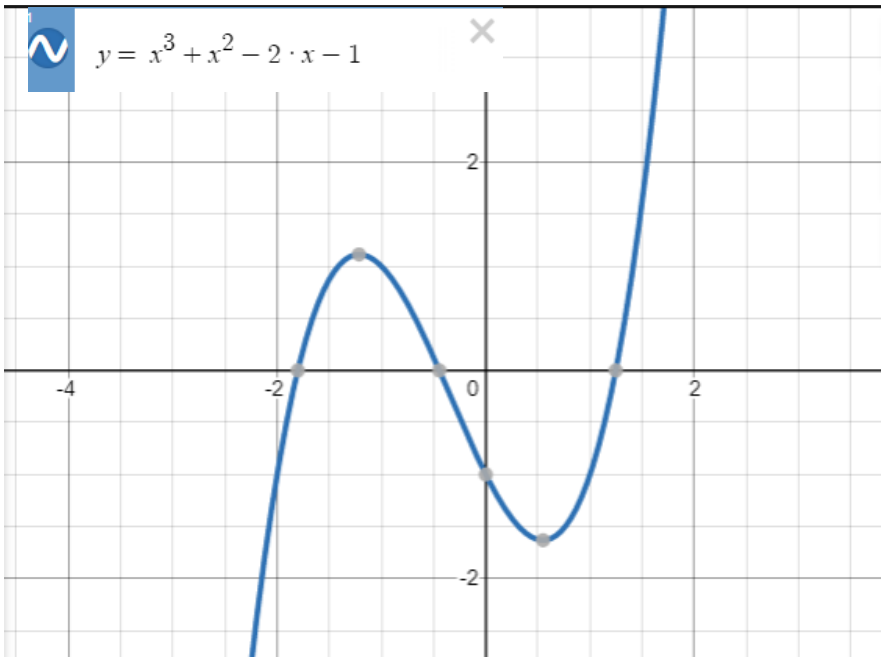
Достаточные условия сходимости метода Ньютона:

Пусть $f(x)$ определена и дважды дифференцируема на отрезке a ≤ x ≤ b, причем f(a) * f(b) < 1, производные $f'(x)$ и $f''(x)$ знакопостоянны и $f'(x) \neq 0$. Тогда исход из начально приближения, удовлетворяющего неравенству $f(x_0) * f''(x_0) > 0$, можно построить последовательность, сходящуюся к единственному корню.

Решение:

$$x^3 + x^2 - 2x - 1 = 0.$$

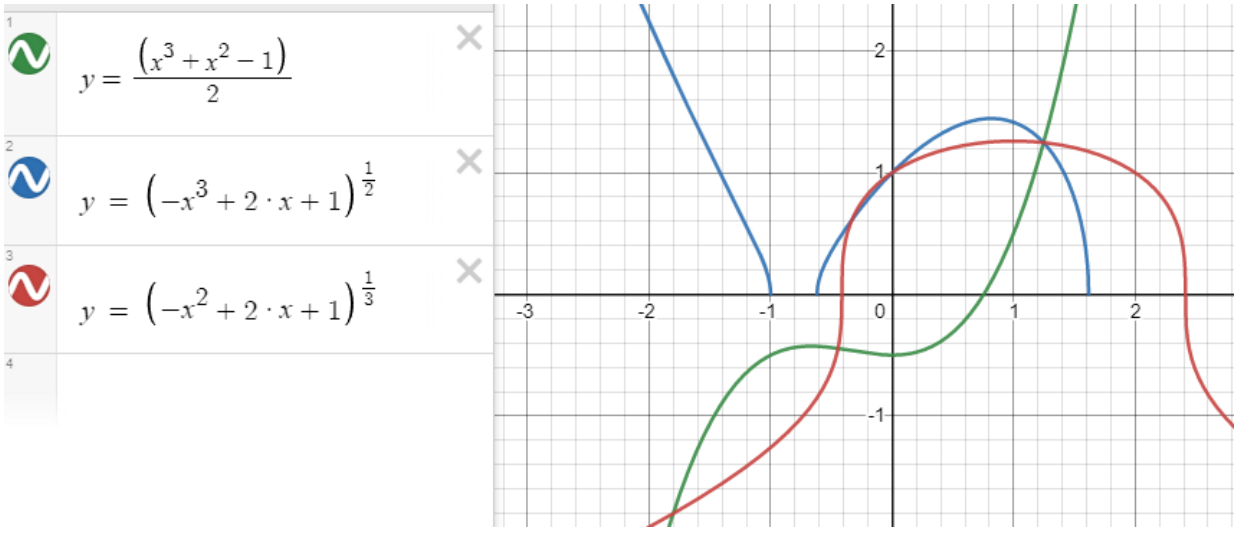
Для обоих методов начально приближение определяем графически:



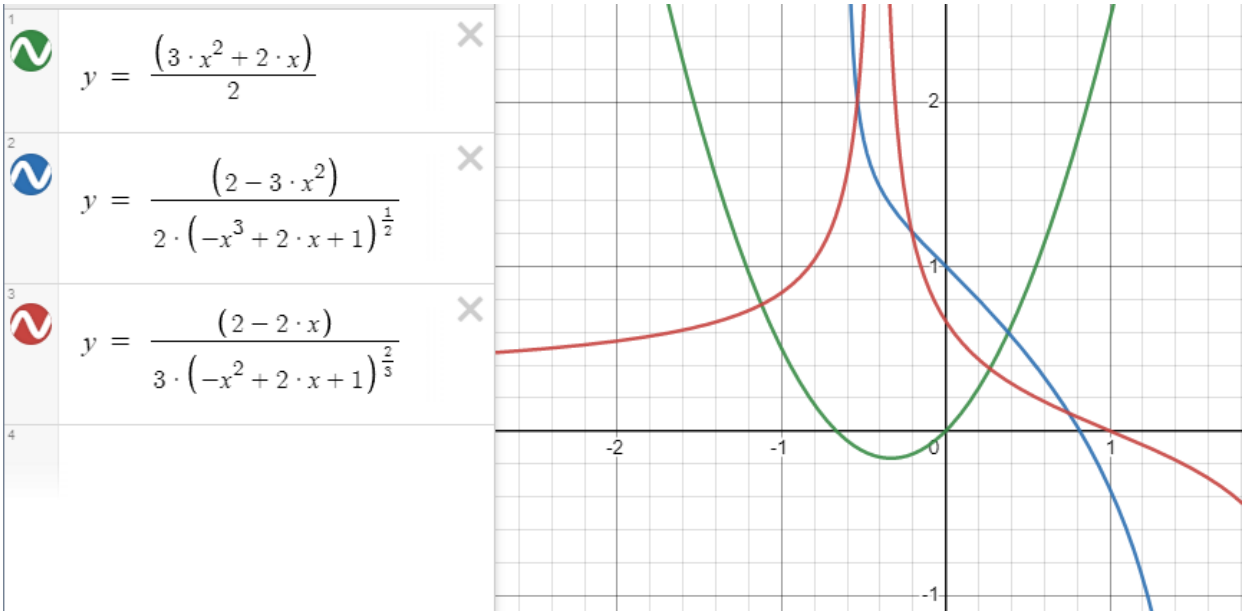
Таким образом, положительный корень существует на отрезке от 1 до 2.

Метод простой итерации:

Выразим $x = \varphi(x)$ и построим графики для каждой полученной функции.



По достаточному условию сходимости необходимо, чтобы значение первой производной функции в рассматриваемом отрезке было по модулю меньше единицы.



На заданном отрезке выражаем третью функцию.

```
def simple_iteration1(ph, dph, a, b, eps):

    x = (a + b) / 2
    k = 0
    cov = True
    while cov:
        if (abs(dph(x)) >= 1):    #Достаточное условие сходимости
            print('Достаточное условие сходимости не выполнено')
            return
        k += 1
        x_next = ph(x)
        print(f"x: {x_next} k: {k} |x_next - x|: {abs(x_next - x)}")
        if abs(x_next - x) <= eps:     #Условие выхода
            cov = False
        x = x_next
```

Метод Ньютона:

```
def newton1(f, df, ddf, a, b, exp):
    if (f(a) * ddf(a) < (df(a)) ** 2):
        x = a
    else:
        if (f(b) * ddf(b) < (df(b)) ** 2):
            x = b
        else:
            print('Достаточное условие сходимости не выполнено')
            return
    k = 0
    cov = True
    while cov:
        if (df(x) == 0):
            print('Достаточное условие сходимости не выполнено')
            return
        k += 1
        x_next = x - f(x) / df(x)
        print(f"x: {x_next} k: {k} |x_next - x|: {abs(x_next - x)}")
        if abs(x_next - x) <= exp:
            cov = False


        x = x_next
```

Вспомогательный код:

```
#Нелинейное уравнение

def fun(x):
    return x ** 3 + x ** 2 - 2 * x - 1

def dfun(x):
    return 3 * (x ** 2) + 2 * x - 2

def ddfun(x):
    return 6 * x + 2

def phi3(x):
    if x < 0:
        return abs((-x ** 2 + 2 * x + 1)) ** (1/3)*(-1)
    else:
        return (-x ** 2 + 2 * x + 1) ** (1/3)

def dphi3(x):
    return (2 - 2 * x) / (3 * ((-x ** 2 + 2 * x + 1) ** (2/3)))
```

Результаты выполнения программы:

Входные значения: a = 1, b = 2, exp = 0.00001

Метод простой итерации:

x: 1.205071132087615 k: 1 |x_next - x|: 0.29492886791238493

x: 1.251027598901561 k: 2 |x_next - x|: 0.04595646681394583

x: 1.2465473027318688 k: 3 |x_next - x|: 0.004480296169692144

x: 1.2470253379270608 k: 4 |x_next - x|: 0.0004780351951920725

x: 1.2469747604973989 k: 5 |x_next - x|: 5.0577429661968765e-05

x: 1.2469801165562924 k: 6 |x_next - x|: 5.356058893513094e-06

Метод Ньютона:

x: 1.3333333333333333 k: 1 |x_next - x|: 0.33333333333333326

x: 1.2530864197530864 k: 2 |x_next - x|: 0.08024691358024683

x: 1.2470135821315298 k: 3 |x_next - x|: 0.006072837621556637

x: 1.246979604778425 k: 4 |x_next - x|: 3.397735310484151e-05

x: 1.246979603717467 k: 5 |x_next - x|: 1.0609579881304398e-09

2.2. Методы простой итерации и Ньютона решения систем нелинейных уравнений

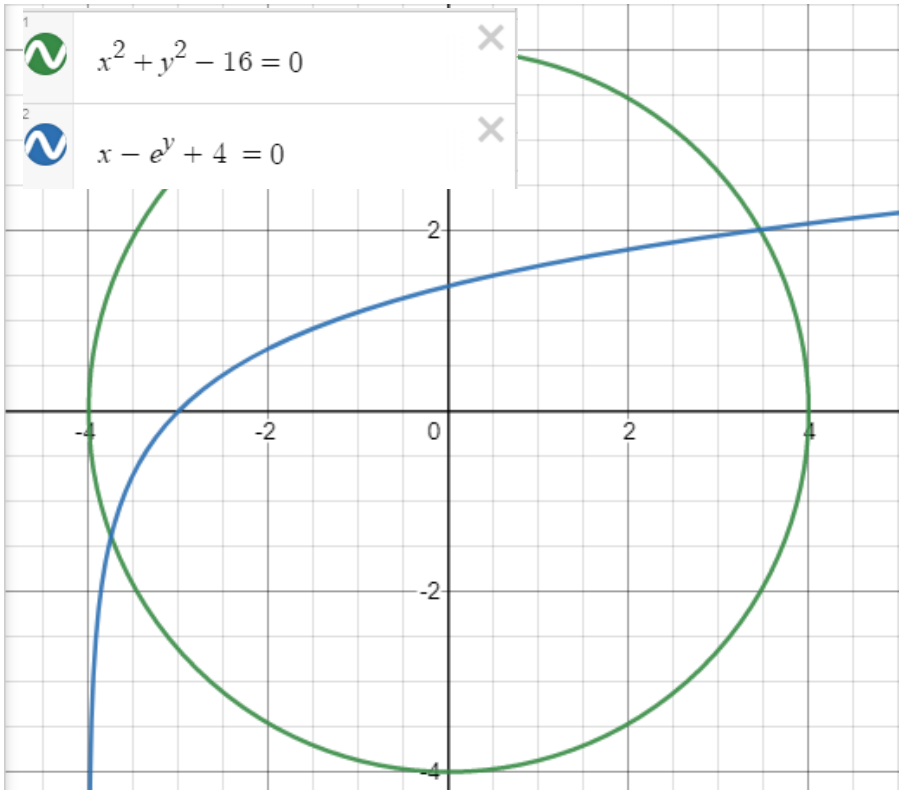
Дано:

$$\begin{cases} x_1^2 + x_2^2 - 16 = 0, \\ x_1 - e^{x_2} + 4 = 0. \end{cases}$$

Соответствующая эквивалентная система имеет вид:

$$\begin{cases} x_1 = \sqrt{-x_2^2 + 16}, \\ x_2 = \ln(x_1 + 4). \end{cases}$$

Для обоих методов начально приближение определяем графически:



Начальное приближение для положительного корня: $x_1 = 3.5, x_2 = 2$.

Метод простой итерации:

```
def simple_iteration2(x1, x2, eps):
    x1_next = x1
    x2_next = x2
    k = 0
    while True:
        J = Jacobi(dph1_dx1(x1, x2), dph1_dx2(x1, x2),dph2_dx1(x1, x2), dph2_dx2(x1, x2))
        if norm_matrix(J) >= 1:
            print('Достаточное условие сходимости не выполнено')
            return
        k += 1
        x1_next, x2_next = ph1(x1, x2), ph2(x1, x2)
        print(f"x1: {x1_next} x2: {x2_next} k: {k}")
        if norm(x1_next - x1, x2_next - x2) <= eps:
             break

        x1, x2 = x1_next, x2_next
```

Метод Ньютона:

```
def newton2(x1, x2, eps):
```

```
    x1_next = x1
    x2_next = x2
    k = 0
    while True:
        J = Jacobi(df1_dx1(x1, x2), df1_dx2(x1, x2), df2_dx1(x1, x2), df2_dx2(x1, x2))
        if det2(J) == 0:
            print('Достаточное условие сходимости не выполнено')
        k += 1
        [[x1_next],[x2_next]] = [[x1],[x2]] - numpy.dot(inv2(J), [[f1(x1, x2)], [f2(x1, x2)]])
        print(f"x1: {x1_next} x2: {x2_next} k: {k}")
        if norm(x1_next - x1, x2_next - x2) <= eps:
            break

        x1, x2 = x1_next, x2_next
```

Вспомогательный код:

```
#Первое уравнение системы
def f1(x1, x2):
    return x1 ** 2  + x2 ** 2 - 16

def df1_dx1(x1, x2):
    return 2 * x1

def df1_dx2(x1, x2):
    return 2 * x2

def ph1(x1, x2):
    return (-x2 ** 2 + 16) ** (1/2)

def dph1_dx1(x1, x2):
    return 0

def dph1_dx2(x1, x2):
    return x2 / ((x2 ** 2 + 16) ** (1/2))

#Второе уравнение системы
def f2(x1, x2):
    return x1 - (numpy.exp(x2)) + 4

def df2_dx1(x1, x2):
    return 1

def df2_dx2(x1, x2):
    return -(numpy.exp(x2))

def ph2(x1, x2):
    return math.log(x1 + 4)

def dph2_dx1(x1, x2):
    return 1/(x1 + 4)

def dph2_dx2(x1, x2):
    return 0

def Jacobi(a11, a12, a21, a22):
       return [
    [a11, a12],
    [a21, a22]
    ]

def norm(a ,b):
  return (a ** 2 + b ** 2) ** (1/2)

def norm_matrix(A):
```

```
  n = 0
  for i in range(len(A)):
    for j in range (len(A)):
      n = n + (A[i][j]) ** 2
  return n **(1/2)

def det2(A):
  return A[0][0] * A[1][1] - A[1][0] * A[0][1]

def inv2(A):
  detA = det2(A)
  B = A
  B[0][0] = (1/detA) * A[1][1]
  B[1][0] = -(1/detA) * A[1][0]
  B[0][1] = -(1/detA) * A[0][1]
  B[1][1] = (1/detA) * A[0][0]

  return B

#Меню
while True:
    print("\nЗадания:")
    print("1. Метод простой итерации решения нелинейных уравнений")
    print("2. Метод Ньютона решения нелинейных уравнений")
    print("3. Метод простой итерации решения систем нелинейных уравнений")
    print("4. Метод Ньютона решения систем нелинейных уравнений")
    print("\nВведите номер задания:")
    a = input()
    #print("\nВведите точность вычислений:")
    #eps = float(input())
    eps = 0.00001
    if a == '1': simple_iteration1(phi3, dphi3, 1, 2, eps)
    elif a == '2': newton1(fun, dfun, ddfun, 1, 2, eps)
    elif a == '3': simple_iteration2(3.5, 2, eps)
    elif a == '4': newton2(3.5, 2, eps)
    elif a == '0': break
    else: print("Неверный номер. Повторите попытку, либо введите '0' для выхода из программы.")
```

Результаты выполнения программы:

Входные значения: x1 = 3.5, x2 = 2, exp = 0.000001

Метод простой итерации:

x: 1.205071132087615 k: 1 |x_next - x|: 0.29492886791238493

x: 1.251027598901561 k: 2 |x_next - x|: 0.04595646681394583

x: 1.2465473027318688 k: 3 |x_next - x|: 0.004480296169692144

x: 1.2470253379270608 k: 4 |x_next - x|: 0.0004780351951920725

x: 1.2469747604973989 k: 5 |x_next - x|: 5.057742966196876e-05

x: 1.2469801165562924 k: 6 |x_next - x|: 5.356058893513094e-06

Метод Ньютона:

x: 1.3333333333333333 k: 1 |x_next - x|: 0.33333333333333326

x: 1.2530864197530864 k: 2 |x_next - x|: 0.08024691358024683

x: 1.2470135821315298 k: 3 |x_next - x|: 0.006072837621556637

x: 1.246979604778425 k: 4 |x_next - x|: 3.397735310484151e-05

x: 1.246979603717467 k: 5 |x_next - x|: 1.0609579881304398e-09

Выводы:

На основе проделанной работы можно сделать вывод: чем меньше погрешность, тем большее количество итераций требуется для поиска корня.

Также, сравнивая метод простой итерации и метод Ньютона для решения нелинейных уравнений и систем нелинейных уравнений, можно сказать, что для одинаковой точности вычисления метод Ньютона требует меньшее число итераций.

Лабораторная работа 3

Методы приближения функций. Численное дифференцирование и интегрирование

3.1. 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^*.

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 $x = X^*$.

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.

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

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.

3.1. Интерполяционные многочлены Лагранжа и Ньютона

Дано:

$y = \arccos(x)$

А) $X_i = -0.4, -0.1, 0.2, 0.5$

Б) $X_i = -0.4, 0, 0.2, 0.5$

$X^* = 0.1$

Многочлен Лагранжа

Интерполяционный многочлен Лагранжа n-й степени:

$$L_n(x) = \sum_{i=0}^{n} y_i \frac{(x - x_0)(x - x_1) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_n)}{(x_i - x_0)(x_i - x_1) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_n)}$$

Запишем вид многочлена Лагранжа третьей степени:

$$L_3(x) = y_0 \frac{(x - x_1)(x - x_2)(x - x_3)}{(x_0 - x_1)(x_0 - x_2)(x_0 - x_3)} + y_1 \frac{(x - x_0)(x - x_2)(x - x_3)}{(x_1 - x_0)(x_1 - x_2)(x_1 - x_3)} + y_2 \frac{(x - x_0)(x - x_2)(x - x_3)}{(x_2 - x_0)(x_2 - x_1)(x_2 - x_3)} + y_3 \frac{(x - x_0)(x - x_2)(x - x_3)}{(x_3 - x_0)(x_3 - x_1)(x_3 - x_2)}$$

Введем вспомогательную функцию: $\omega_i(x) = \prod_{j=0, j \neq i}^{n} (x - x_j)$

Абсолютную ошибку вычислим по формуле: $|L_3(x^*) - f(x^*)|$.

```
def omega(i, X, x):
    res = 1
    for j in range(len(X)):
        if i != j:
            res *= x - X[j]
    return res

def L(x, X):
    l = 0
    for i in range(len(X)):
        fomega = f(X[i]) / omega(i, X, X[i])
        l += fomega * omega(i, X, x)
    return l
```

Многочлен Ньютона

Введем понятие раздельной разности. Раздельная разность нулевого порядка совпадает со значением функции в узле.

Разделенная разность нулевого порядка: $f(x_i, x_j) = \frac{f_i - f_j}{x_i - x_j}$

Разделенная разность n – k + 2 порядка: $f(x_i, x_j, x_k, \dots, x_l, x_n) = \frac{f(x_i, x_j, x_k, \dots, x_l) - f(x_j, x_k, \dots, x_n)}{x_i - x_n}$.

Интерполяционный многочлен Ньютона n-й степени:

$$P_n(x) = f(x_0) + (x - x_0)f(x_0, x_1) + (x - x_0)(x - x_1)f(x_0, x_1, x_2) + \cdots +$$

$$+(x - x_0)(x - x_1)\cdots(x - x_n)f(x_0, x_1, \cdots, x_n).$$

Запишем вид многочлена Ньютона третьей степени:

$$P_3(x) = f(x_0) + (x - x_0)f(x_0, x_1) + (x - x_0)(x - x_1)f(x_0, x_1, x_2) + (x - x_0)(x - x_1)(x - x_2)f(x_0, x_1, x_2, x_3).$$

```
def X_new(i, k, X):
    return [X[j] for j in range(i, k)]

#Разделенная разность
def separate(X):
    if len(X) == 2:
        return (f(X[0]) - f(X[1])) / (X[0] - X[1])
    else:
        return (separate(X_new(0, len(X) - 1, X)) - separate(X_new(1, len(X), X))) / (X[0] -
X[len(X) - 1])

def xxx(x, i, X):
    res = 1
    for j in range(i):
        res *= (x - X[j])
    return res

def P(x, X):
    p = f(X[0])
    for i in range(1, len(X)):
        X_ = X_new(0, i + 1, X)
        p += xxx(x, i, X) * separate(X_)
    return p
```

Вспомогательный код:

```
def f(x):
```

```
    return math.acos(x)

X1 = numpy.array([-0.4, -0.1, 0.2, 0.5])
X2 = numpy.array([-0.4, 0, 0.2, 0.5])
X = 0.1

print("1. Пункт а) -0.4, -0.1, 0.2, 0.2")
print("2. Пункт б) -0.4, 0, 0.2, 0.5")
print("\nВведите номер задания:")
a = input()
if a == '1': X_a = X1
elif a == '2': X_a = X2
print("МНОГОЧЛЕН ЛАГРАНЖА\n")
print("\tf(x*) = {0}\n\tL(x*) = {1}\n\tПогрешность: {2}\n".format(f(X), L(X, X_a), f(X) - L(X,
X_a)))
print("МНОГОЧЛЕН НЬЮТОНА\n")
print("\tf(x*) = {0}\n\tP(x*) = {1}\n\tПогрешность: {2}\n".format(f(X), P(X, X_a), f(X) - P(X,
X_a)))

xmin = -0.5
xmax = 0.6
dx = 0.01
xarr = numpy.arange(xmin, xmax, dx)
ylist = [f(x) for x in xarr]
y_X_a = [f(x) for x in X_a]

Larr = X_a
Llist = [L(x, X_a) for x in Larr]
Parr = X_a
Plist = [P(x, X_a) for x in Parr]

plt.figure(figsize=(12, 8))
plt.plot(xarr, ylist)
plt.plot(Larr, Llist, linestyle = '--')
plt.plot(Parr, Plist, linestyle = ':')
plt.plot(X_a, y_X_a, 'o')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend(['f(x)', 'L(x)', 'P(x)', 'X'])
```

```
plt.show()
```**Результаты выполнения программы:**

1. Пункт а) -0.4, -0.1, 0.2, 0.2

2. Пункт б) -0.4, 0, 0.2, 0.5

Введите номер задания: 1

МНОГОЧЛЕН ЛАГРАНЖА

f(x*) = 1.4706289056333368

L(x*) = 1.4707404487843843

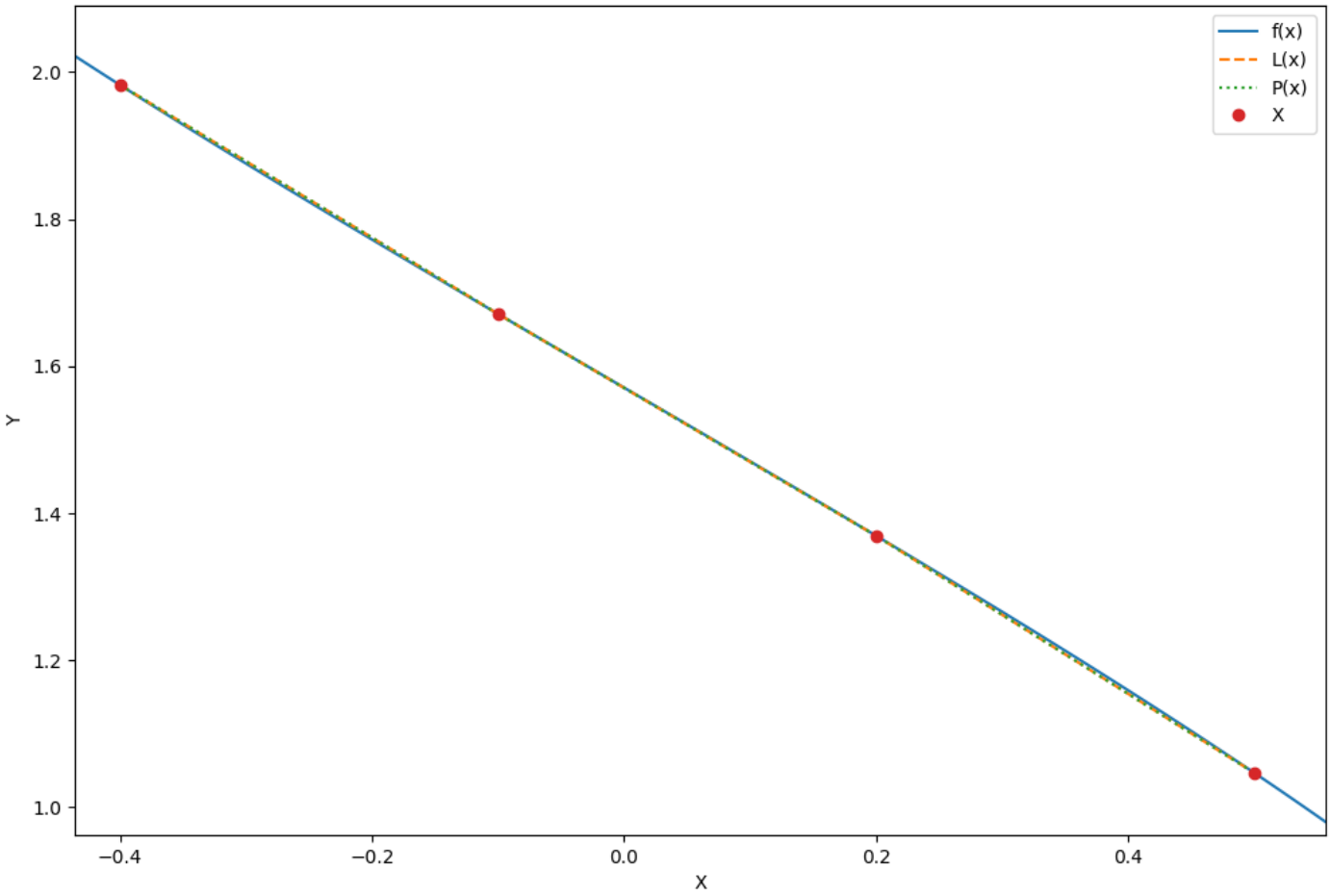
Погрешность: -0.00011154315104744406

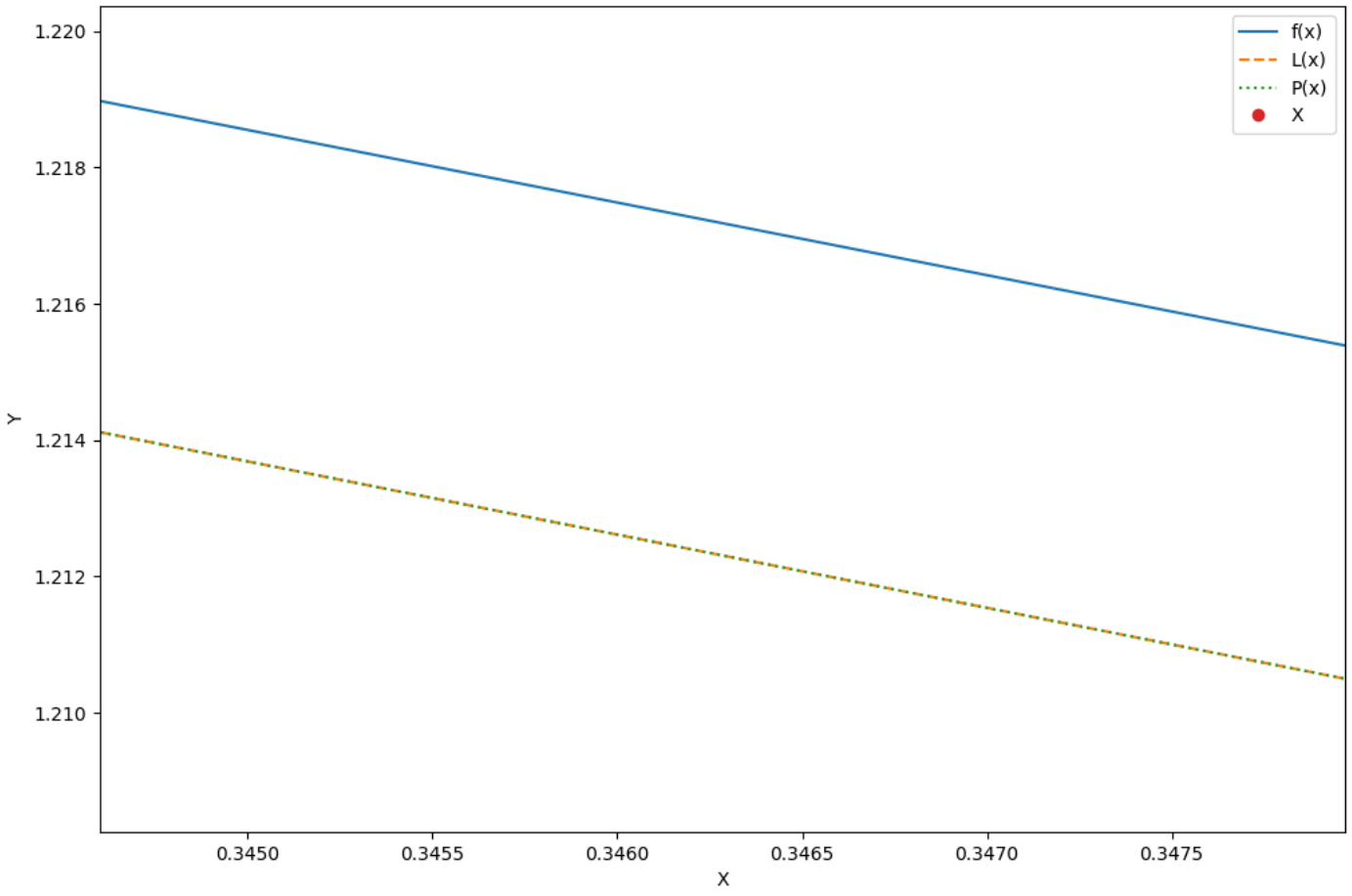
МНОГОЧЛЕН НЬЮТОНА

f(x*) = 1.4706289056333368

P(x*) = 1.4707404487843845

Погрешность: -0.0001115431510476661


f(x)
L(x)
P(x)
X
2.0
1.8
1.6
1.4
1.2
1.0
Y
−0.4
−0.2
0.0
0.2
0.4
X


f(x)
L(x)
P(x)
X
1.220
1.218
1.216
1.214
1.212
1.210
Y
0.3450
0.3455
0.3460
0.3465
0.3470
0.3475
X

1. Пункт а) -0.4, -0.1, 0.2, 0.2

2. Пункт б) -0.4, 0, 0.2, 0.5

Введите номер задания: 2

МНОГОЧЛЕН ЛАГРАНЖА

f(x*) = 1.4706289056333368

L(x*) = 1.470702680154511

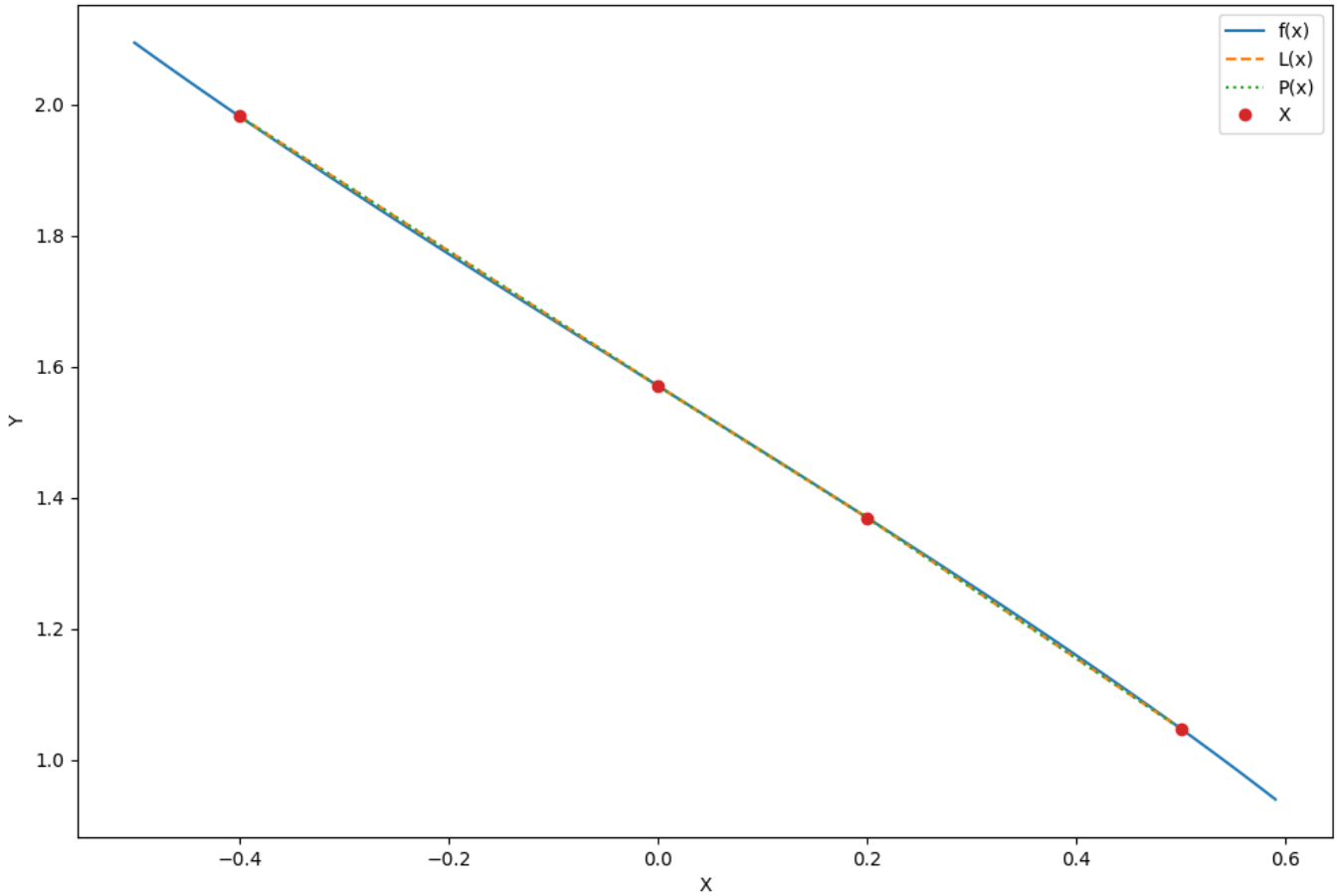
Погрешность: -7.377452117429684e-05

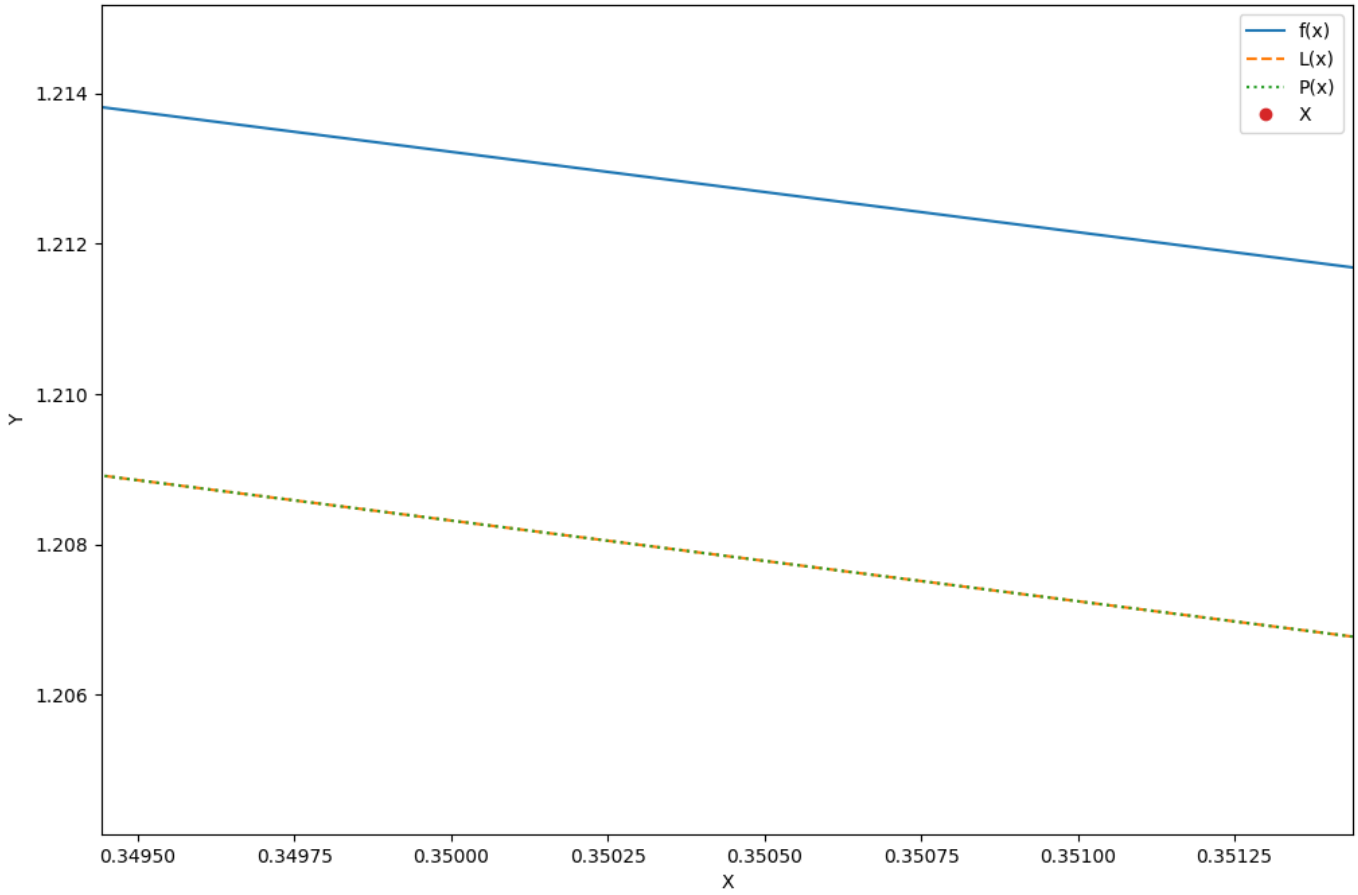
МНОГОЧЛЕН НЬЮТОНА

f(x*) = 1.4706289056333368

P(x*) = 1.470702680154511

Погрешность: -7.377452117429684e-05





3.2. Кубический сплайн

Дано:

$X^* = 0.1$

i	0	1	2	3	4
x_i	-0.4	-0.1	0.2	0.5	0.8
f_i	1.9823	1.6710	1.3694	1.0472	0.64350

Интерполяций, использующая сразу все n узлов таблицы, называется глобальной интерполяцией.

Начиная с $n \geq 7$ глобальная многочленная интерполяция становится неустойчивой, поэтому обычную многочленную интерполяцию осуществляют максимум по 3-4 узлам (для 3-х узлов 2-ой степени, 4-х узлов 3-й степени).

От этих недостатков свободна сплайн-интерполяция, которая требует непрерывности в узлах стыковки локальных многочленов по производным соответственно порядка один, два и т. д.

Сплайном степени m дефекта r называется (m - r) раз непрерывно дифференцируемая функция, которая на каждом отрезке представляет собой многочлен степени m.

Кубический сплайн - кусочно-заданный интерполяционный многочлен третьей степени:

$$S_3(x) = a_i + b_i(x - x_{i-1}) + c_i(x - x_{i-1})^2 + d_i(x - x_{i-1})^3$$

Для построения кубического сплайна необходимо построить n многочленов третьей степени, т. е. определить 4n неизвестных a_i, b_i, c_i, d_i.

Вычислим коэффициенты: c_1=0, остальные с вычисляются как решения СЛАУ:

$$\begin{cases} 2(h_1+h_2)c_2 + h_2c_3 = 3[\frac{y_2-y_1}{h_2} - \frac{y_1-y_0}{h_1}] \\ h_{i-1}c_{i-1} + 2(h_{i-1}+h_i)c_i + h_ic_{i+1} = 3\left[\frac{y_i-y_{i-1}}{h_i} - \frac{y_{i-1}-y_{i-2}}{h_{i-1}}\right] \text{ где } i = 3,4,\dots,n-1 \\ h_{n-1}c_{n-1} + 2(h_{n-1}+h_n)c_n = 3[\frac{y_n-y_{n-1}}{h_n} - \frac{y_{n-1}-y_{n-2}}{h_{n-1}}] \end{cases}$$

Данную систему решим методом прогонки. Функции вычисления остальных коэффициентов:

$$a_i = y_{i-1};$$

$$b_i = \frac{y_i - y_{i-1}}{h_i} - \frac{1}{3}h_i(c_{i+1} + 2c_i);$$

$$d_i = \frac{c_{i+1} - c_i}{3h_i}, i = 1, 2..n-1;$$

Код:

```
import matplotlib.pyplot as plt
xi = [-0.4, -0.1, 0.2, 0.5, 0.8]
yi = [1.9823, 1.6710, 1.3694, 1.0472, 0.6435]
point_x = 0.1
#Метод прогонки
def race_method(A, b):
    P = [-item[2] for item in A]
    Q = [item for item in b]
    P[0] /= A[0][1]
    Q[0] /= A[0][1]
    for i in range(1, len(b)):
        z = (A[i][1] + A[i][0] * P[i-1])
        P[i] /= z
        Q[i] -= A[i][0] * Q[i-1]
        Q[i] /= z
    x = [item for item in Q]
    for i in range(len(x) - 2, -1, -1):
        x[i] += P[i] * x[i + 1]
    return x

def h_coeff(x):
    h = [x[i] - x[i - 1] for i in range(1, len(x))]
    return h

def c_coeff(h, f):
    A = [[h[i - 1], 2.0 * (h[i - 1] + h[i]), h[i]] for i in range(1, len(h))]
    A[0][0] = A[-1][2] = 0.0
    B = [3.0 * ((f[i + 1] - f[i]) / h[i] - (f[i] - f[i - 1]) / h[i - 1]) for i in range(1,
len(h))]
    return [0.0] + race_method(A, B)

def a_coeff(f):
    return f[:len(f) - 1]

def b_coeff(f, h, c):
    b = [(f[i] - f[i - 1]) / h[i - 1] - (1.0/3.0) * h[i - 1] * (c[i] + 2.0 * c[i - 1]) for i in
range(1, len(h))]
    b.append((f[-1] - f[-2]) / h[-1] - (2.0/3.0) * h[-1] * c[-1])
    return b

def d_coeff(h, c):
    d = [(c[i + 1] - c[i]) / (3.0 * h[i]) for i in range(len(h) - 1)]
    d.append(-c[-1] / (3.0 * h[-1]))
```

```
    return d

def find_position(x, arr):
    for i in range(len(arr) - 1):
        if (arr[i] <= x and x <= arr[i + 1]):
            return i

def interpolation(X, Y):
    h = h_coeff(X)
    c = c_coeff(h, Y)
    a = a_coeff(Y)
    b = b_coeff(Y, h, c)
    d = d_coeff(h, c)
    def interpol(x):
        pos = find_position(x, X)
        if pos < 0:
            return b[0]*x + a[0] - b[0]*X[0]
        elif pos == len(X) - 1:
            return Y[-1] + (b[-1] + 2.0*c[-1]*h[-1] + 3.0*d[-1]*h[-1]*h[-1])*(x - X[-1])
        return a[pos] + b[pos]*(x - X[pos]) + c[pos]*((x - X[pos])**2) + d[pos] * ((x -
X[pos])**3)
    return interpol

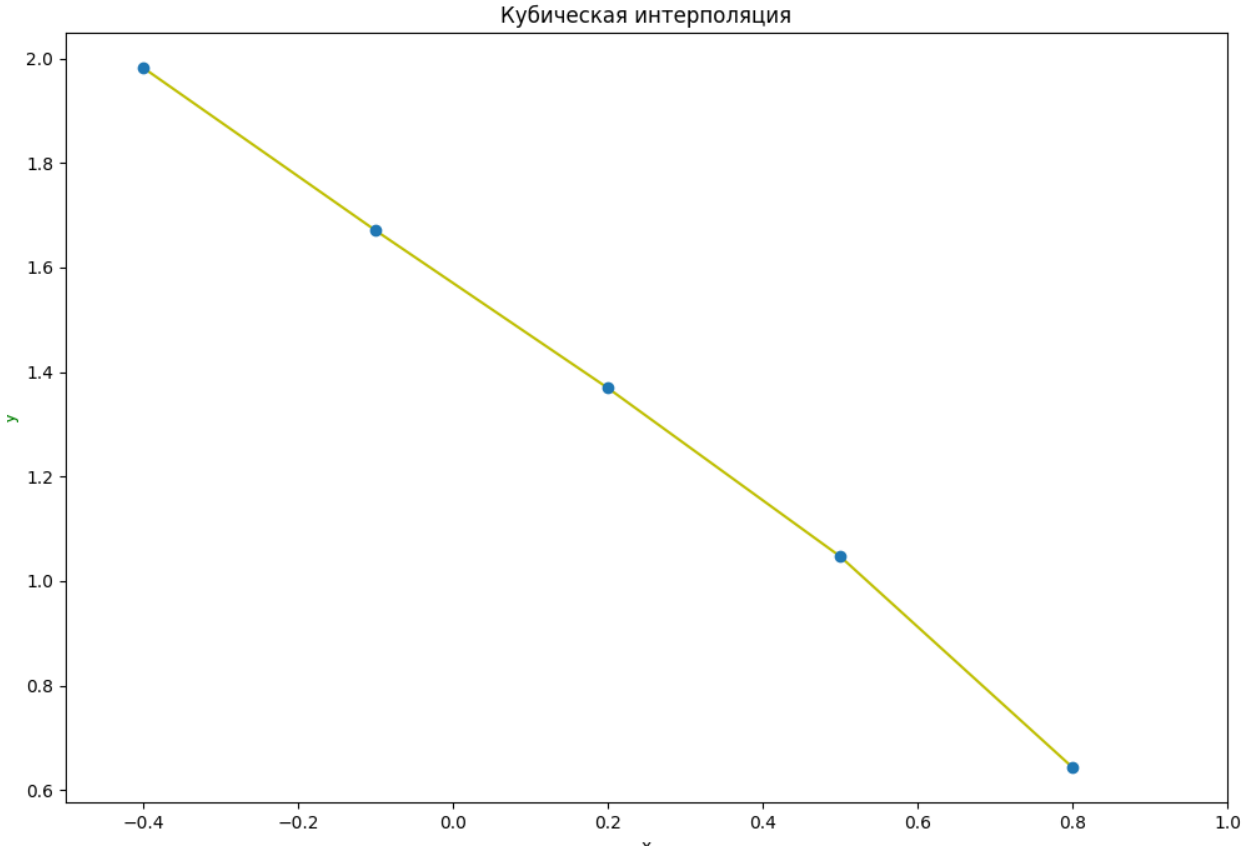
Cubic = interpolation(xi, yi)
print("Значение интерполяции:", Cubic(point_x))

x = xi
y = list(map(Cubic, x))

fig = plt.figure(figsize=(12, 8))
ax1 = fig.add_subplot(111)
line1, = ax1.plot(x, y, 'y', label="f(x)")
ax1.set_xlabel('x')
ax1.set_ylabel('y', color='g')
ax1.plot(list(xi), list(yi), 'o')
plt.title('Кубическая интерполяция')
plt.xlim(-0.5, 1)
plt.show()
```

Результаты выполнения программы:

Значение интерполяции: 1.4694391534391535



3.3. Решение нормальной системы МНК

Дано:

i	0	1	2	3	4	5
x_i	-0.7	-0.4	-0.1	0.2	0.5	0.8
y_i	2.3462	1.9823	1.671	1.3694	1.0472	0.6435

Задача заключается в нахождении коэффициентов линейной зависимости, при которых функция двух переменных a и b формула принимает наименьшее значение. То есть, при данных a и b сумма квадратов отклонений экспериментальных данных от найденной прямой будет наименьшей. В этом вся суть метода наименьших квадратов.

МНК позволяет построить многочлен степени n вида:

$$F_n(x) = \sum_{i=0}^{n} a_i x^i$$

```
def MNK_func(X, Y, n):
    a = count_a(X, Y, n)
    #print("Коэффицеты a приближающего многочлена:", a)
    return lambda x: sum([a[i] * x**i for i in range(n + 1)])
```

Коэффициенты a_i находятся решением системы из n + 1 уравнения:

$$\begin{cases} \sum_{i=0}^{n} a_i \sum_{j=0}^{N} x_j^i = \sum_{j=0}^{N} y_j \\ \dots \\ \sum_{i=0}^{n} a_i \sum_{j=0}^{N} x_j^{i+k} = \sum_{j=0}^{N} y_j x_j^k \\ \dots \\ \sum_{i=0}^{n} a_i \sum_{j=0}^{N} x_j^{i+n} = \sum_{j=0}^{N} y_j x_j^n \end{cases}$$

Данную систему можно решить с помощью метода LU-разложения.

```
def count_a(X, Y, n):
    A = []
    b = []
    N = len(X)
    for k in range(n + 1):
        A.append([sum(map(lambda x: x**(i + k), X)) for i in range(n + 1)])
        b.append(sum(map(lambda x: x[0] * x[1]**k, zip(Y, X))))
    return LU.LU_solution(A, b)
```

Функция вычисления суммы квадратов ошибок вычислений $E = \sum_{j=0}^{N} (F_n(x_j) - y_i)^2$ приближенной функции F_n степени n в N заданных точках (x_j, y_j):

```
def error_value(F, X, Y):
    return reduce(lambda x, y: x + y, map(lambda v: (F(v[0]) - v[1])**2, zip(X, Y)))
```

Вспомогательный код:

```
x = [-0.7, -0.4, -0.1, 0.2, 0.5, 0.8]

y = [2.3462, 1.9823, 1.671, 1.3694, 1.0472, 0.6435]

F1 = MNK_func(x, y, 1)
F2 = MNK_func(x, y, 2)

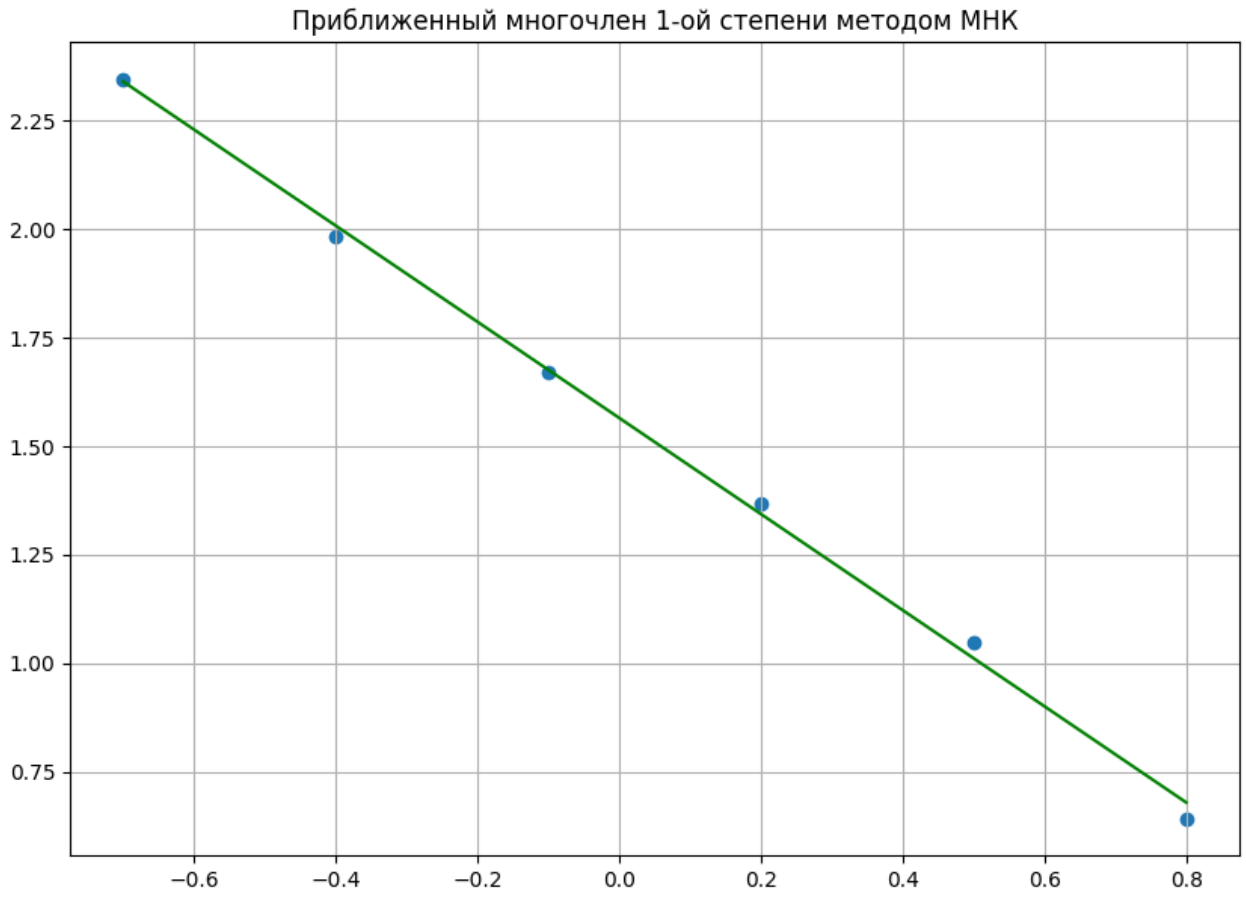
print("Значение квадрата ошибки n=1:", error_value(F1, x, y))
print("Значение квадрата ошибки n=2:", error_value(F2, x, y))

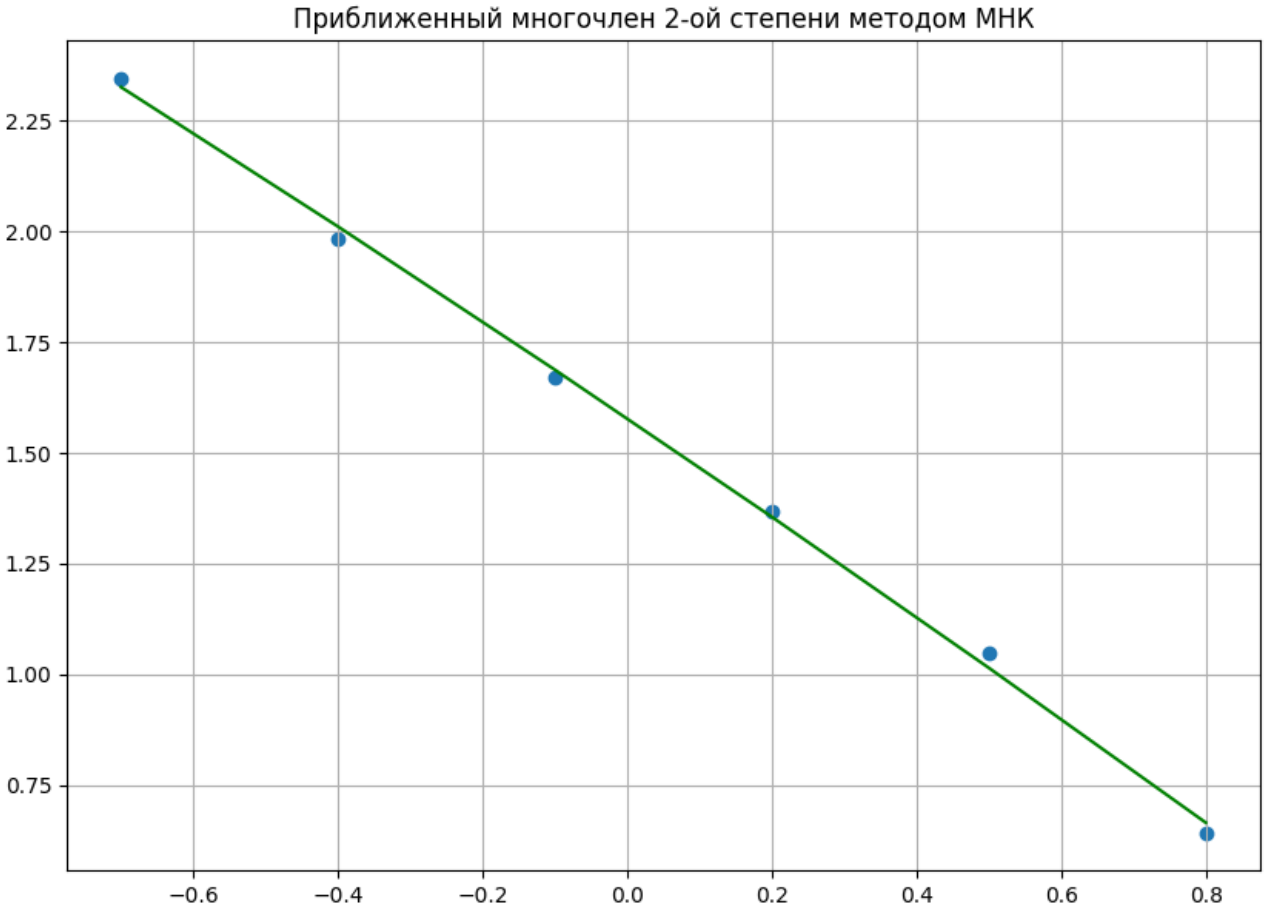
y1 = list(map(F1, x))
plt.figure(figsize=(10,7))
plt.scatter(x, y)
plt.title('Приближенный многочлен 1-ой степени методом МНК')
plt.plot(x, y1, color='g')
plt.grid()

y2 = list(map(F2, x))
plt.figure(figsize=(10,7))
plt.scatter(x, y)
plt.title('Приближенный многочлен 2-ой степени методом МНК')
plt.plot(x, y2, color='g')
plt.grid()
plt.show()
```

Результаты выполнения программы:

Значение квадрата ошибки n=1: 0.003940351047619047
Значение квадрата ошибки n=2: 0.0032396991428571185


Приближенный многочлен 1-ой степени методом МНК
2.25
2.00
1.75
1.50
1.25
1.00
0.75
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8


Приближенный многочлен 2-ой степени методом МНК
2.25
2.00
1.75
1.50
1.25
1.00
0.75
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8

3.4. Первая и вторая производная

Дано: $X^* = 1.0$

i	0	1	2	3	4
x_i	-1.0	0.0	1.0	2.0	3.0
y_i	2.3562	1.5708	0.7854	0.46365	0.32175

Таблично-заданные функции могут быть аппроксимированы отрезками прямой или интерполяционным многочленом второй степени. В случае аппроксимации отрезками прямой имеем:

$$y(x) \approx \varphi(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i), x \in [x_i, x_{i+1}]$$

Тогда первая производная имеет вид:

$$y'(x) \approx \varphi'(x) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} = const, x \in [x_i, x_{i+1}]$$

В таком случае производная рассчитывается с первым порядком точности в крайних точках интервала и со вторым порядком точности в средних точках интервала.

При аппроксимации интерполяционным многочленом второй степени имеем:

$$y(x) \approx \varphi(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) + \frac{\frac{y_{i+2} - y_{i+1}}{x_{i+2} - x_{i+1}} - \frac{y_{i+1} - y_i}{x_{i+1} - x_i}}{x_{i+2} - x_i}(x - x_i)(x - x_{i+1})$$

где $x \in [x_i, x_{i+1}]$

$$y'(x) \approx \varphi'(x) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} + \frac{\frac{y_{i+2} - y_{i+1}}{x_{i+2} - x_{i+1}} - \frac{y_{i+1} - y_i}{x_{i+1} - x_i}}{x_{i+2} - x_i}(2x - x_i - x_{i+1}), x \in [x_i, x_{i+1}]$$

Данная формула обеспечивает второй порядок точности. Также мы можем вычислить вторую производную:

$$y''(x) \approx \varphi''(x) = 2\frac{\frac{y_{i+2} - y_{i+1}}{x_{i+2} - x_{i+1}} - \frac{y_{i+1} - y_i}{x_{i+1} - x_i}}{x_{i+2} - x_i}, x \in [x_i, x_{i+1}]$$

Для известного шага h можно использовать формулы:

$$y'(x) \approx= \frac{-3y_i + 4y_{i+1} - y_{i+2}}{2\mathrm{h}}$$

$$y''(x) \approx= \frac{y_i - 2y_{i+1} + y_{i+2}}{\mathrm{h}^2}$$

Код:

```
x0 = 1
xi = [-1, 0, 1, 2, 3]
yi = [2.3562, 1.5708, 0.7854, 0.46365, 0.32175]
def Derivative1(x, y, x0, i):
    num1 = (y[i + 1] - y[i]) / (x[i + 1] - x[i])
    num2 = (y[i + 2] - y[i + 1]) / (x[i + 2] - x[i + 1]) - num1
    num2 = num2 / (x[i + 2] - x[i])
    result = num1 + num2 * (2 * x0 - x[i] - x[i + 1])
    print(result)
```

```
def Derivative2(x, y, i):
    num1 = (y[i + 2] - y[i + 1]) / (x[i + 2] - x[i + 1])
    num2 = (y[i + 1] - y[i]) / (x[i + 1] - x[i])
    result = 2 * (num1 - num2) / (x[i + 2] - x[i])
    print(result)

def Derivative22(y, i):
    result = y[i] - 2 * y[i+1] + y[i+2]
    print('Второй способ:', result)
```

Результаты выполнения программы:

Задания:

1. Первая производная

-0.5535749999999999

2. Вторая производная

Первый способ: 0.46365

Второй способ: 0.46365

3.5. Методы прямоугольников, трапеций, Симпсона

Дано:

$$y = \frac{x}{x^2 + 9}, \qquad X_0 = 0, \quad X_k = 2, \quad h_1 = 0.5, \quad h_2 = 0.25;$$

Метод прямоугольников

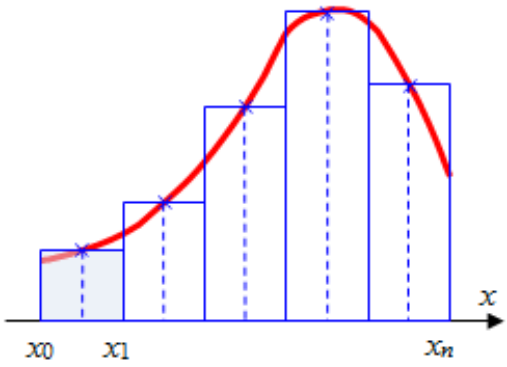
Метод прямоугольников имеет три модификации:

1) метод левых прямоугольников;
2) метод правых прямоугольников;
3) метод средних прямоугольников.

Методы получили свое название из-за того, что высоты прямоугольников на промежуточных отрезках равны значениям функций в правых/левых/срединных частях данных отрезков.
Формула прямоугольников (средних) численного интегрирования:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N} h_i f\left(\frac{x_{i-1} + x_i}{2}\right)$$

Физическая интерпретация:



Метод прямоугольников является методом первого порядка точности.

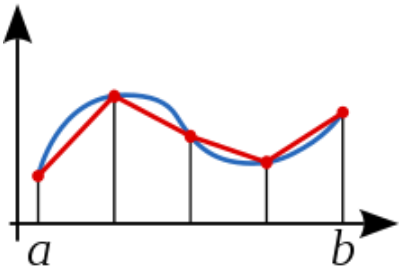
```
def Rectangle(f, h, x0, xk):
    x = np.arange(x0, xk, h)
    return h * sum(f((x[i-1] + x[i]) / 2) for i in range(1, len(x)))
```

Метод трапеций

Формула трапеций численного интегрирования:

$$\int_a^b f(x)dx \approx \frac{h}{2}\left(y_0 + y_n + 2\sum_{i=1}^{n-1} y_i\right)$$

Физическая интерпретация:




```
def Trapezoidal(f, h, x0, xk):
    x = np.arange(x0, xk, h)
    return h * ((f(x[0]) + f(x[len(x) - 1])) / 2 + sum(f(x[i]) for i in range(1, len(x) - 1)))
```

Метод трапеций является методом второго порядка точности.

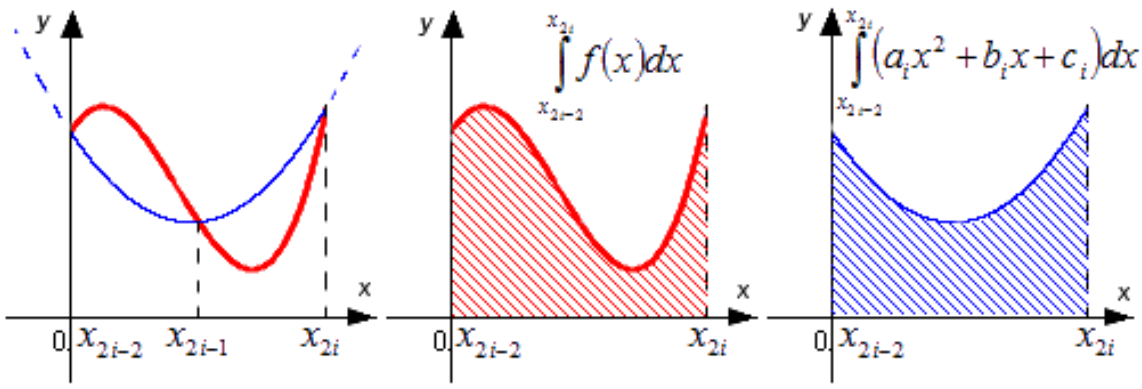
Метод Симпсона

Суть метода заключается в приближении подынтегральной функции на отрезке [a,b] интерполяционным многочленом второй степени, то есть приближение графика функции на отрезке параболой.

Формула Симпсона численного интегрирования:

$$\int_a^b f(x)dx \approx \frac{h}{3}\left(y_0 + y_n + 4\sum_{i=1}^{m} y_{2i-1} + 2\sum_{i=1}^{m-1} y_{2i}\right)$$

Физическая интерпретация:



Метод Симпсона является методом четвертого порядка точности.

```
def Simpson(f, h, x0, xk):
    x = np.arange(x0, xk, h)
    return (h / 3) * (f(x[0]) + f(x[len(x) - 1]) + sum(4 * f(x[i]) for i in range(1, len(x)-1,
2)) + sum(2 * f(x[i]) for i in range(2, len(x)-1, 2)))
```

Метод Рунге-Ромберга

В случае приближенного интегрирования F_1 и F_2 с постоянными шагами h_1 и h_2 и порядком точности p, где $h_1 < h_2$ формула приближенного вычисления порядка точности p+1:

$$F_p = F_1 + \frac{F_1 - F_2}{\left(\frac{h_2}{h_1}\right)^p - 1}$$

```
def Runge_Romberg(F1, F2, h1, h2, p):
    if h1 < h2:
        return F1 + (F1 - F2) / ((h2 / h1) ** p - 1)

    return F2 + (F2 - F1) / ((h1 / h2) ** p - 1)
```

Вспомогательный код:

```
def f(x):
    return x / (x ** 2 + 9)

x0 = 0
xk = 2
h1 = 0.5
h2 = 0.25

p1 = Rectangle(f, h1, x0, xk)
t1 = Trapezoidal(f, h1, x0, xk)
s1 = Simpson(f, h1, x0, xk)

p2 = Rectangle(f, h2, x0, xk)
t2 = Trapezoidal(f, h2, x0, xk)
s2 = Simpson(f, h2, x0, xk)

p = 2   #второй порядок
rp = Runge_Romberg(p1, p2, h1, h2, p)
rt = Runge_Romberg(t1, t2, h1, h2, p)
rs = Runge_Romberg(s1, s2, h1, h2, p)

print('h1 = ', h1)
print("Прямоугольник: ", p1)
print("Трапеция: ", t1)
print("Симпсон: ", s1)
```

```
print('h2 = ', h2)
print("Прямоугольник: ", p2)
print("Трапеция: ", t2)
print("Симпсон: ", s2)

print('\nТочное значение по Рунге-Ромбергу для прямоугольника: {0};\nПогрешность: {1} и
{2}'.format(rp, abs(rp - p1), abs(rp - p2)))
print('\nТочное значение по Рунге-Ромбергу для трапеции: {0};\nПогрешность: {1} и
{2}'.format(rt, abs(rt - t1), abs(rt - t2)))
print('\nТочное значение по Рунге-Ромбергу для Симпсона: {0};\nПогрешность: {1} и
{2}'.format(rs, abs(rs - s1), abs(rs - s2)))
```

Результаты выполнения программы:

h1 = 0.5

Прямоугольник: 0.11218038735592176

Трапеция: 0.11036036036036037

Симпсон: 0.09159159159159158

h2 = 0.25

Прямоугольник: 0.14662198405387739

Трапеция: 0.14607175555071447

Симпсон: 0.1347746328191169

Точное значение по Рунге-Ромбергу для прямоугольника: 0.15810251628652927;

Погрешность: 0.045922128930607514 и 0.011480532232651885

Точное значение по Рунге-Ромбергу для трапеции: 0.15797555394749918;

Погрешность: 0.04761519358713881 и 0.01190379839678471

Точное значение по Рунге-Ромбергу для Симпсона: 0.14916897989495867;

Погрешность: 0.05757738830336709 и 0.014394347075841779

Лабораторная работа 4

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

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.

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.

4.1. Методы Эйлера, Рунге-Кутты и Адамса 4-го порядка

Дано:

№№	Задача Коши	Точное решение
9	$y''-(\frac{1}{x^{1/2}})y'+(\frac{1}{4x^2})(x+x^{1/2}-8)y=0$ $y(1)=2e,$ $y'(1)=2e,$ $x\in[1,2], h=0.1$	$y=\left(x^2+\frac{1}{x}\right)e^{x^{1/2}}$

Исходную систему второго порядка свожу к системе первого порядка заменой: $z = y'$

$$\begin{cases} z = y' \\ z' = \frac{1}{\sqrt{x}}\ z - \frac{1}{4x^2}(x-\sqrt{x}-8)y \\ y(1) = 2e \\ y'(1) = 2e \end{cases}$$

Погрешность буду считать

$$\sqrt{\sum_{i=1}^{n}(y_i - f(x_i))}$$

```
def Accuracy(x, ans, y):
    norm = 0
    for i in range(len(x)):
        norm += (ans[i] - y(x[i]))**2
    return norm**0.5
```

А также методом Рунге-Ромберга-Ридчардсона:

```
def Runge_Romberg(y1, y2, p):
    norm = 0
    for i in range(len(y2)):
```

```
        norm += (y1[i*2] - y2[i])**2
    return (norm**0.5) / (2**p - 1)
```

Метод Эйлера

Явный метод Эйлера:

$$\begin{cases} y_k = y_{k-1} + hz_{k-1} \\ z_k = z_{k-1} + hf(x_{k-1}, y_{k-1}, z_{k-1}) \\ x_k = x_{k-1} + h \end{cases}$$

```
def Euler(x0, y0, z0, x, h):
    m = int((x - x0)/h)
    result = []
    result.append(y0)
    result2 = [z0]
    for _ in range(1, m+1):
        z0 += h*g(x0, y0, z0)
        y0 += h*g1(x0, y0, z0)
        x0 += h
        result.append(y0)
        result2.append(z0)
    return [result, result2]
```

Решение задачи Коши:

```
    [ans, dans] = Euler(a, y1, z1, b, h)
    [ans_, dans_] = Euler(a, y1, z1, b, h/2)
    print('Точное решение в узлах сетки:\n')
    for i in x1:
        print(G(i))
    print('\nРешение методом Эйлера в узлах сетки:')
    for i in range(len(x1)):
        print(ans[i])
    print('\nПогрешность решения (относительно точного решения):', Accuracy(x1, ans, G))
    print('\nПогрешность решения (Рунге-Ромберг):', Runge_Romberg(ans_, ans, 2))
```

Результаты выполнения программы:

Точное решение в узлах сетки:

5.43656365691809

6.048413634095278

6.798434045243182

7.690794011893978

8.731160370607025

9.926284939128585

11.283746789336352

12.811786234461984

14.519195159647559

16.41524310741032

18.509626704523193

Решение методом Эйлера в узлах сетки:

5.43656365691809

6.116134114032851

6.934438472346004

7.896120340092422

9.007089659904448

10.27418447797158

11.704958203993561

13.307541791741219

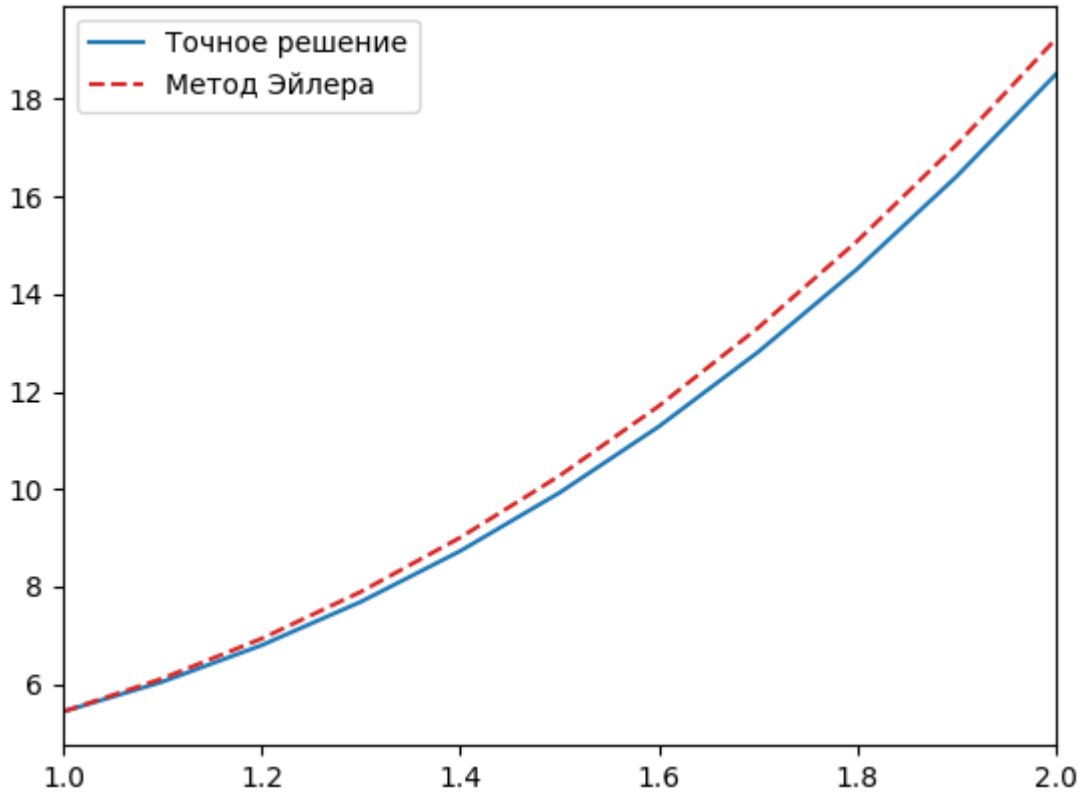
15.090552291082133

17.063031012324792

19.23440112535975

Погрешность решения (относительно точного решения): 1.3989217138102168

Погрешность решения (Рунге-Ромберг): 0.2317057381276317



Метод Рунге-Кутты

Метод Рунге-Кутты 4-го порядка:

$$\begin{cases} \Delta y_{k-1} = \dfrac{K_1^{k-1} + 2K_2^{k-1} + 2K_3^{k-1} + K_4^{k-1}}{6} \\ \Delta z_{k-1} = \dfrac{L_1^{k-1} + 2L_2^{k-1} + 2L_3^{k-1} + L_4^{k-1}}{6} \\ K_1^{k-1} = hz_{k-1} \\ L_1^{k-1} = hf(x_{k-1}, y_{k-1}, z_{k-1}) \\ K_2^{k-1} = h\left(z_{k-1} + \dfrac{L_1^{k-1}}{2}\right) \\ L_2^{k-1} = hf(x_{k-1} + \dfrac{h}{2}, y_{k-1} + \dfrac{K_1^{k-1}}{2}, z_{k-1} + \dfrac{L_1^{k-1}}{2}) \\ K_3^{k-1} = h\left(z_{k-1} + \dfrac{L_2^{k-1}}{2}\right) \\ L_3^{k-1} = hf(x_{k-1} + \dfrac{h}{2}, y_{k-1} + \dfrac{K_2^{k-1}}{2}, z_{k-1} + \dfrac{L_2^{k-1}}{2}) \\ K_4^{k-1} = h\left(z_{k-1} + \dfrac{L_3^{k-1}}{2}\right) \\ L_4^{k-1} = hf(x_{k-1} + \dfrac{h}{2}, y_{k-1} + \dfrac{K_3^{k-1}}{2}, z_{k-1} + \dfrac{L_3^{k-1}}{2}) \end{cases}$$

В случае метода Рунге-Кутты четвертого порядка точности следует на каждом шаге h рассчитывать параметр

$$\theta^k = \left|\frac{K_2^k - K_3^k}{K_1^k - K_2^k}\right|$$

Если величина θ^k порядка нескольких сотых единицы, то расчет продолжается с тем же шагом, если θ^k больше одной десятой, то шаг следует уменьшить, если же θ^k меньше одной сотой, то шаг можно увеличить.

```
def Runge_Kutta(x0, y0, z0, x, h):
    m = int((x - x0)/h)
    result = []
    result.append(y0)
    result2 = [z0]
    for _ in range(1, m+1):
        k1 = h * g(x0, y0, z0)
        l1 = h * g1(x0, y0, z0)

        k2 = h * g(x0 + h/2, y0 + l1/2, z0 + k1/2)
        l2 = h * g1(x0 + h/2, y0 + l1/2, z0 + k1/2)

        k3 = h * g(x0 + h/2, y0 + l2/2, z0 + k2/2)
        l3 = h * g1(x0 + h/2, y0 + l2/2, z0 + k2/2)

        k4 = h * g(x0 + h, y0 + l3, z0 + k3)
        l4 = h * g1(x0 + h, y0 + l3, z0 + k3)

        z0 += (k1 + 2*k2 + 2*k3 + k4)/6
        y0 += (l1 + 2*l2 + 2*l3 + l4)/6

        if abs(k1-k2) > 0:
            teta = abs((k2 - k3) / (k1 - k2))
        if teta > 1/10: h = h - 0.00001
        if teta < 1/100: h = h + 0.00001
        x0 += h
```

```
        result.append(y0)
        result2.append(z0)
    return [result, result2]
```

Решение задачи Коши:

```
[ans, dans] = Runge_Kutta(a, y1, z1, b, h)
[ans_, dans_] = Runge_Kutta(a, y1, z1, b, h/2)
print('Точное решение в узлах сетки:\n')
for i in x1:
    print(G(i))
print('\nРешение методом Эйлера в узлах сетки:')
for i in range(len(x1)):
    print(ans[i])
print('\nПогрешность решения (относительно точного решения):', Accuracy(x1, ans, G))
print('\nПогрешность решения (Рунге-Ромберг):', Runge_Romberg(ans_, ans, 2))
```

Результаты выполнения программы:

Точное решение в узлах сетки:

5.43656365691809

6.048413634095278

6.798434045243182

7.690794011893978

8.731160370607025

9.926284939128585

11.283746789336352

12.811786234461984

14.519195159647559

16.41524310741032

18.509626704523193

Решение методом Рунге-Кутта в узлах сетки:

5.43656365691809

6.048414190831023

6.798353039079775

7.690507864380861

8.730500004851532

9.925030981880015

11.28162423396915

12.808619652986561

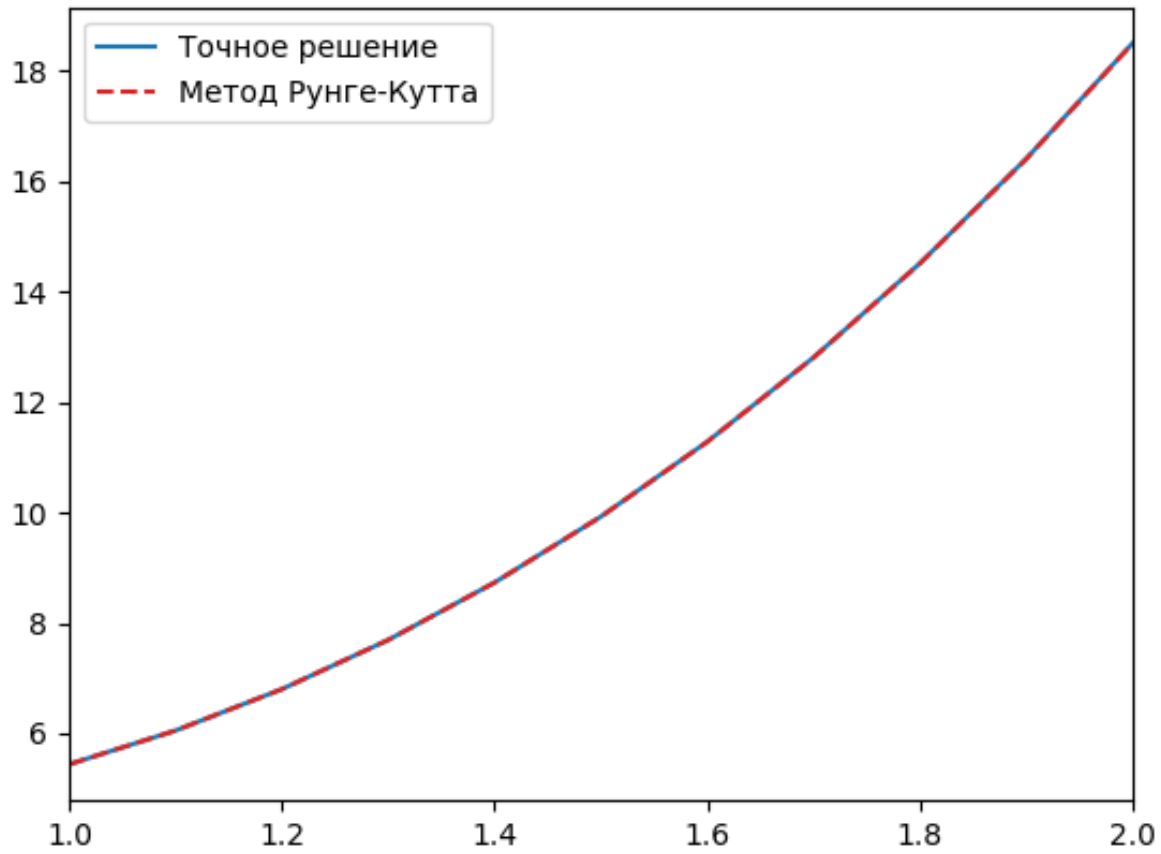
14.514795509115453

16.409406942148777

18.502135465713913

Погрешность решения (относительно точного решения): 0.01123235605782097

Погрешность решения (Рунге-Ромберг): 0.00424845742226109



Метод Адамса

Этап предиктор

По значениям в узлах $x_{k-3}, x_{k-2}, x_{k-1}, x_k$ рассчитывается “предварительное” значение решения в узле x_{k+1}.

$$\hat{y}_{k+1} = y_k + \frac{h}{24}(55f_k - 59f_{k-1} + 37f_{k-2} - 9f_{k-3})$$

С помощью полученного значения $\hat{y}_{k+1}$ рассчитывается “предварительное” значение функции $f_{k+1} = f(x_{k+1}, \hat{y}_{k+1})$ в новой точке.

Этап корректор

На корректирующем этапе по методу Адамса 4-го порядка по значениям в узлах $x_{k-2}, x_{k-1}, x_k, x_{k+1}$ рассчитывается “окончательное” значение решения в узле x_{k+1}.

$$y_{k+1} = y_k + \frac{h}{24}(9f_{k+1} + 19f_k - 5f_{k-1} + f_{k-2})$$

```
def Adams(x, y, z, h):
    n = len(x)
    x = x[:4]
    x1 = x[:5]
    y = y[:4]
    y1 = y[:5]
```

```
    z = z[:4]
    z1 = z[:5]
    #этап предиктор
    for i in range(3, n - 1):
        z_i = z[i] + h * (55 * g(x[i], y[i], z[i]) -
                          59 * g(x[i - 1], y[i - 1], z[i - 1]) +
                          37 * g(x[i - 2], y[i - 2], z[i - 2]) -
                           9 * g(x[i - 3], y[i - 3], z[i - 3])) / 24
        z1.append(z_i)
        y_i = y[i] + h * (55 * g1(x[i], y[i], z[i]) -
                          59 * g1(x[i - 1], y[i - 1], z[i - 1]) +
                          37 * g1(x[i - 2], y[i - 2], z[i - 2]) -
                           9 * g1(x[i - 3], y[i - 3], z[i - 3])) / 24
        y1.append(y_i)
        x1.append(x[i] + h)
        #этап корректор
        z_i = z[i] + h / 24 * (9 * g(x1[i+1], y1[i+1], z1[i+1]) +
                               19 * g(x1[i], y1[i], z[i]) -
                               5 * g(x1[i - 1], y1[i - 1], z1[i - 1]) +
                               1 * g(x1[i - 2], y1[i - 2], z1[i - 2]))
        z.append(z_i)
        y_i = y[i] + h / 24 * (9 * g1(x1[i+1], y1[i+1], z1[i+1]) +
                               19 * g1(x1[i], y1[i], z1[i]) -
                               5 * g1(x1[i - 1], y1[i - 1], z1[i - 1]) +
                               1 * g1(x1[i - 2], y1[i - 2], z1[i - 2]))
        y.append(y_i)
        x.append(x[i] + h)
    return [y, z]
```

Решение задачи Коши:

```
runge_x, runge_y, runge_z = Runge_Kutta(a, y1, z1, b, h)
runge_x_, runge_y_, runge_z_ = Runge_Kutta(a, y1, z1, b, h/2)
[ans, dans] = Adams(runge_x, runge_y, runge_z, h)
[ans_, dans_] = Adams(runge_x_, runge_y_, runge_z_, h/2)
print('Точное решение в узлах сетки:\n')
for i in x1:
    print(G(i))
print('\nРешение методом Адамса в узлах сетки:')
for i in range(len(x1)):
    print(ans[i])
print('\nПогрешность решения (относительно точного решения):', Accuracy(x1, ans, G))
print('\nПогрешность решения (Рунге-Ромберг):', Runge_Romberg(ans_, ans, 2))
```

Результаты выполнения программы:

Точное решение в узлах сетки:

5.43656365691809

6.048413634095278

6.798434045243182

7.690794011893978

8.731160370607025

9.926284939128585

11.283746789336352

12.811786234461984

14.519195159647559

16.41524310741032

18.509626704523193

Решение методом Адамса в узлах сетки:

5.43656365691809

6.048414190831023

6.798353039079775

7.690507864380861

8.730786552218495

9.925776210320668

11.283169603876466

12.811156749850458

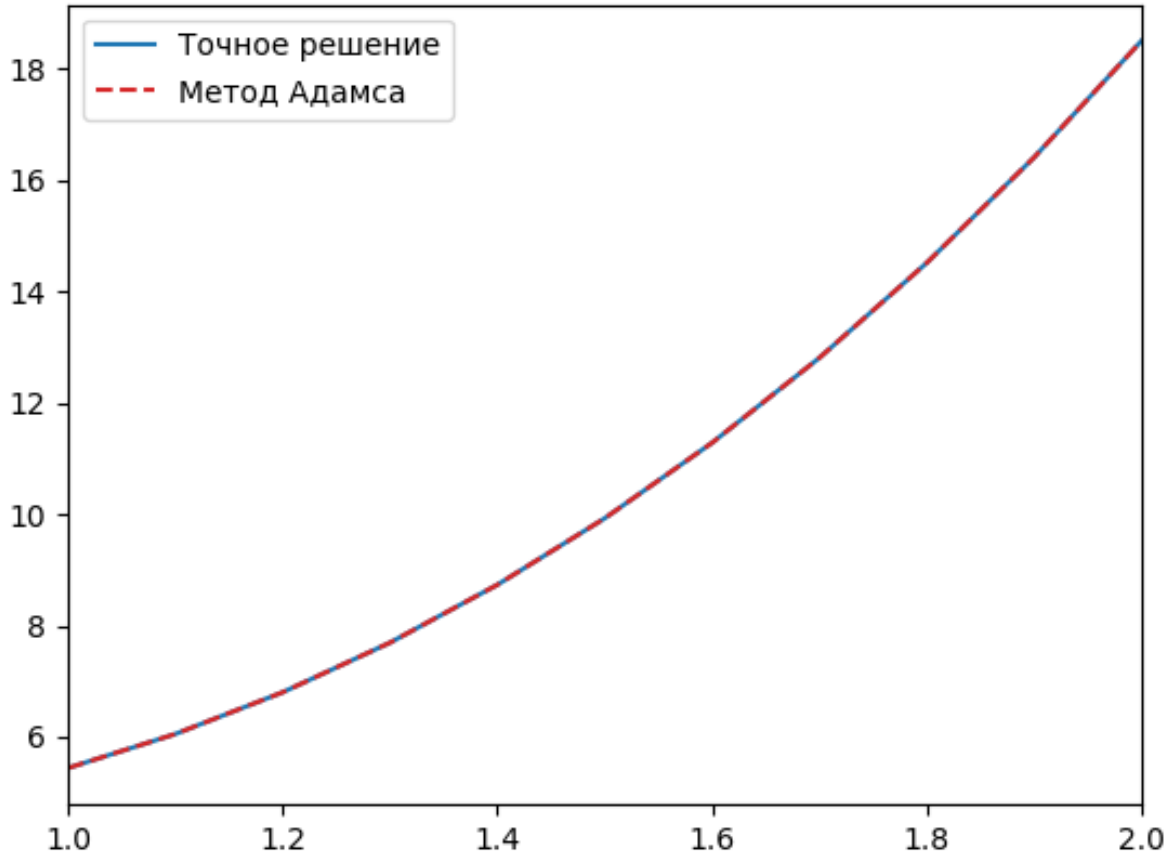
14.518529496608734

16.41455083418037

18.508916332993735

Погрешность решения (относительно точного решения): 0.0016258410262418668

Погрешность решения (Рунге-Ромберг): 0.00020432570071748492



4.2. Метод стрельбы и конечно-разностный метод

Дано:

y″-2(1+(tgx)^2)y=0, y(0)=0, $y(\frac{\pi}{6}) = -\frac{\sqrt{3}}{3}$	y(x)= – tgx

Исходную систему второго порядка свожу к системе первого порядка заменой: $z = y'$

$$\begin{cases} z = y' \\ z' = 2(1 + tg^2 x)y \\ y(0) = 0 \\ y\left(\frac{\pi}{6}\right) = -\frac{\sqrt{3}}{3} \end{cases}$$

Погрешность буду считать

$$\sqrt{\sum_{i=1}^{n}(y_i - f(x_i))}$$

```
def Accuracy(x, ans, y):
    norm = 0
    for i in range(len(x)):
        norm += (ans[i] - y(x[i]))**2
    return norm**0.5
```

А также методом Рунге-Ромберга-Ридчардсона:

```
def Runge_Romberg(y1, y2, p):
    norm = 0
    for i in range(len(y2)):
        norm += (y1[i*2] - y2[i])**2
    return (norm**0.5) / (2**p - 1)
```

Метод стрельбы

Метод стрельбы заключается в многократном решении задачи Коши для приближенного краевой задачи. Начальные условия задачи Коши будут иметь вид:

$$y(a) = y_0$$

$$y'(a) = \eta$$

где η – значение тангенса угла наклона касательной к решению в точке x = a.

Решение задачи Коши будем осуществлять методом Рунге-Кутты 4-го порядка, а значение η будем искать:

$$\eta_{j+2} = \eta_{j+1} - \frac{\eta_{j+1} - \eta_j}{\Phi(\eta_{j+1}) - \Phi(\eta_j)}\Phi(\eta_{j+1})$$

```
def shooting_method(x, y0, y1, h, f = f_xyz, e = 0.00001):
    n0 = 1
    n = 0.8
    y_prev = RK.runge_kutta_method(x, y0, n0, h, f)
    y_i = RK.runge_kutta_method(x, y0, n, h, f)
    Fi_prev = y_prev[-1] - y1 #нахождение корня уравнения Ф
    Fi_i = y_i[-1] - y1
    while abs(Fi_i) > e:
        n0, n = n, next_iter(Fi_prev, Fi_i, n0, n)
        y_prev, y_i = y_i, RK.runge_kutta_method(x, y0, n, h, f)
        Fi_prev, Fi_i = Fi_i, y_i[-1] - y1
```

```
    for i in range (len(x)):
        print('x: ', round(x[i],2), 'y: ', y_i[i])
    return y_i
```

Решение краевой задачи:

```
x = find_node_points(a, b, h)
print("Метод стрельбы")
y = shooting_method(x, y0, y1, h)
p = 4
x_ = find_node_points(a, b, h/2)
print()
y_ = shooting_method(x_, y0, y1, h/2)
print()
print('Погрешность решения относительно точного')
print(accuracy(x, y))
print('Погрешность с помощью метода Рунге-Ромберга-Ричардсона')
print(runge_romberg_richardson(y_, y, p))
print()
```

Результаты выполнения программы:

Метод стрельбы

x: 0.0 y: 0

x: 0.1 y: -0.10603721315103914

x: 0.2 y: -0.21423105667859293

x: 0.3 y: -0.3269172305121615

x: 0.4 y: -0.4468221537709553

x: 0.5 y: -0.5773502691896262

x: 0.0 y: 0

x: 0.05 y: -0.052885716753312134

x: 0.1 y: -0.10603696804478832

x: 0.15 y: -0.1597246480970402

x: 0.2 y: -0.21423060227416343

x: 0.25 y: -0.26985369884421734

x: 0.3 y: -0.3269166666966438

x: 0.35 y: -0.38577404488331757

x: 0.4 y: -0.44682168128243094

x: 0.45 y: -0.5105083520393586

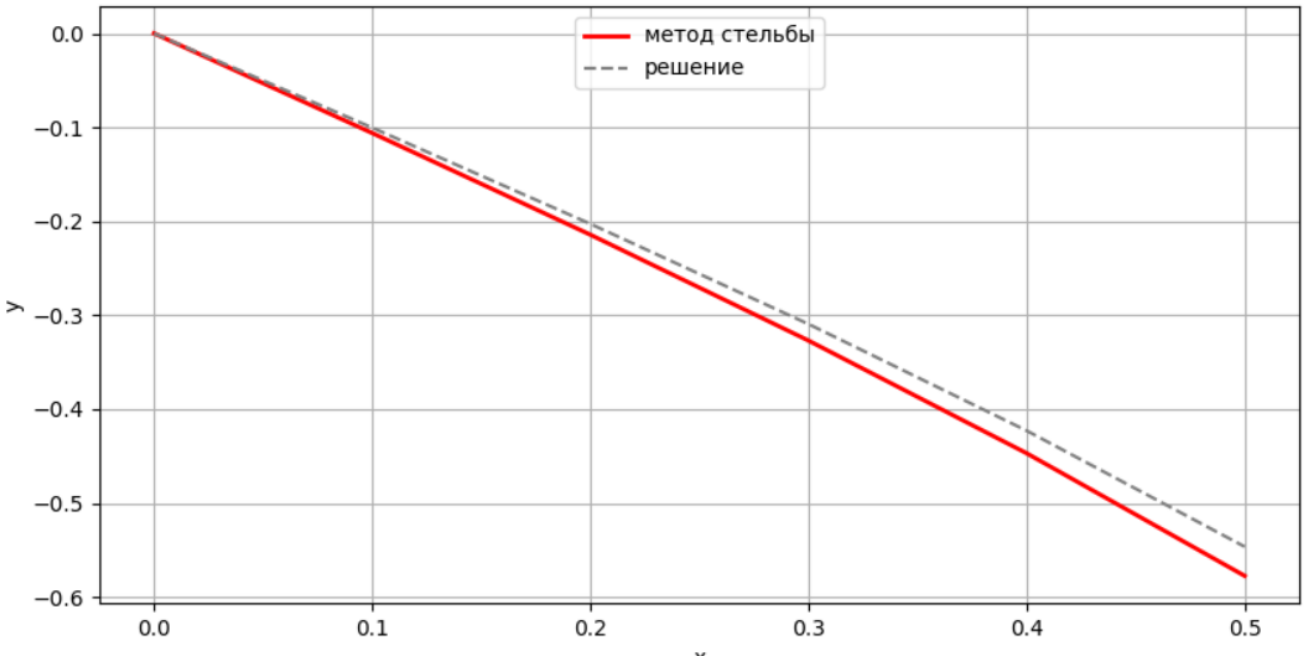
x: 0.5 y: -0.5773502691896274

Погрешность решения относительно точного

0.044896526747798704

Погрешность с помощью метода Рунге-Ромберга-Ричардсона

5.991452442540626e-08



Конечно – разностный метод

Для конечно – разностного метода необходимо представить искомое уравнение в виде:

$$y'' + p(x)y' + q(x)y = f(x)$$

Значение функции определяются решением методом прогонки трехдиагональной системы уравнений вида:

$$\begin{cases} y_0 = y_a \\ \dfrac{y_{k+1} - 2y_k + y_{k-1}}{h^2} + p(x_k)\dfrac{y_{k+1} - y_{k-1}}{2h} + q(x_k)y_k = f(x_k), k = 1, N-1 \\ y_N = y_b \end{cases}$$

```
def find_tridig_A(h, p, q, x):
    A = [[1 - (p(x[i]))/2, (-2 + h*h*q(x[i])), 1 + (p(x[i])*h)/2] for i in range(1, len(x[:-
1]))]
    A[0][0] = 0
    A[-1][-1] = 0
    return A

def find_b(h, p, f, x, y0, y1):
    b = [h*h*f(x[i]) for i in range(1, len(x[:-1]))]
    b[0] -= y0*(1 - p(x[1])*h/2) #1
    b[-1] -= y1*(1 + p(x[-2])*h/2) #последний
    return b

def finite_differences_method(x, y0, y1, h, p = p, q = q, f = f):
    A = find_tridig_A(h, p, q, x)
    b = find_b(h, p, f, x, y0, y1)
    y = [y0] + Progon.tridig_matrix_alg(A, b) + [y1]
    for i in range (len(x)):
        print('x: ', round(x[i],2), 'y: ', y[i])
    return y
```

Решение краевой задачи:

```
print("Конечно-разностный метод")
yd = finite_differences_method(x, y0, y1, h)
print()
```

```
yd_ = finite_differences_method(x_, y0, y1, h/2)
print('Погрешность решения относительно точного')
print(accuracy(x, yd))
p = 2
print('Погрешность с помощью метода Рунге-Ромберга-Ричардсона')
print(runge_romberg_richardson(yd_, yd, p))

x1, y1 = x, yd
```

Результаты выполнения программы:

Конечно-разностный метод

x: 0.0 y: 0

x: 0.1 y: -0.10610325662119513

x: 0.2 y: -0.21434994130301385

x: 0.3 y: -0.3270597834165261

x: 0.4 y: -0.446936741116784

x: 0.5 y: -0.5773502691896257

x: 0.0 y: 0

x: 0.05 y: -0.05289426842749417

x: 0.1 y: -0.10605367047900822

x: 0.15 y: -0.15974867911903726

x: 0.2 y: -0.21426067593479697

x: 0.25 y: -0.2698879974414627

x: 0.3 y: -0.3269527417433728

x: 0.35 y: -0.3858086785200956

x: 0.4 y: -0.4468506956234324

x: 0.45 y: -0.5105263481875222

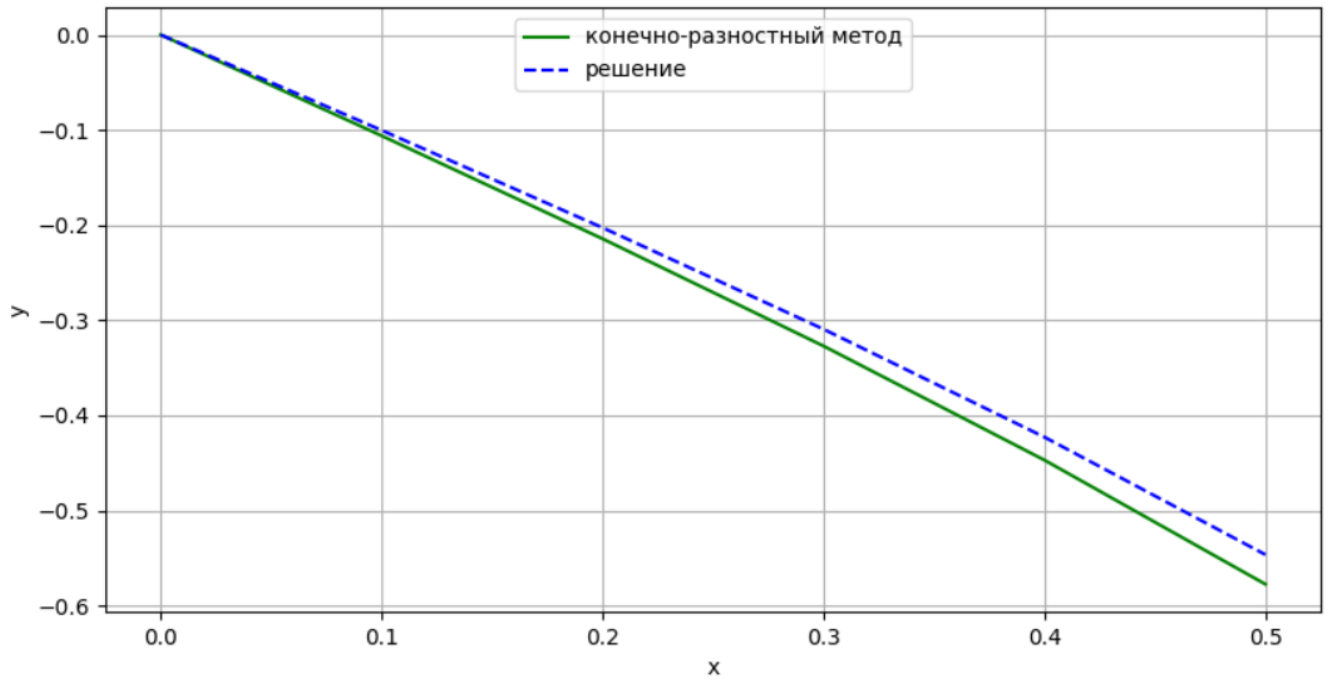
x: 0.5 y: -0.5773502691896257

Погрешность решения относительно точного

0.045052878933607605

Погрешность с помощью метода Рунге-Ромберга-Ричардсона

5.7046606849577875e-05


конечно-разностный метод
решение
0.0
−0.1
−0.2
−0.3
−0.4
−0.5
−0.6
y
0.0
0.1
0.2
0.3
0.4
0.5
x