ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ   
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ   
ВЫСШЕГО ОБРАЗОВАНИЯ  
МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Отчёт по лабораторным работам

по курсу «Численные методы»

III курс, VI семестр  
Вариант 15

Студент: Корнев М. С.

Группа: М8О-312Б-22

Руководитель: Демидова О. Л.

Москва 2025

**Оглавление**

[**1 Лабораторная работа** 3](#_Toc206539125)

[1.1 LU – разложение 3](#_Toc206539126)

[1.2 Метод прогонки 8](#_Toc206539127)

[1.3 Метод простых итераций и метод Зейделя 13](#_Toc206539128)

[1.4 Метод вращений 23](#_Toc206539129)

[1.5 QR - разложение 32](#_Toc206539130)

[**2 Лабораторная работа** 38](#_Toc206539131)

[2.1 Решение нелинейных уравнений 38](#_Toc206539132)

[2.2 Решение систем нелинейных уравнений 45](#_Toc206539133)

[**3 Лабораторная работа** 57](#_Toc206539134)

[3.1 Интерполяция 57](#_Toc206539135)

[3.2 Сплайны 65](#_Toc206539136)

[3.3 Метод наименьших квадратов 69](#_Toc206539137)

[3.4 Численное дифференцирование 74](#_Toc206539138)

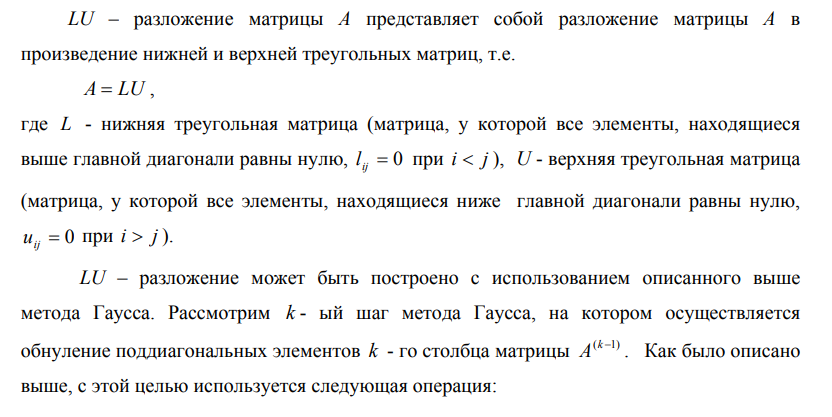
[3.5 Численное интегрирование 78](#_Toc206539139)

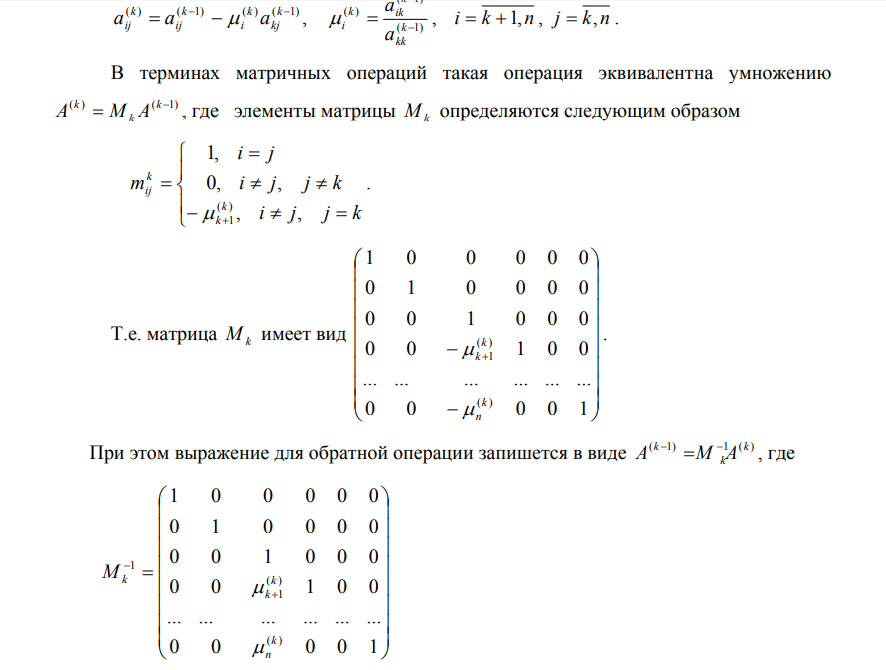
[**4 Лабораторная работа** 83](#_Toc206539140)

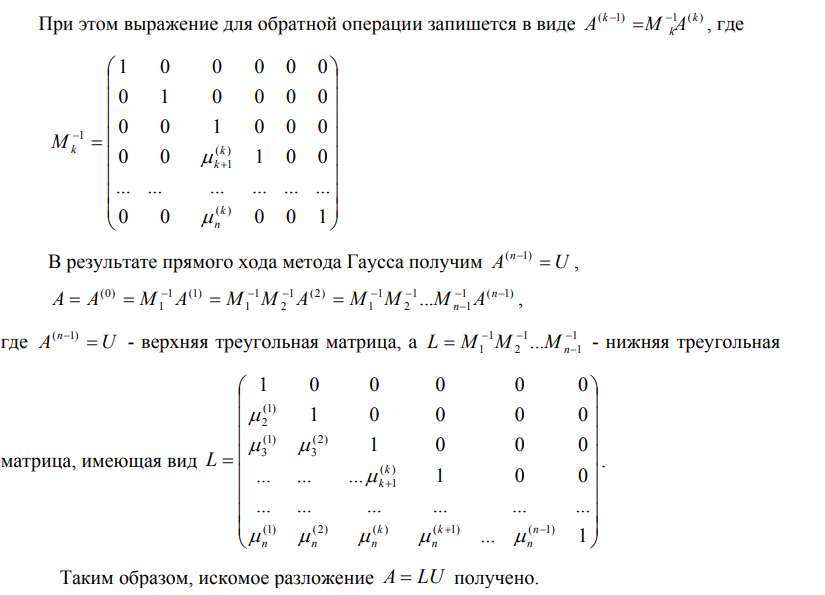
# **1 Лабораторная работа**

## 1.1 LU – разложение

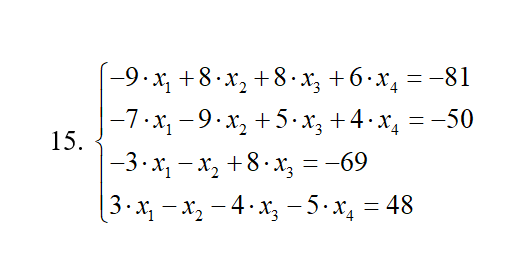
Реализовать алгоритм LU - разложения матриц (с выбором главного элемента) в виде программы. Используя разработанное программное обеспечение, решить систему линейных алгебраических уравнений (СЛАУ). Для матрицы СЛАУ вычислить определитель и обратную матрицу.







Условие:



Код программы:

def determinant(A):

    n = len(A)

    det = 1

    for i in range(n):

        det \*= A[i][i]

    return det

def matrix\_multiply(A, B):

    n = len(A)

    m = len(B[0])

    result = [[0] \* m for \_ in range(n)]

    for i in range(n):

        for j in range(m):

            result[i][j] = sum(A[i][k] \* B[k][j] for k in range(len(B)))

    return result

def lu\_decomposition(A):

    n = len(A)

    for k in range(n):

        for i in range(k + 1, n):

            A[i][k] /= A[k][k]

            for j in range(k + 1, n):

                A[i][j] -= A[i][k] \* A[k][j]

    return A

def solve\_lu(A, b):

    n = len(A)

    y = [0] \* n

    for i in range(n):

        y[i] = b[i] - sum(A[i][j] \* y[j] for j in range(i))

    x = [0] \* n

    for i in range(n - 1, -1, -1):

        x[i] = (y[i] - sum(A[i][j] \* x[j] for j in range(i + 1, n))) / A[i][i]

    return x

def inverse\_matrix(A):

    n = len(A)

    inv = [[0] \* n for \_ in range(n)]

    for i in range(n):

        e = [0] \* n

        e[i] = 1

        inv\_col = solve\_lu(A, e)

        for j in range(n):

            inv[j][i] = inv\_col[j]

    return inv

def check\_solution(A, x, b):

    n = len(A)

    Ax = [sum(A[i][j] \* x[j] for j in range(n)) for i in range(n)]

    return all(abs(Ax[i] - b[i]) < 1e-6 for i in range(n))

A = [

    [-9, 8, 8, 6],

    [-7, -9, 5, 4],

    [-3, -1, 8, 0],

    [3, -1, -4, -5]

]

b = [-81, -50, -69, 48]

A\_lu = lu\_decomposition([row[:] for row in A])

x = solve\_lu(A\_lu, b)

det\_A = determinant(A\_lu)

inv\_A = inverse\_matrix(A\_lu)

is\_correct = check\_solution(A, x, b)

print("Решение СЛАУ (x):", x)

print("\nОпределитель матрицы A:", det\_A)

print("\nОбратная матрица A:")

for row in inv\_A:

    print(row)

print("\nПроверка решения (Ax = b):", is\_correct)

Результат:

Решение СЛАУ (x): [-1.0, -0.0, -9.0, -3.0]

Определитель матрицы A: -2739.0

Обратная матрица A:

[-0.13983205549470606, -0.10514786418400876, 0.0795910916392844, -0.2519167579408543]

[0.05732018985031034, -0.06133625410733845, -0.009127418765972981, 0.019715224534501648]

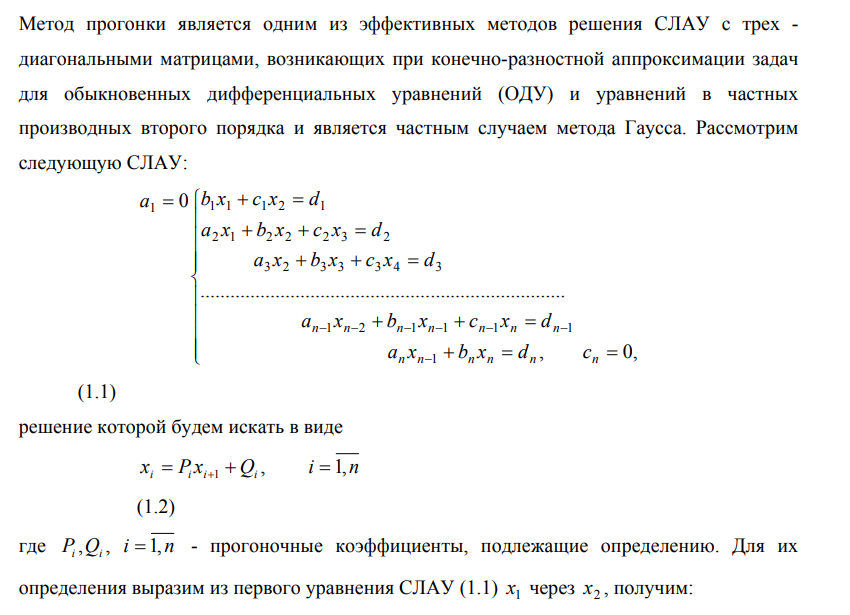
[-0.04527199707922598, -0.04709748083242059, 0.15370573201898502, -0.09200438116100766]

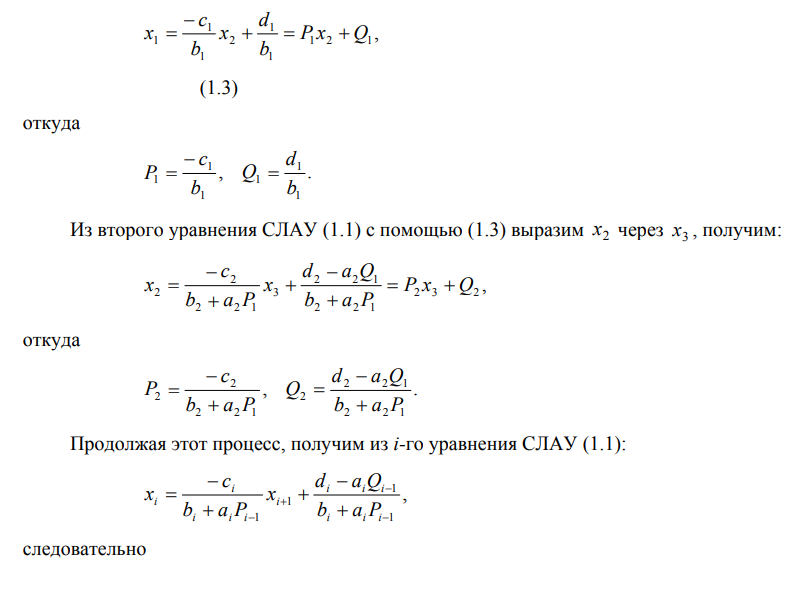
[-0.059145673603504915, -0.01314348302300109, -0.07338444687842278, -0.28148959474260676]

Проверка решения (Ax = b): True

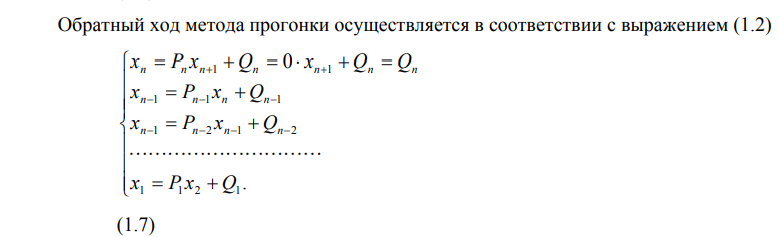
## 1.2 Метод прогонки

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

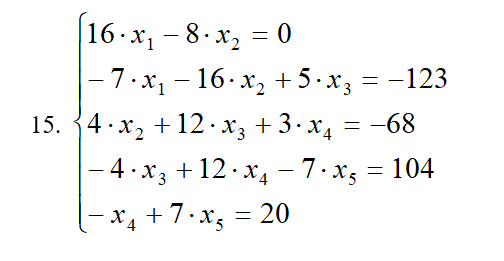








Условие:



Код программы:

def is\_tridiagonal(matrix):

    n = len(matrix)

    for i in range(n):

        for j in range(n):

            if abs(i - j) > 1 and matrix[i][j] != 0:

                return False

    return True

def get\_diagonal(matrix, offset):

    n = len(matrix)

    if offset == 0:  # главная диагональ

        return [matrix[i][i] for i in range(n)]

    elif offset == -1:  # поддиагональ

        return [0] + [matrix[i][i-1] for i in range(1, n)]

    elif offset == 1:  # наддиагональ

        return [matrix[i][i+1] for i in range(n-1)] + [0]

    else:

        raise ValueError("Неподдерживаемый offset")

def run\_through\_algorithm(matrix, d):

    if not is\_tridiagonal(matrix):

        raise ValueError("матрица не является трехдиагональной")

    n = len(matrix)

    a = get\_diagonal(matrix, -1) # поддиагональ

    b = get\_diagonal(matrix, 0)# главная диагональ

    c = get\_diagonal(matrix, 1) # наддиагональ

    d = d.copy()

    if any(b\_i == 0 for b\_i in b):

        raise ValueError("главная диагональ содержит нули")

    # Прямой ход

    p = [0] \* n

    q = [0] \* n

    p[0] = -c[0] / b[0]

    q[0] = d[0] / b[0]

    for i in range(1, n):

        denom = b[i] + a[i] \* p[i-1]

        if denom == 0:

            raise ValueError("деление на ноль")

        p[i] = -c[i] / denom

        q[i] = (d[i] - a[i] \* q[i-1]) / denom

    x = [0] \* n

    x[-1] = q[-1]

    for i in range(n-2, -1, -1):

        x[i] = p[i] \* x[i+1] + q[i]

    return x

def matrix\_multiply(A, B):

    n = len(A)

    m = len(B[0])

    result = [[0] \* m for \_ in range(n)]

    for i in range(n):

        for j in range(m):

            result[i][j] = sum(A[i][k] \* B[k][j] for k in range(len(B)))

    return result

A = [

        [16, -8, 0, 0, 0],

        [-7, -16, 5, 0, 0],

        [0, 4, 12, 3, 0],

        [0, 0, -4, 12, -7],

        [0, 0, 0, -1, 7]

    ]

B = [0, -123, -68, 104, 20]

solution = run\_through\_algorithm(A, B)

print("Рещение СЛАУ методом прогонки:", solution)

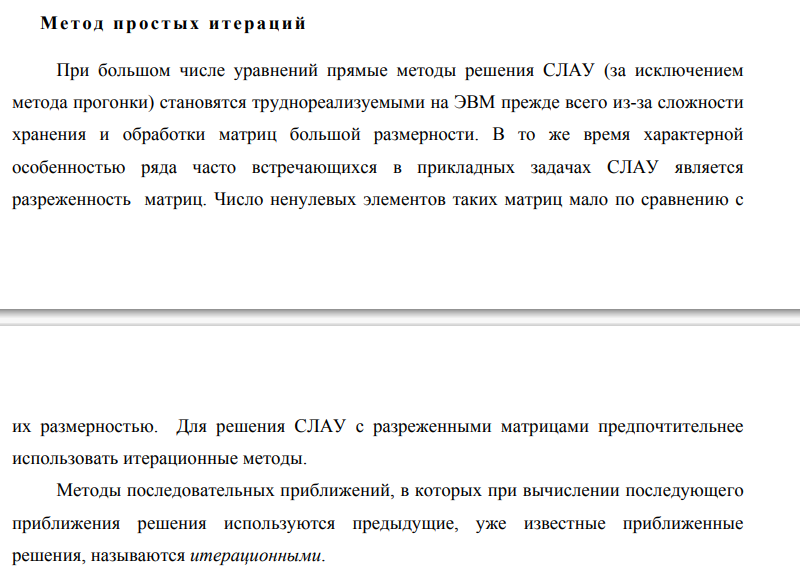
print("\nпроверка:", matrix\_multiply(A, [[x] for x in solution]))

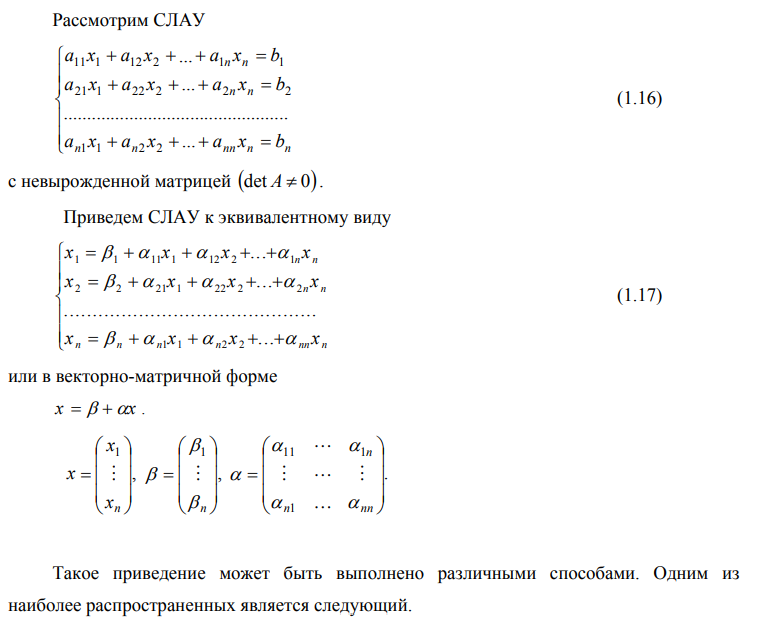
Результат:

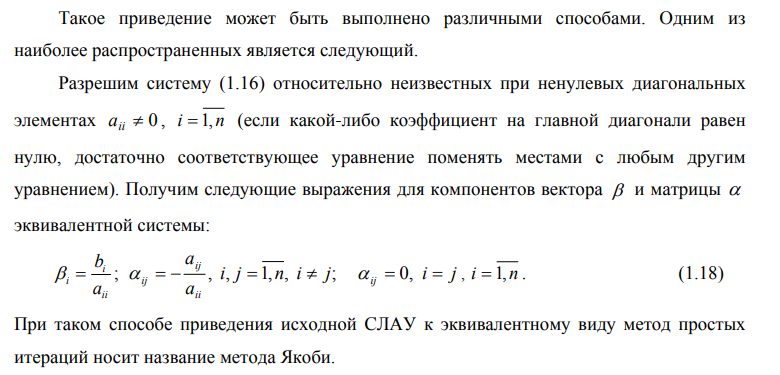
Рещение СЛАУ методом прогонки: [2.0, 4.0, -9.0, 8.0, 4.0]

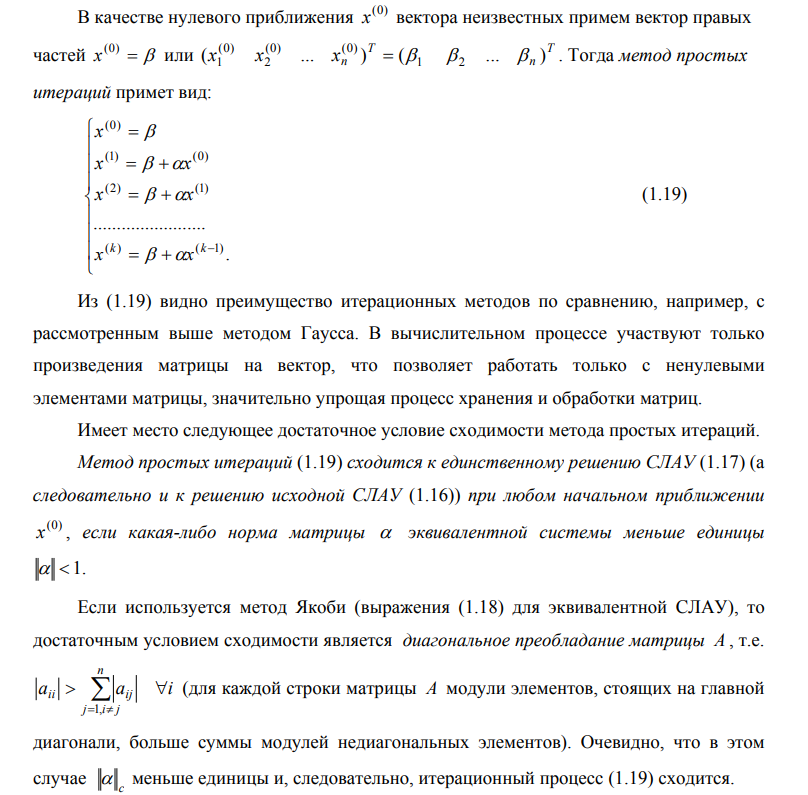
## 1.3 Метод простых итераций и метод Зейделя

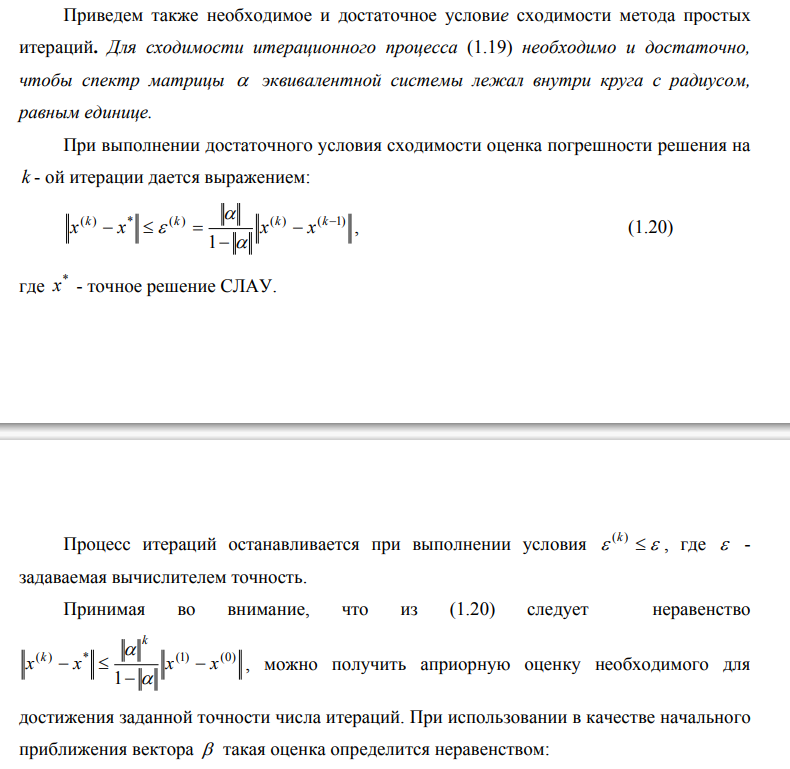
Реализовать метод простых итераций и метод Зейделя в виде программ, задавая в качестве входных данных матрицу системы, вектор правых частей и точность вычислений. Используя разработанное программное обеспечение, решить СЛАУ. Проанализировать количество итераций, необходимое для достижения заданной точности

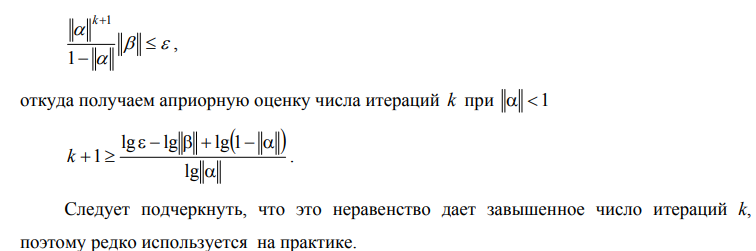




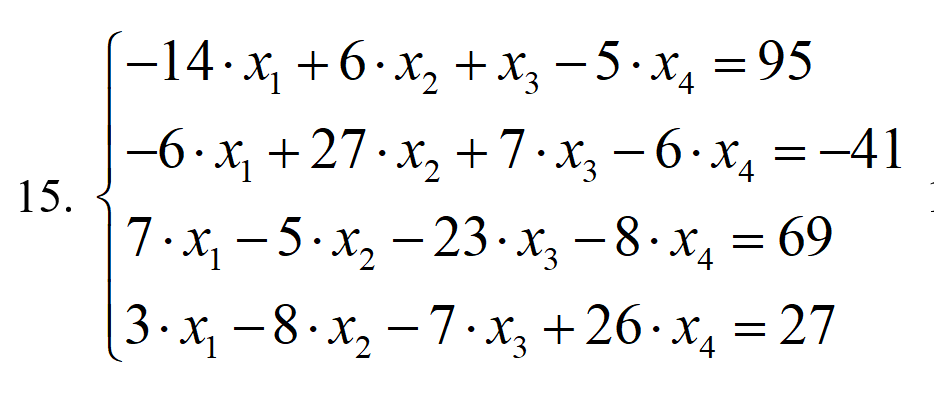








Условие:



Код программы:

# Функция для проверки диагонального преобладания матрицы A

def check\_matrix(A):

    n = len(A)

    flag = False

    # Проход по всем строкам

    for i in range(n):

        diagonal = abs(A[i][i])

        row\_sum = sum(abs(A[i][j]) for j in range(n) if j != i)  # сумма элементов строки кроме диагонали

        if diagonal <= row\_sum:

            # Если по строке не выполнено преобладание, проверяем по столбцу

            flag = False

            for i in range(n):

                diagonal = abs(A[i][i])

                column\_sum = sum(abs(A[j][i]) for j in range(n) if j != i)  # сумма элементов столбца кроме диагонали

                if diagonal <= column\_sum:

                    return False  # Сходимость не гарантирована

                elif diagonal > column\_sum:

                    flag = True

        elif diagonal > row\_sum:

            flag = True  # Сходимость возможна

    return flag  # True если хотя бы одна проверка выполнена

# Метод простых итераций для решения СЛАУ

def simple\_iteration\_method(A: list[list[int]], b: list[int], x0: list[int],

                            tolerance=1e-10, max\_iterations=1000):

    n = len(b)

    # Строим матрицу C и вектор d для итерационного процесса

    C = [[-A[i][j] / A[i][i] if i != j else 0 for j in range(n)] for i in range(n)]

    d = [b[i] / A[i][i] for i in range(n)]

    X0 = x0.copy()  # Начальное приближение

    iteration = 0

    iteration\_history = []

    residual\_history = []

    # Итерационный процесс

    while iteration < max\_iterations:

        # Вычисляем новое приближение

        X = [sum(C[i][j] \* X0[j] for j in range(n)) + d[i] for i in range(n)]

        # Вычисляем погрешность (максимальное изменение компоненты)

        residual = max(abs(X[i] - X0[i]) for i in range(n))

        # Сохраняем итерацию и погрешность

        iteration\_history.append(iteration + 1)

        residual\_history.append(residual)

        # Проверка достижения нужной точности

        if residual <= tolerance:

            # Дополнительная проверка A \* X ≈ b

            for i in range(n):

                calc = sum(A[i][j] \* X[j] for j in range(n))

                if abs(calc - b[i]) > 1e-6:

                    raise ValueError("СЛАУ не решена точно")

            # Возвращаем результат

            return C, X, iteration + 1, iteration\_history, residual\_history

        X0 = X.copy()  # Переход к следующей итерации

        iteration += 1

    # Если не сошлось за max\_iterations итераций

    raise Exception("Метод не сошёлся за максимальное количество итераций")

# Метод Зейделя для решения СЛАУ

def gauss\_seidel(A: list[list[int]], b: list[int], x0: list[int],

                 tolerance=1e-10, max\_iterations=1000):

    n = len(b)

    x = x0.copy()  # Начальное приближение

    iteration = 0

    iteration\_history = []

    residual\_history = []

    # Итерационный процесс

    while iteration < max\_iterations:

        for i in range(n):

            # Разделяем на сумму до и после текущего элемента

            sum1 = sum(A[i][j] \* x[j] for j in range(i))

            sum2 = sum(A[i][j] \* x[j] for j in range(i + 1, n))

            # Вычисляем новую компоненту

            x[i] = (b[i] - sum1 - sum2) / A[i][i]

        # Вычисляем невязку (разницу между A \* x и b)

        residual = [b[i] - sum(A[i][j] \* x[j] for j in range(n)) for i in range(n)]

        max\_residual = max(abs(r) for r in residual)

        # Сохраняем итерацию и невязку

        iteration\_history.append(iteration + 1)

        residual\_history.append(max\_residual)

        # Проверка достижения нужной точности

        if max\_residual < tolerance:

            return x, iteration + 1, iteration\_history, residual\_history

        iteration += 1

    # Если не сошлось за max\_iterations итераций

    raise Exception("Метод не сошёлся за максимальное количество итераций")

def validate\_solution(A, x, b):

    computed\_b = [sum(A[i][j] \* x[j] for j in range(len(x))) for i in range(len(A))]

    print("Проверка перемножения A \* x:")

    for i in range(len(b)):

        print(f"Строка {i + 1}: {computed\_b[i]:.6f} ≈ {b[i]} {True if abs(computed\_b[i] - b[i]) < 1e-6 else False}")

# Задание СЛАУ

A = [

    [-14, 6, 1, -5],

    [-6, 27, 7, -6],

    [7, -5, -23, -8],

    [3, -8, -7, 26]

]

B = [95, -41, 69, 27]

x0 = [0, 0, 0, 0]  # Начальное приближение

print("Матрица A:", A)

print("Вектор B:", B)

# Решение методом простых итераций

C, solution\_s\_i, iterations\_s\_i, s\_i\_iter\_history, s\_i\_res\_history = simple\_iteration\_method(A, B, x0)

# Проверка условия сходимости

Chek = check\_matrix(A)

if(not Chek):

    print('Предупреждение: матрица не имеет диагонального преобладания. Сходимость не гарантирована')

else:

    print("Условие сходимости метода простых итераций выполнено:", Chek)

# Вывод результата для метода простых итераций

print("Метод простых итераций:")

print("Решение:", solution\_s\_i)

print("Количество итераций:", iterations\_s\_i)

validate\_solution(A, solution\_s\_i, B)

# Решение методом Зейделя

solution\_gauss\_seidel, iterations\_gauss\_seidel, seidel\_iter\_history, seidel\_res\_history = gauss\_seidel(A, B, x0)

# Вывод результата для метода Зейделя

print("\nМетод Зейделя:")

print("Решение:", solution\_gauss\_seidel)

print("Количество итераций:", iterations\_gauss\_seidel)

validate\_solution(A, solution\_gauss\_seidel, B)

Результат:

Матрица A: [[-14, 6, 1, -5], [-6, 27, 7, -6], [7, -5, -23, -8], [3, -8, -7, 26]]

Вектор B: [95, -41, 69, 27]

Условие сходимости метода простых итераций выполнено: True

Метод простых итераций:

Решение: [-7.999999999969478, -2.000000000023576, -5.000000000010847, 2.2219559525638033e-11]

Количество итераций: 48

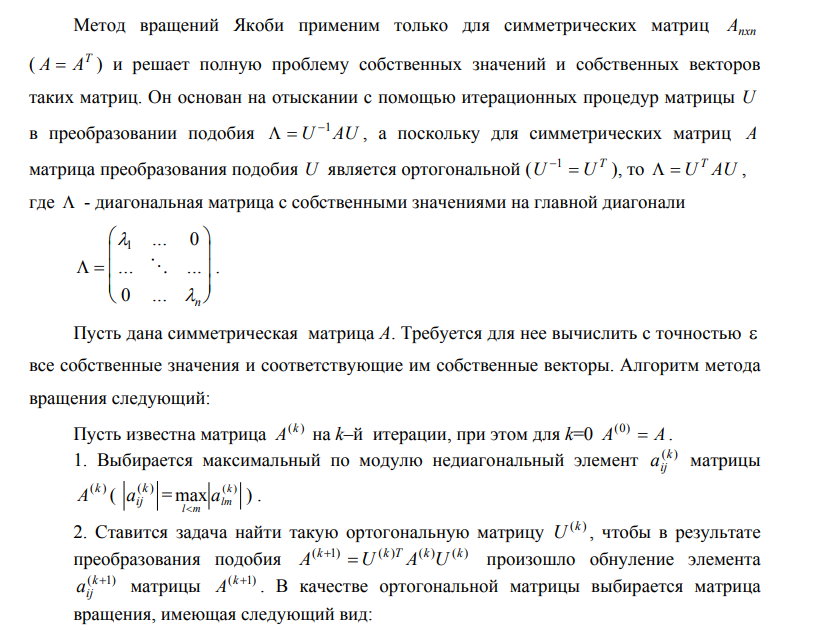
Метод Зейделя:

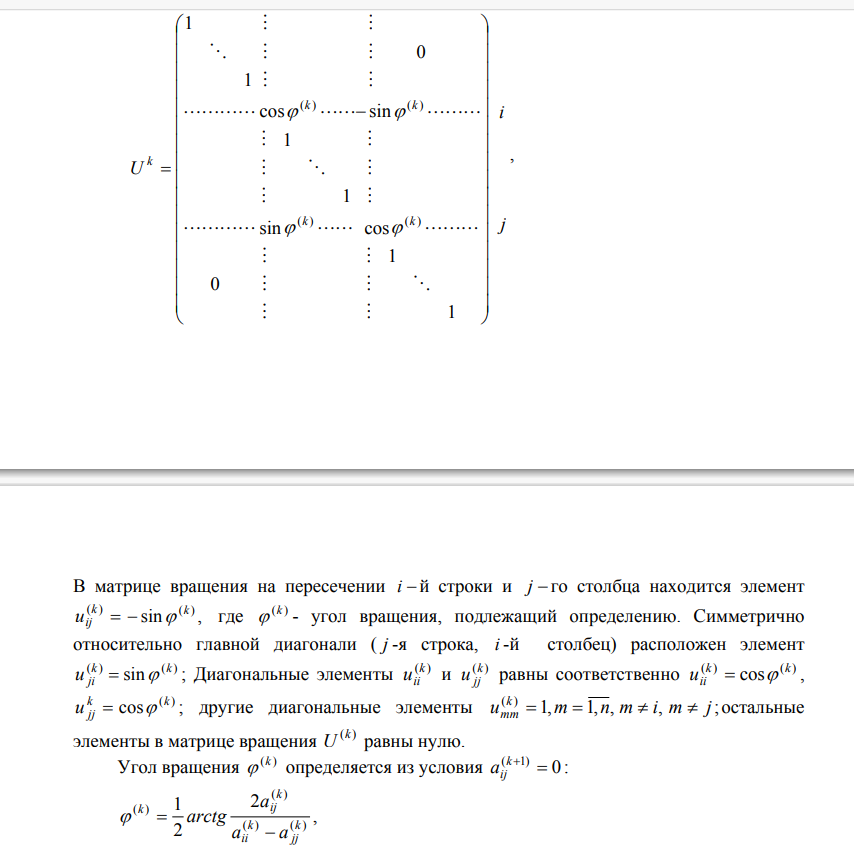
Решение: [-7.999999999996591, -1.9999999999993985, -4.999999999999361, -3.634699379080512e-14]

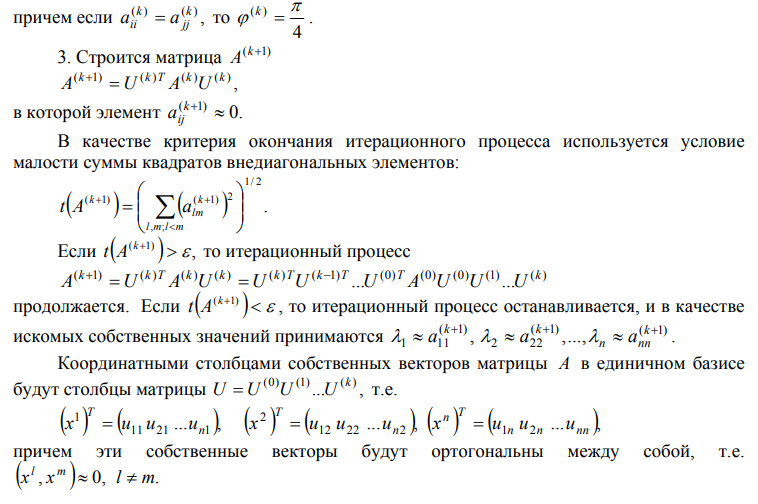
Количество итераций: 17

## 1.4 Метод вращений

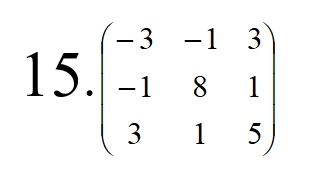
Реализовать метод вращений в виде программы, задавая в качестве входных данных матрицу и точность вычислений. Используя разработанное программное обеспечение, найти собственные значения и собственные векторы симметрических матриц. Проанализировать зависимость погрешности вычислений от числа итераций.







Условие:



Код программы:

import math

# Функция создания единичной матрицы размера n x n

def create\_eye\_matrix(n):

    return [[1.0 if i == j else 0.0 for j in range(n)] for i in range(n)]

# Функция транспонирования матрицы (поворот по диагонали)

def transpose(matrix):

    return [[matrix[j][i] for j in range(len(matrix))] for i in range(len(matrix[0]))]

# Функция умножения матриц A и B

def matrix\_multiply(A, B):

    n = len(A)

    m = len(B[0])

    p = len(B)

    result = [[0.0 for \_ in range(m)] for \_ in range(n)]

    for i in range(n):

        for j in range(m):

            for k in range(p):

                result[i][j] += A[i][k] \* B[k][j]

    return result

# Восстановление исходной матрицы A по её собственным значениям и векторной матрице

def create\_matrix(eigen\_values, eigen\_vectors):

    n = len(eigen\_values)

    # Создаём диагональную матрицу собственных значений (Лямбда)

    Lambda = [[0.0 for \_ in range(n)] for \_ in range(n)]

    for i in range(n):

        Lambda[i][i] = eigen\_values[i]

    # Перемножаем V \* Лямбда

    VL = matrix\_multiply(eigen\_vectors, Lambda)

    # Перемножаем результат на V^T (транспонированную матрицу векторов)

    A\_reconstructed = matrix\_multiply(VL, transpose(eigen\_vectors))

    return A\_reconstructed

# Проверка симметричности матрицы (A[i][j] == A[j][i] для всех i, j)

def check\_symmetry(A):

    n = len(A)

    for i in range(n):

        for j in range(i + 1, n):

            if abs(A[i][j] - A[j][i]) > 1e-10:

                return False

    return True

# Метод вращений Якоби для поиска собственных значений и собственных векторов

def jacobi\_rotation(A, precision=1e-10, max\_iter=100):

    n = len(A)

    eigenvectors = create\_eye\_matrix(n)  # Изначально матрица собственных векторов — единичная

    iteration = 0

    iterations = []

    # Итерации вращений до достижения заданной точности

    while iteration < max\_iter:

        max\_off\_diag = 0  # Наибольший внедиагональный элемент

        p, q = 0, 0

        # Поиск наибольшего внедиагонального элемента

        for i in range(n):

            for j in range(i + 1, n):

                if abs(A[i][j]) > max\_off\_diag:

                    max\_off\_diag = abs(A[i][j])

                    p, q = i, j

        iterations.append(iteration + 1)

        # Если все внедиагональные элементы малы — выходим

        if max\_off\_diag < precision:

            break

        # Вычисление угла поворота phi для зануления A[p][q]

        if A[p][p] == A[q][q]:

            phi = math.pi / 4

        else:

            phi = 0.5 \* math.atan(2 \* A[p][q] / (A[p][p] - A[q][q]))

        c = math.cos(phi)  # cos(phi)

        s = math.sin(phi)  # sin(phi)

        # Формируем матрицу вращения R

        R = create\_eye\_matrix(n)

        R[p][p] = c

        R[p][q] = -s

        R[q][p] = s

        R[q][q] = c

        # Применяем вращение: A' = R^T \* A \* R

        A = matrix\_multiply(matrix\_multiply(transpose(R), A), R)

        # Обновляем матрицу собственных векторов

        eigenvectors = matrix\_multiply(eigenvectors, R)

        iteration += 1

    # Собственные значения — диагональные элементы A после итераций

    eigenvalues = [A[i][i] for i in range(n)]

    return eigenvalues, eigenvectors, iterations

# === Основной блок программы ===

try:

    # Исходная симметричная матрица A

    A = [

        [-3, -1, 3],

        [-1, 8, 1],

        [3, 1, 5]

    ]

    print("Исходная матрица A:\n")

    for row in A:

        print([round(val, 4) for val in row])

    # Проверка симметричности

    if not check\_symmetry(A):

        raise ValueError("Матрица не является симметричной.")

    # Запуск метода Якоби

    eigenvalues, eigenvectors, iter\_history = jacobi\_rotation([row[:] for row in A])

    print("\nСобственные значения:")

    print([round(val, 6) for val in eigenvalues])

    print("\nМатрица собственных векторов:")

    for row in eigenvectors:

        print([round(val, 6) for val in row])

    print(f"\nКоличество итераций: {len(iter\_history)}")

    # Восстановление матрицы из найденных собственных значений и векторов

    A\_reconstructed = create\_matrix(eigenvalues, eigenvectors)

    print("\nПроверка восстановления матрицы A:")

    for row in A\_reconstructed:

        print([round(val, 4) for val in row])

# Вывод ошибок, если матрица не симметрична

except ValueError as e:

    print(f"Ошибка: {e}")

Результат:

Исходная матрица A:

[-3, -1, 3]

[-1, 8, 1]

[3, 1, 5]

Собственные значения:

[-4.13231, 8.303678, 5.828632]

Матрица собственных векторов:

[0.941462, -0.010281, 0.336961]

[0.10403, 0.959614, -0.261378]

[-0.320666, 0.281132, 0.90451]

Количество итераций: 8

\Проверка A:

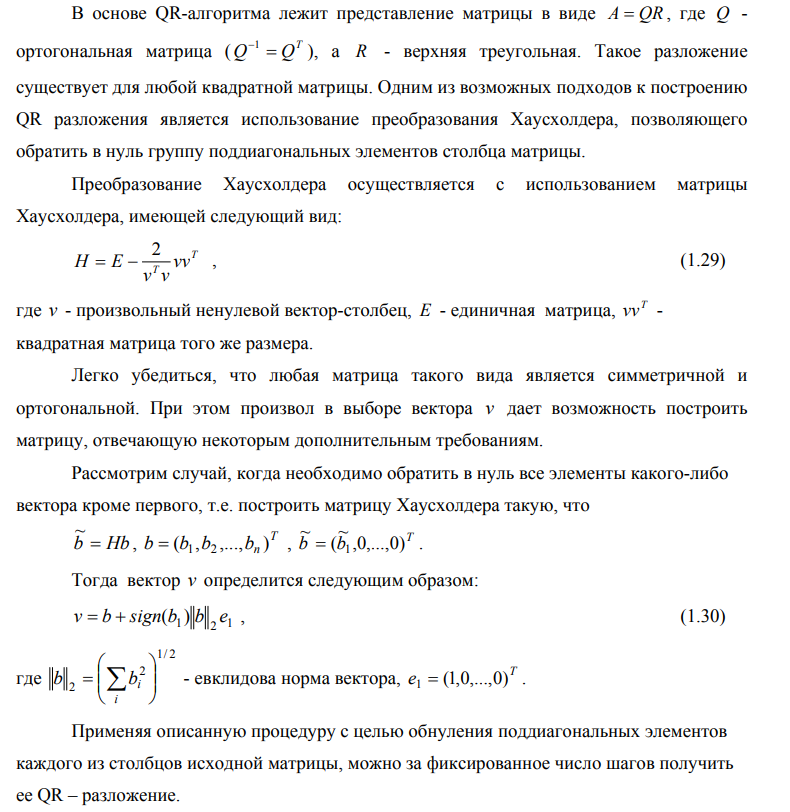
[-3.0, -1.0, 3.0]

[-1.0, 8.0, 1.0]

[3.0, 1.0, 5.0]

## 1.5 QR - разложение

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



Условие:

15. 

Код программы:

# Функция умножения двух матриц A и B

def multiply\_matrix(A, B):

    # Проверка, совместимы ли размеры для умножения

    if len(A[0]) != len(B):

        raise ValueError("Невозможно умножить матрицы такого размера")

    # Размер результата

    rows\_A = len(A)

    cols\_B = len(B[0])

    # Инициализация результирующей матрицы нулями

    C = [[0 for \_ in range(cols\_B)] for \_ in range(rows\_A)]

    # Умножение матриц по определению

    for i in range(rows\_A):

        for j in range(cols\_B):

            for k in range(len(B)):

                C[i][j] += A[i][k] \* B[k][j]

    return C

# Функция транспонирования матрицы A

def transpose\_matrix(A):

    return [[A[j][i] for j in range(len(A))] for i in range(len(A[0]))]

# Функция для создания единичной матрицы размера n x n

def identity\_matrix(n):

    return [[1 if i == j else 0 for j in range(n)] for i in range(n)]

# Функция для выполнения QR-разложения методом отражений Хаусхолдера

def qr\_decomposition(A):

    n = len(A)

    Q = identity\_matrix(n)  # Начинаем с единичной матрицы Q

    A\_k = [row.copy() for row in A]  # Работаем с копией матрицы A

    # Итерация по каждому столбцу, кроме последнего

    for i in range(n - 1):

        v = [0] \* n

        sum\_sq = 0

        # Считаем норму вектора столбца начиная с элемента i

        for j in range(i, n):

            sum\_sq += A\_k[j][i] \*\* 2

        norm = sum\_sq \*\* 0.5  # Евклидова норма

        # Выбор знака для повышения устойчивости

        sign = 1 if A\_k[i][i] >= 0 else -1

        v[i] = A\_k[i][i] + sign \* norm  # Формируем вектор отражения

        # Заполняем оставшиеся элементы вектора

        for j in range(i + 1, n):

            v[j] = A\_k[j][i]

        # Строим матрицу v \* v^T (матрицу ранга 1)

        v\_matrix = [[v[i] \* v[j] for j in range(n)] for i in range(n)]

        v\_norm = sum(v[i]\*\*2 for i in range(n))  # Квадрат нормы вектора v

        # Строим матрицу Хаусхолдера H

        H = identity\_matrix(n)

        for x in range(n):

            for y in range(n):

                H[x][y] -= 2 \* v\_matrix[x][y] / v\_norm

        Q = multiply\_matrix(Q, H)

        A\_k = multiply\_matrix(H, A\_k)

    # Возвращаем ортогональную матрицу Q и верхнетреугольную A\_k (R)

    return Q, A\_k

# Функция для вычисления собственных значений через QR-алгоритм

def calculate\_eigenvalues(A, eps=1e-10, max\_iter=1000):

    A\_k = [row.copy() for row in A]  # Копируем исходную матрицу для работы

    # Основной цикл QR-алгоритма

    for \_ in range(max\_iter):

        converged = True  # Флаг сходимости

        # Проверка, что все элементы ниже главной диагонали малы

        for i in range(1, len(A)):

            for j in range(i):

                if abs(A\_k[i][j]) > eps:

                    converged = False

                    break

            if not converged:

                break

        if converged:

            break  # Если сходимость достигнута, выходим

        # QR-разложение текущей матрицы

        Q, R = qr\_decomposition(A\_k)

        # Пересобираем матрицу как R \* Q (а не Q \* R!)

        A\_k = multiply\_matrix(R, Q)

    # Собственные значения — диагональные элементы A\_k после сходимости

    eigenvalues = [A\_k[i][i] for i in range(len(A\_k))]

    return eigenvalues

# === Пример использования ===

# Исходная матрица

matrix = [

    [1, 7, -1],

    [-2, 2, -2],

    [9, -7, 3]

]

# Вызываем алгоритм для вычисления собственных значений

eigenvalues = calculate\_eigenvalues(matrix)

# Выводим результат

print(f'Полученные собственные значения: {eigenvalues}')

Результат:

Полученные собственные значения: [1.6475929460379108, 6.622176049434543, -2.269768995472492]

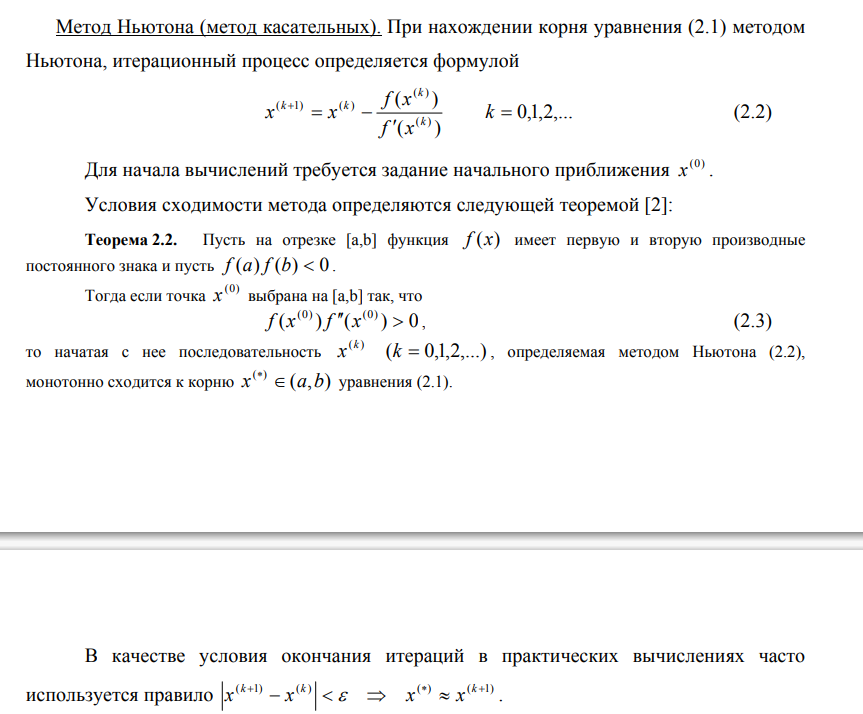
# **2 Лабораторная работа**

## 2.1 Решение нелинейных уравнений

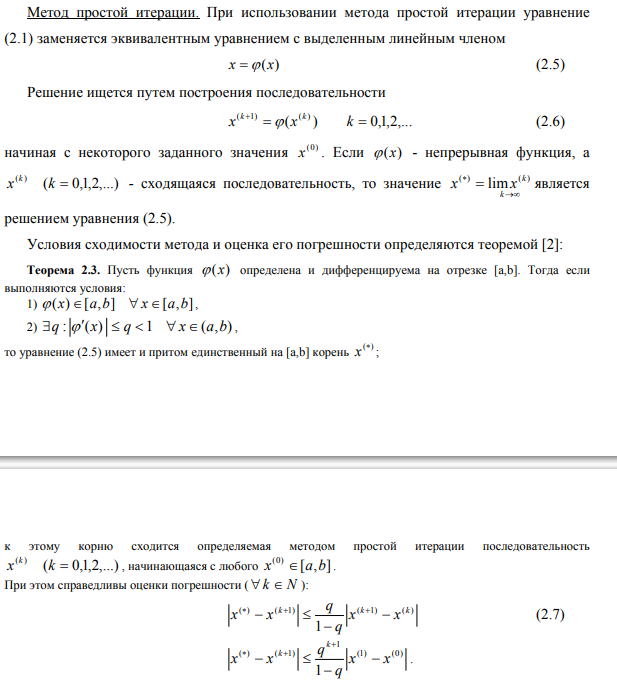
Реализовать методы простой итерации и Ньютона решения нелинейных уравнений в виде программ, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения найти положительный корень нелинейного уравнения (начальное приближение определить графически). Проанализировать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

Условие:

****1) Метод Ньютона



2)Метод простой итерации



Код программы:

import math

import matplotlib.pyplot as plt

# Функция f(x) = sin(x) - x^2 + 1

def f(x):

    return math.sin(x) - x\*\*2 + 1

# Первая производная

def df(x):

    return math.cos(x) - 2 \* x

# Вторая производная

def d2f(x):

    return -math.sin(x) - 2

# Итерационная функция phi(x) = sqrt(sin(x) + 1)

def phi(x):

    value = math.sin(x) + 1

    if value < 0:

        raise ValueError("Отрицательный аргумент под корнем")

    return math.sqrt(value)

# Производная phi(x)

def phi\_derivative(x):

    value = math.sin(x) + 1

    if value <= 0:

        raise ValueError("Отрицательный аргумент под корнем")

    return 0.5 \* math.cos(x) / math.sqrt(value)

# Метод простой итерации

def simple\_iteration(phi, x0, epsilon, a, b, max\_iter=1000):

    x\_prev = x0

    iterations = 0

    for \_ in range(max\_iter):

        x\_next = phi(x\_prev)

        q = check\_simple\_iteration\_conditions(phi, x\_next, a, b)

        if abs(x\_next - x\_prev) \* q / (1 - q) < epsilon:

            break

        x\_prev = x\_next

        iterations += 1

    return x\_next, iterations

# Проверка условий на концах интервала для метода простой итерации

def check\_simple\_iteration(a, b):

    try:

        da = abs(phi\_derivative(a))

        db = abs(phi\_derivative(b))

        if da < 1 and db < 1:

            return True

        else:

            print(f"Производные на концах интервала не удовлетворяют условию сходимости: |phi'(a)|={da}, |phi'(b)|={db}")

            return False

    except ValueError:

        print("Ошибка при вычислении производной phi на концах интервала")

        return False

# Проверка условий сходимости внутри интервала

def check\_simple\_iteration\_conditions(phi, x0, a, b, q\_max=0.99):

    num\_points = 100

    h = (b - a) / num\_points

    max\_derivative = 0

    for i in range(num\_points + 1):

        x = a + i \* h

        try:

            derivative = phi\_derivative(x)

            current\_abs\_derivative = abs(derivative)

            if current\_abs\_derivative > max\_derivative:

                max\_derivative = current\_abs\_derivative

        except ValueError:

            continue

    if max\_derivative >= q\_max:

        print(f"Условие сходимости не выполнено (max |phi'| = {max\_derivative:.4f})")

        return False

    return max\_derivative

# Метод Ньютона

def newton\_method(f, df, x0, epsilon=1e-6, max\_iter=100):

    x\_prev = x0

    iterations = 0

    for \_ in range(max\_iter):

        fx = f(x\_prev)

        dfx = df(x\_prev)

        if abs(dfx) < 1e-12:

            raise ValueError("Производная слишком близка к нулю!")

        x\_next = x\_prev - fx / dfx

        if abs(x\_next - x\_prev) < epsilon:

            break

        x\_prev = x\_next

        iterations += 1

    return x\_next, iterations

# Проверка условий для метода Ньютона

def check\_newton\_conditions(f, d2f, a, b):

    if f(a) \* f(b) >= 0:

        print("Начальное приближение вне отрезка [a, b]")

        return False, None

    if abs(d2f(a) \* f(a)) < df(a) \*\* 2:

        return True, a

    elif abs(d2f(b) \* f(b)) < df(b) \*\* 2:

        return True, b

    else:

        print("Не выполняется условие сходимости метода Ньютона")

        return False, None

# ===== Проверка на интервале [0.5, 1.5] =====

a, b = 0.5, 1.5

epsilon = 1e-8

print("\nПроверка интервала [0.5, 1.5]:")

if check\_simple\_iteration(a, b):

    x0\_si = a if abs(phi\_derivative(a)) < abs(phi\_derivative(b)) else b

    root\_si, iter\_si = simple\_iteration(phi, x0\_si, epsilon, a, b)

    print(f"Метод простой итерации: корень = {root\_si:.8f}, итераций = {iter\_si}")

newton\_ok, x0\_nm = check\_newton\_conditions(f, d2f, a, b)

if newton\_ok:

    root\_nm, iter\_nm = newton\_method(f, df, x0\_nm, epsilon)

    print(f"Метод Ньютона: корень = {root\_nm:.8f}, итераций = {iter\_nm}")

# ===== Проверка на интервале [-1.0, -0.5] =====

a, b = -1.0, -0.5

print("\nПроверка интервала [-1.0, -0.5]:")

if check\_simple\_iteration(a, b):

    x0\_si = a if abs(phi\_derivative(a)) < abs(phi\_derivative(b)) else b

    root\_si, iter\_si = simple\_iteration(phi, x0\_si, epsilon, a, b)

    print(f"Метод простой итерации: корень = {root\_si:.8f}, итераций = {iter\_si}")

newton\_ok, x0\_nm = check\_newton\_conditions(f, d2f, a, b)

if newton\_ok:

    root\_nm, iter\_nm = newton\_method(f, df, x0\_nm, epsilon)

    print(f"Метод Ньютона: корень = {root\_nm:.8f}, итераций = {iter\_nm}")

# ===== График функции =====

x\_values = [i \* 0.01 for i in range(-100, 200)]  # от -1 до 2

y\_values = [f(x) for x in x\_values]

plt.plot(x\_values, y\_values, label=r'$f(x) = \sin(x) - x^2 + 1$')

plt.axhline(0, color='black', linewidth=0.5)

plt.axvline(0, color='black', linewidth=0.5)

plt.xlabel('x')

plt.ylabel('f(x)')

plt.title('График функции f(x) = sin(x) - x^2 + 1')

plt.grid(True)

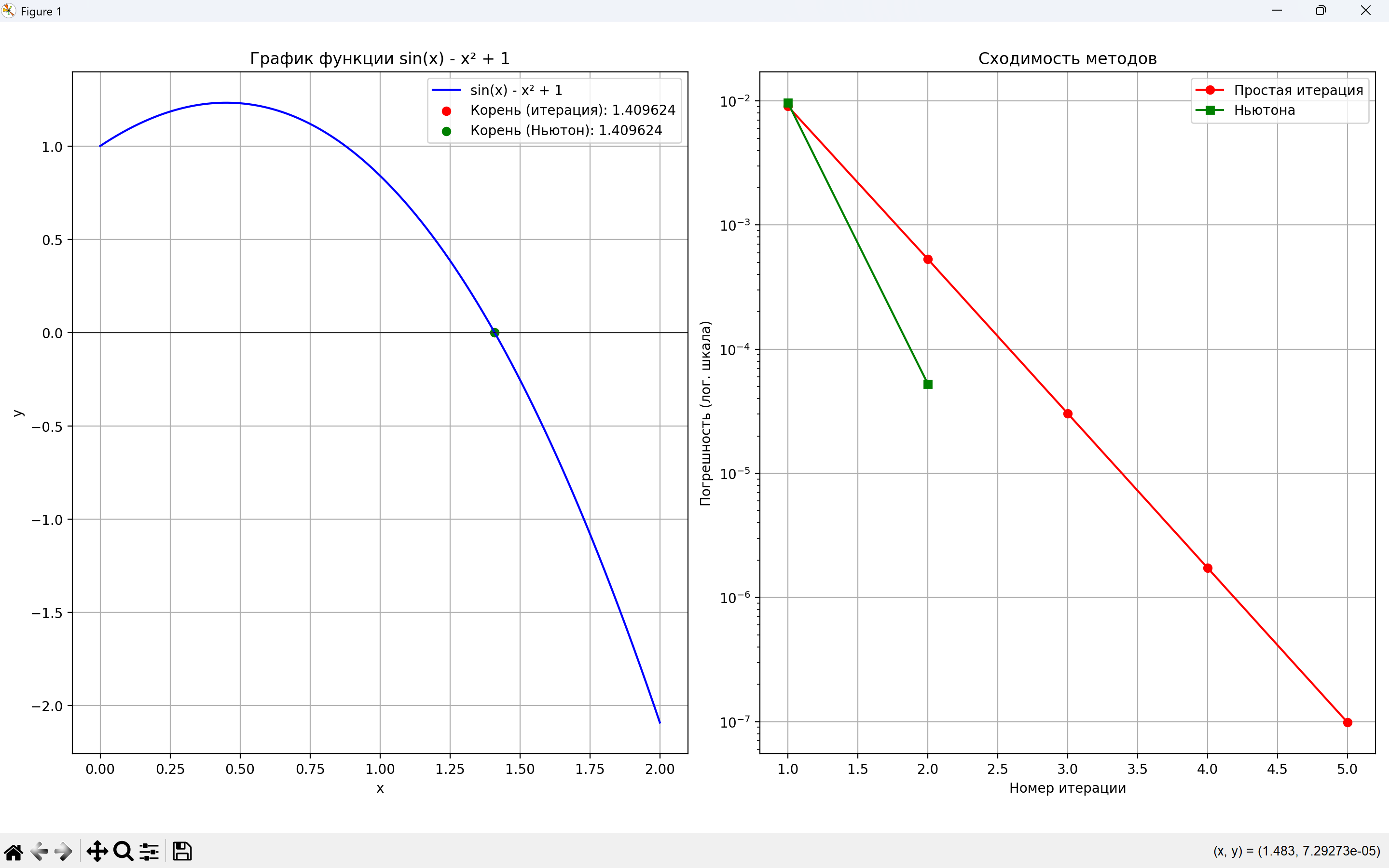
plt.legend()

plt.show()

Результат:

Метод простой итерации: корень = 1.409624, итераций = 5

Метод Ньютона: корень = 1.409624, итераций = 2

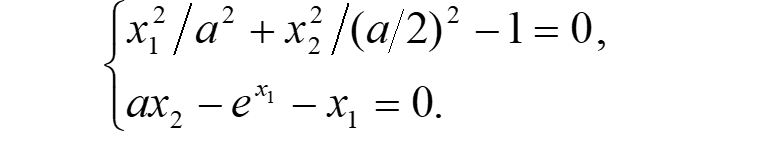


## 2.2 Решение систем нелинейных уравнений

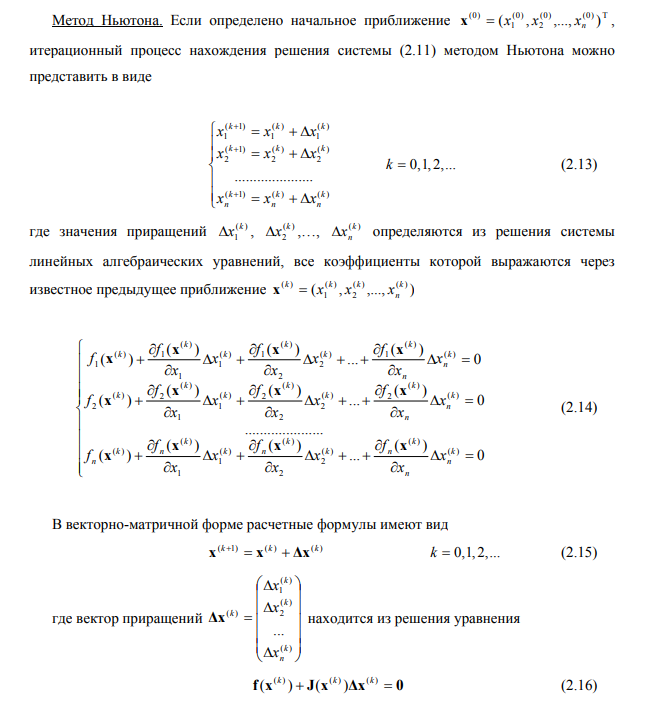
Реализовать методы простой итерации и Ньютона решения систем нелинейных уравнений в виде программного кода, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения решить систему нелинейных уравнений (при наличии нескольких решений найти то из них,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

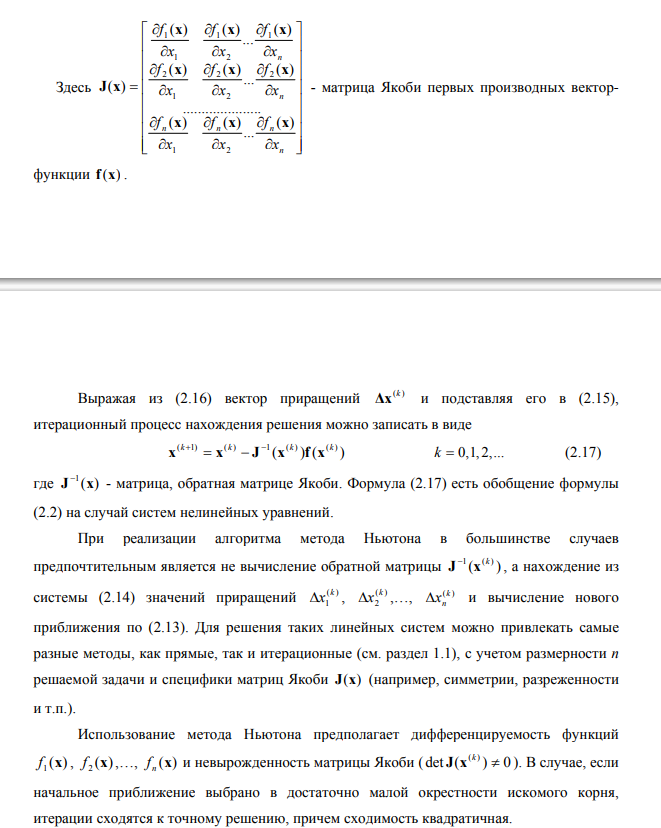
Условие:

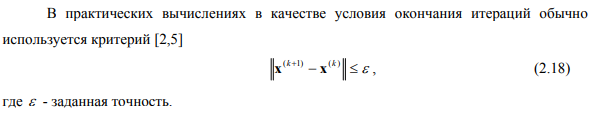
a = 4



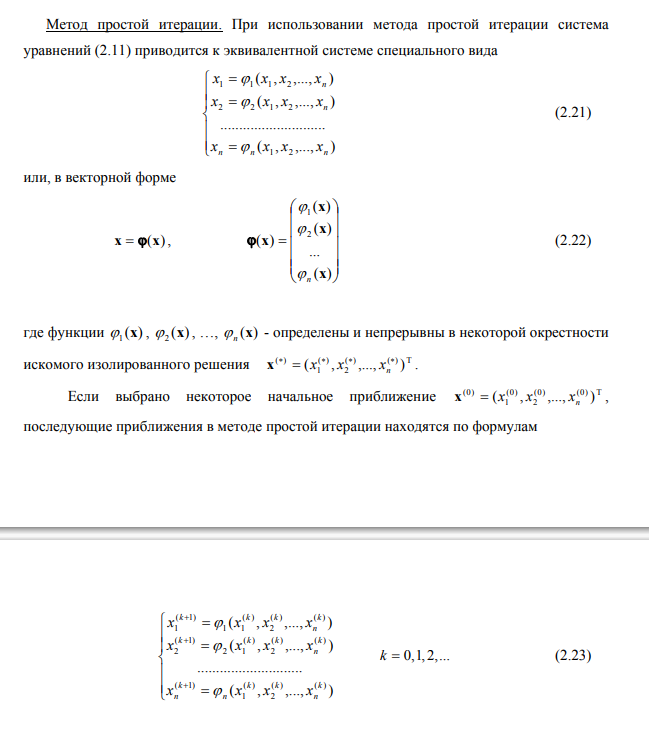
1) Метод Ньютона

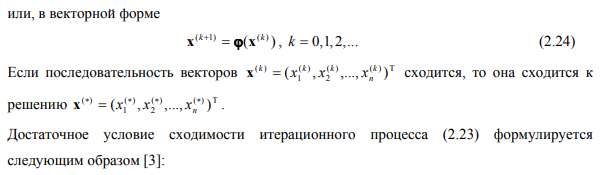






2)Метод простой итерации





Код программы:

import math

import matplotlib.pyplot as plt

# Параметр уравнений

a = 4

def f\_system(x):

    """Система нелинейных уравнений"""

    return [

        x[0]\*\*2 / (a\*\*2) + x[1]\*\*2 / ((a/2)\*\*2) - 1,

        a \* x[1] - math.exp(x[0]) - x[0]

    ]

def J\_system(x):

    """Матрица Якоби"""

    return [

        [2 \* x[0] / (a\*\*2), 2 \* x[1] / ((a/2)\*\*2)],

        [-math.exp(x[0]) - 1, a]

    ]

def phi\_system(x):

    """Функция для метода простой итерации"""

    try:

        x2\_new = math.sqrt((1 - x[0]\*\*2 / (a\*\*2)) \* ((a/2)\*\*2))

        x1\_new = math.log(a \* x2\_new - x[0] + 1)  # Приближенная итерация

    except (ValueError, ZeroDivisionError):

        x2\_new = x[1]

        x1\_new = x[0]

    return [x1\_new, x2\_new]

def phi\_derivative(x):

    """Приближённая матрица производных (нулевая для оценки сходимости)"""

    return [

        [0, 0],

        [0, 0]

    ]

def compute\_spectral\_norm(J):

    norm1 = abs(J[0][0]) + abs(J[0][1])

    norm2 = abs(J[1][0]) + abs(J[1][1])

    return max(norm1, norm2)

def simple\_iteration(phi, phi\_der, x0, epsilon, max\_iter=100):

    x = x0.copy()

    flag = 1

    for k in range(max\_iter):

        x\_new = phi(x)

        J\_phi = phi\_der(x)

        q = compute\_spectral\_norm(J\_phi)

        if q >= 1:

            flag = 0

            raise ValueError(f"Условие сходимости нарушено: q = {q:.6f} >= 1 на итерации {k}")

        delta\_norm = max(abs(x\_new[0] - x[0]), abs(x\_new[1] - x[1]))

        if delta\_norm \* q / (1 - q) < epsilon:

            print(f"Решение найдено за {k + 1} итераций")

            break

        x = x\_new

    else:

        print(f"Достигнуто максимальное число итераций {max\_iter}")

    return x, k + 1, flag

def newton\_method(f, J, x0, epsilon, max\_iter=100):

    x = x0.copy()

    for k in range(max\_iter):

        f\_val = f(x)

        J\_val = J(x)

        det = J\_val[0][0] \* J\_val[1][1] - J\_val[0][1] \* J\_val[1][0]

        if abs(det) < 1e-12:

            raise ValueError("Матрица Якоби вырождена")

        det\_x = (-f\_val[0] \* J\_val[1][1]) - (J\_val[0][1] \* -f\_val[1])

        det\_y = (J\_val[0][0] \* -f\_val[1]) - (-f\_val[0] \* J\_val[1][0])

        delta\_x = det\_x / det

        delta\_y = det\_y / det

        x\_new = [x[0] + delta\_x, x[1] + delta\_y]

        if max(abs(x\_new[0] - x[0]), abs(x\_new[1] - x[1])) < epsilon:

            break

        x = x\_new

    return x, k + 1

def check\_newton\_convergence(x0):

    J = J\_system(x0)

    det = J[0][0] \* J[1][1] - J[0][1] \* J[1][0]

    return abs(det) != 0

def plot\_system():

    def f1(x2):

        try:

            return math.sqrt(16 \* (1 - x2\*\*2 / 4))

        except ValueError:

            return float('nan')

    def f2(x1):

        return (math.exp(x1) + x1) / a

    x2\_values = [i \* 0.05 for i in range(-40, 40)]

    x1\_values\_f1 = [f1(x2) for x2 in x2\_values]

    x1\_values = [i \* 0.05 for i in range(-40, 40)]

    x2\_values\_f2 = [f2(x1) for x1 in x1\_values]

    plt.figure(figsize=(12, 10))

    plt.plot(x1\_values\_f1, x2\_values, label=r'$\frac{x\_1^2}{16} + \frac{x\_2^2}{4} - 1 = 0$')

    plt.plot(x1\_values, x2\_values\_f2, label=r'$4x\_2 - e^{x\_1} - x\_1 = 0$')

    plt.xlabel('x1')

    plt.ylabel('x2')

    plt.title('Графическое определение начального приближения')

    plt.grid(True)

    plt.legend()

    plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

    x0 = [1.0, 0.5]  # Подобрано исходя из графика

    epsilon = 1e-7

    print("\nМетод простой итерации:")

    try:

        x\_si, iter\_si, flag = simple\_iteration(phi\_system, phi\_derivative, x0, epsilon)

        print(f"Решение: x1 = {x\_si[0]:.6f}, x2 = {x\_si[1]:.6f}")

        print(f"Итераций: {iter\_si}")

    except ValueError as e:

        print(e)

    print("\nМетод Ньютона:")

    try:

        x\_nm, iter\_nm = newton\_method(f\_system, J\_system, x0, epsilon)

        print(f"Решение: x1 = {x\_nm[0]:.6f}, x2 = {x\_nm[1]:.6f}")

        print(f"Итераций: {iter\_nm}")

    except ValueError as e:

        print(e)

    print("\nПроверка сходимости метода Ньютона:")

    if check\_newton\_convergence(x0):

        print("Условия сходимости выполнены")

    else:

        print("Условия сходимости не выполнены")

    print("\nПостроение графика для выбора начального приближения:")

    plot\_system()

Результат:

Метод простой итерации:

Решение найдено за 1 итераций

Решение: x1 = 1.000000, x2 = 0.500000

Итераций: 1

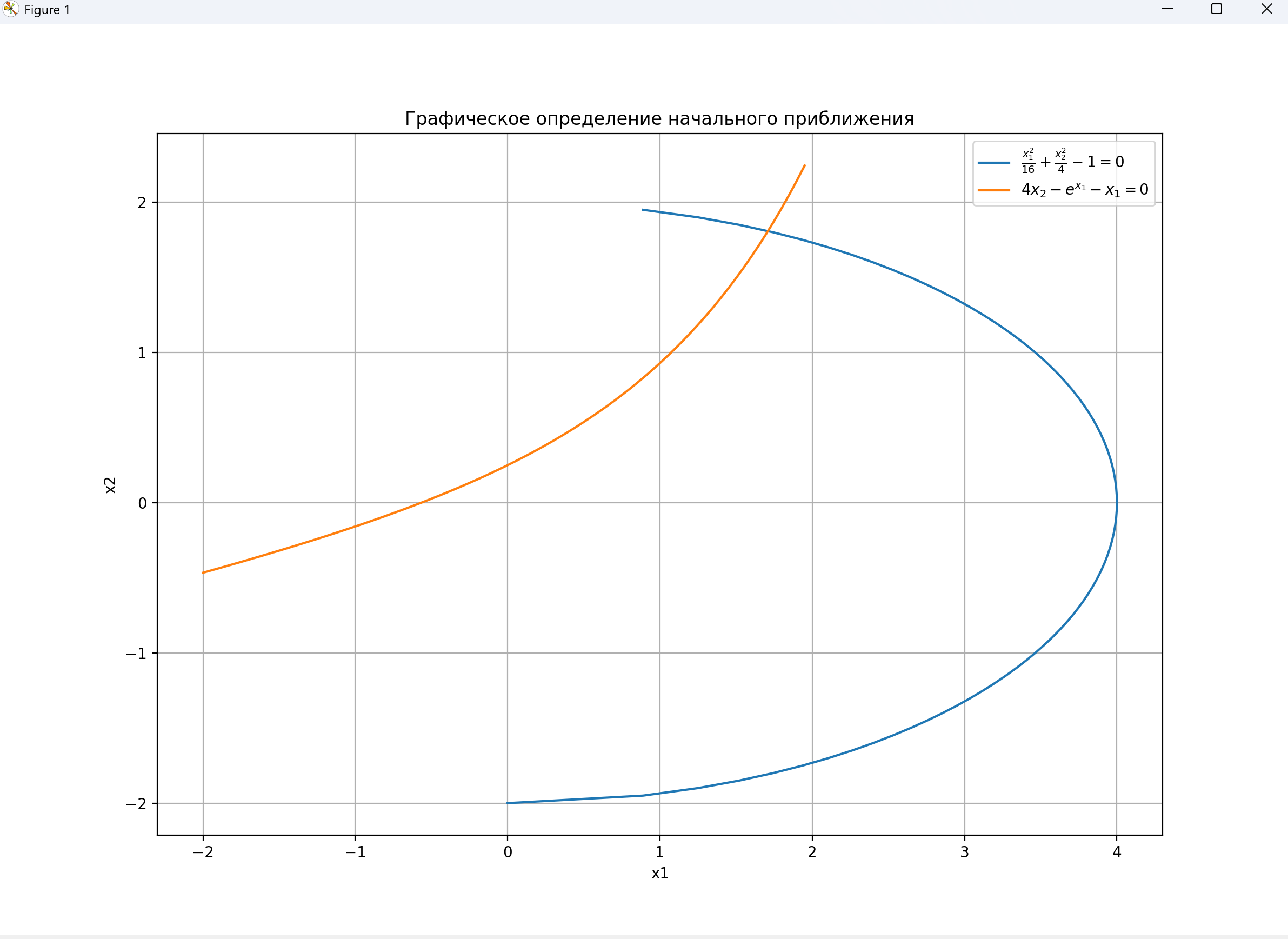
Метод Ньютона:

Решение: x1 = 1.709085, x2 = 1.808247

Итераций: 7

Проверка сходимости метода Ньютона:

Условия сходимости выполнены

Построение графика для выбора начального приближения:

3.1

3.2

3.3

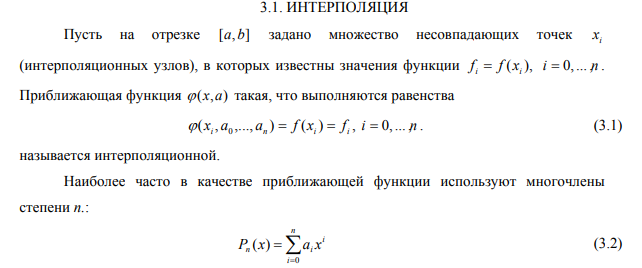
3.4

3.5

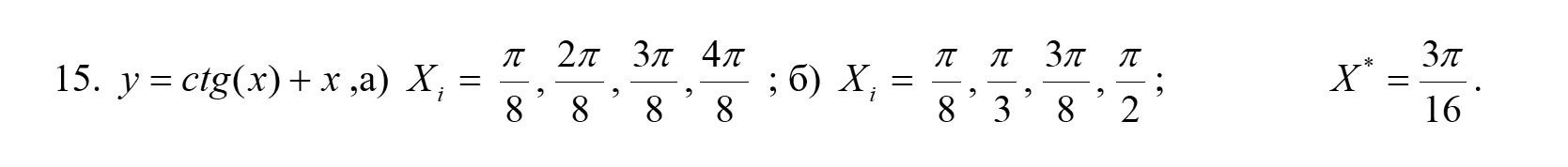
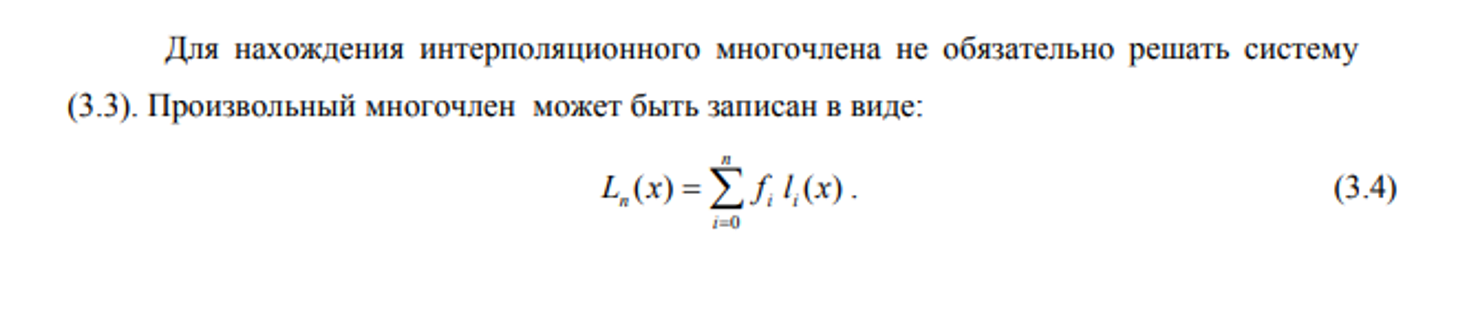
# **3 Лабораторная работа**

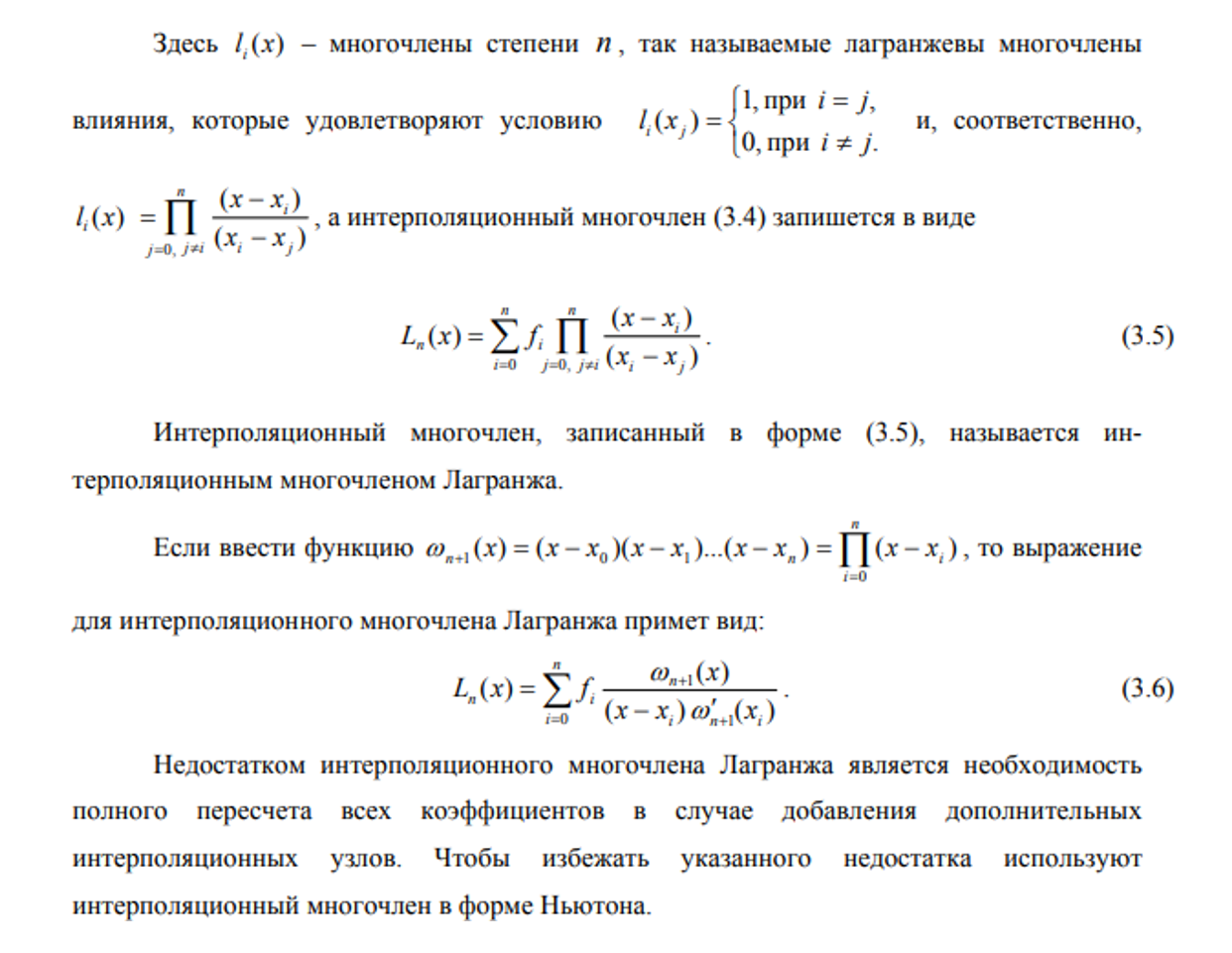
## 3.1 Интерполяция

Используя таблицу значений  функции , вычисленных в точках  построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки . Вычислить значение погрешности интерполяции в точке .

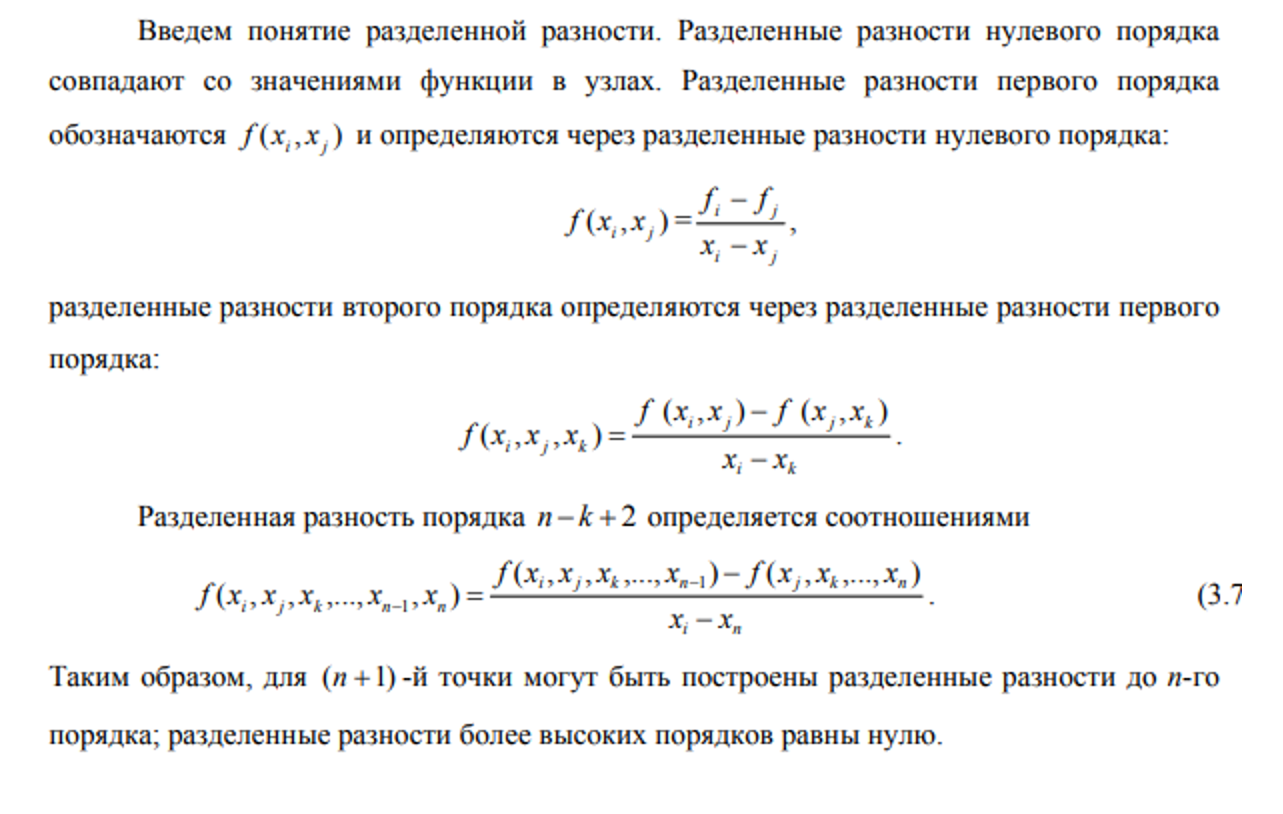


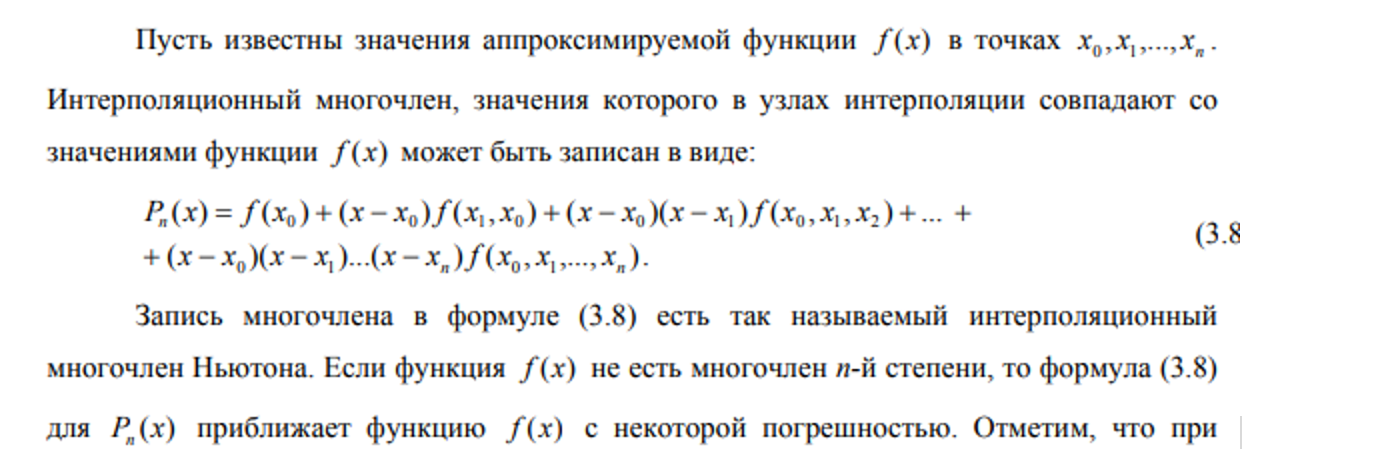
Условие:

А) Лагранж



Б) Ньютон





Код программы:

import math

import matplotlib.pyplot as plt

# функция из условия

def f(x):

    return 1/math.tan(x) + x   # ctg(x) + x

def lagrange\_interpolation(x, y, X\_):

    polynom\_str = 'L(x) ='

    polynom\_X\_ = 0

    for i in range(len(x)):

        cur\_enum, cur\_denom = 1, 1

        for j in range(len(x)):

            if i == j:

                continue

            cur\_enum \*= (X\_[0] - x[j])

            cur\_denom \*= (x[i] - x[j])

        polynom\_str += f' + {(y[i] / cur\_denom):.5f}\*' + ''.join([f'(x-{x[j]:.2f})' for j in range(len(x)) if j != i])

        polynom\_X\_ += y[i] \* cur\_enum / cur\_denom

    return polynom\_str, polynom\_X\_

def newton\_interpolation(x, y, X\_):

    n = len(x)

    difference = [y[i] for i in range(n)]

    for i in range(1, n):

        for j in range(n - 1, i - 1, -1):

            difference[j] = float(difference[j] - difference[j - 1]) / float(x[j] - x[j - i])

    polynom\_str = 'P(x) = '

    polynom\_X\_ = 0

    cur\_coeffs = 1

    for i in range(n):

        polynom\_X\_ += cur\_coeffs \* difference[i]

        if i == 0:

            polynom\_str += f'{difference[i]:.5f}'

        else:

            polynom\_str += ' + ' + ''.join([f'(x-{x[j]:.2f})' for j in range(i)]) + f'\*{difference[i]:.5f}'

        cur\_coeffs \*= (X\_[0] - x[i])

    return polynom\_str, polynom\_X\_

def generate\_linspace(start, end, num\_points):

    step = (end - start) / (num\_points - 1)

    return [start + i \* step for i in range(num\_points)]

def plot\_interpolation(x, y, func, lagrange\_func, newton\_func, title, x\_star):

    x\_plot = generate\_linspace(min(x)-0.5, max(x)+0.5, 500)

    y\_exact = []

    y\_lagrange = []

    y\_newton = []

    for xi in x\_plot:

        y\_exact.append(func(xi))

        \_, yi\_l = lagrange\_func(x, y, (xi, 0))

        \_, yi\_n = newton\_func(x, y, (xi, 0))

        y\_lagrange.append(yi\_l)

        y\_newton.append(yi\_n)

    plt.figure()

    plt.plot(x\_plot, y\_exact, label='y = ctg(x)+x', linewidth=2)

    plt.plot(x\_plot, y\_lagrange, '--', label='Лагранж')

    plt.plot(x\_plot, y\_newton, ':', label='Ньютон')

    plt.scatter(x, y, color='red', label='Точки')

    plt.scatter(x\_star, [func(x\_star)], color='green', label='X\*', zorder=5)

    plt.title(title)

    plt.xlabel('x')

    plt.ylabel('y')

    plt.legend()

    plt.grid(True)

    plt.show()

if \_\_name\_\_ == '\_\_main\_\_':

    # Вариант а

    x\_a = [math.pi/8, 2\*math.pi/8, 3\*math.pi/8, 4\*math.pi/8]

    y\_a = [f(xi) for xi in x\_a]

    # Вариант б

    x\_b = [math.pi/8, math.pi/3, 3\*math.pi/8, math.pi/2]

    y\_b = [f(xi) for xi in x\_b]

    X\_ = 3\*math.pi/16

    Y\_ = f(X\_)

    print('-----------------Интерполяционный многочлен Лагранжа----------------')

    print('----Вариант а----')

    lagrange\_polynom\_a, lagrange\_value\_a = lagrange\_interpolation(x\_a, y\_a, (X\_, 0))

    print(lagrange\_polynom\_a)

    print(f'y(X\*) = {Y\_:.5f}')

    print(f'L(X\*) = {lagrange\_value\_a:.5f}')

    print(f'Погрешность = {abs(Y\_ - lagrange\_value\_a)}')

    print('----Вариант б----')

    lagrange\_polynom\_b, lagrange\_value\_b = lagrange\_interpolation(x\_b, y\_b, (X\_, 0))

    print(lagrange\_polynom\_b)

    print(f'y(X\*) = {Y\_:.5f}')

    print(f'L(X\*) = {lagrange\_value\_b:.5f}')

    print(f'Погрешность = {abs(Y\_ - lagrange\_value\_b)}')

    print('\n-----------------Интерполяционный многочлен Ньютона-----------------')

    print('----Вариант а----')

    newton\_polynom\_a, newton\_value\_a = newton\_interpolation(x\_a, y\_a, (X\_, 0))

    print(newton\_polynom\_a)

    print(f'y(X\*) = {Y\_:.5f}')

    print(f'P(X\*) = {newton\_value\_a:.5f}')

    print(f'Погрешность = {abs(Y\_ - newton\_value\_a)}')

    print('----Вариант б----')

    newton\_polynom\_b, newton\_value\_b = newton\_interpolation(x\_b, y\_b, (X\_, 0))

    print(newton\_polynom\_b)

    print(f'y(X\*) = {Y\_:.5f}')

    print(f'P(X\*) = {newton\_value\_b:.5f}')

    print(f'Погрешность = {abs(Y\_ - newton\_value\_b)}')

    plot\_interpolation(x\_a, y\_a, f, lagrange\_interpolation, newton\_interpolation, 'Вариант а', X\_)

    plot\_interpolation(x\_b, y\_b, f, lagrange\_interpolation, newton\_interpolation, 'Вариант б', X\_)

Результат:

-----------------Интерполяционный многочлен Лагранжа----------------

----Вариант а----

L(x) = + -7.72499\*(x-0.79)(x-1.18)(x-1.57) + 14.74095\*(x-0.39)(x-1.18)(x-1.57) + -13.14674\*(x-0.39)(x-0.79)(x-1.57) + 4.32304\*(x-0.39)(x-0.79)(x-1.18)

y(X\*) = 2.08565

L(X\*) = 2.15155

Погрешность = 0.06589423733451127

----Вариант б----

L(x) = + -4.63499\*(x-1.05)(x-1.18)(x-1.57) + 36.21484\*(x-0.39)(x-1.18)(x-1.57) + -39.44023\*(x-0.39)(x-1.05)(x-1.57) + 6.48456\*(x-0.39)(x-1.05)(x-1.18)

y(X\*) = 2.08565

L(X\*) = 2.20059

Погрешность = 0.11493837206783475

-----------------Интерполяционный многочлен Ньютона-----------------

----Вариант а----

P(x) = 2.80691 + (x-0.39)\*-2.60127 + (x-0.39)(x-0.79)\*2.68599 + (x-0.39)(x-0.79)(x-1.18)\*-1.80775

y(X\*) = 2.08565

P(X\*) = 2.15155

Погрешность = 0.06589423733451127

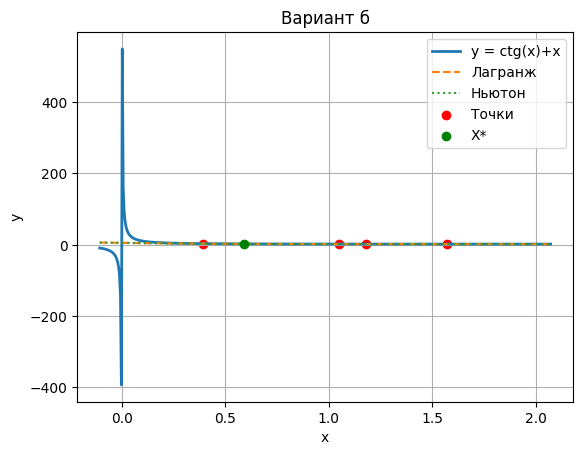
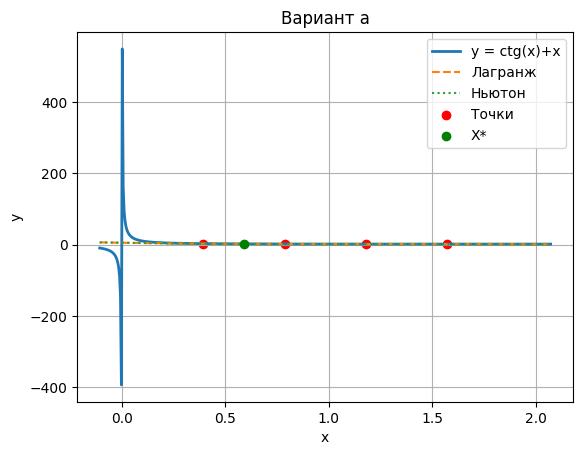
----Вариант б----

P(x) = 2.80691 + (x-0.39)\*-1.80652 + (x-0.39)(x-1.05)\*1.98657 + (x-0.39)(x-1.05)(x-1.18)\*-1.37583

y(X\*) = 2.08565

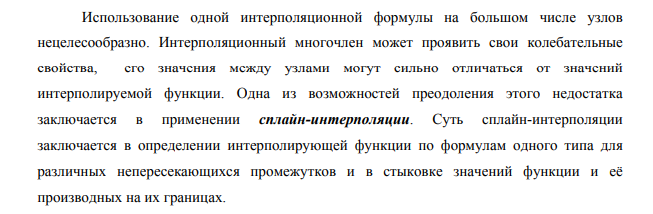
P(X\*) = 2.20059

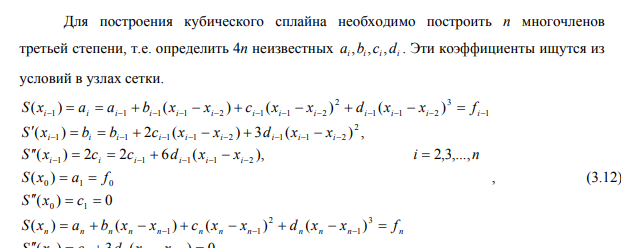
Погрешность = 0.1149383720678343



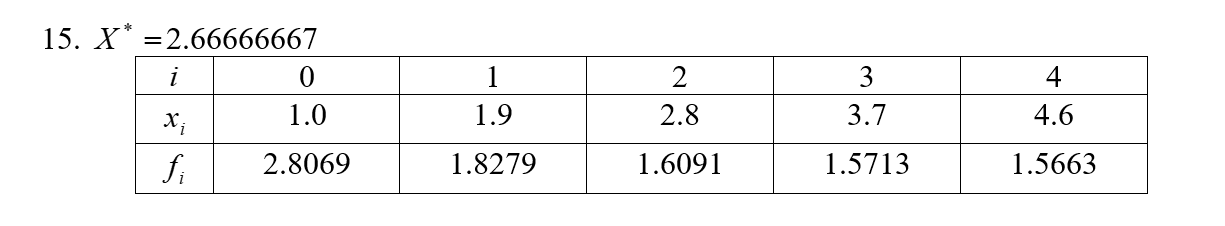
## 3.2 Сплайны

Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при  и . Вычислить значение функции в точке .





Условие:



Код программы:

import matplotlib.pyplot as plt

def compute\_spline\_coeffs(x, f):

    n = len(x)

    h = [x[i+1] - x[i] for i in range(n-1)]

    # Правая часть системы

    alpha = [0.0] \* n

    for i in range(2, n):

        alpha[i] = 3 \* ((f[i] - f[i-1]) / h[i-1] - (f[i-1] - f[i-2]) / h[i-2])

    l = [0.0] \* n ## изменённая главная диагональ

    mu = [0.0] \* n ## вспомогательные коэффициенты для обратного хода

    z = [0.0] \* n ## модифицированная правая часть

    l[1] = 1.0

    mu[1] = 0.0

    z[1] = 0.0

    # Прямой ход прогонки

    for i in range(2, n):

        l[i] = 2 \* (h[i-2] + h[i-1]) - h[i-2] \* mu[i-1]

        mu[i] = h[i-1] / l[i]

        z[i] = (alpha[i] - h[i-2] \* z[i-1]) / l[i]

    l[n-1] = 1.0

    z[n-1] = 0.0

    a = f[:-1]

    c = [0.0] \* n

    b = [0.0] \* (n-1)

    d = [0.0] \* (n-1)

    for j in range(n-2, 0, -1):

        c[j] = z[j] - mu[j] \* c[j+1]

    c[0] = 0.0

    c[n-1] = 0.0

    for j in range(1, n):

        b[j-1] = (f[j] - f[j-1]) / h[j-1] - (h[j-1] \* (c[j] + 2\*c[j-1])) / 3

        d[j-1] = (c[j] - c[j-1]) / (3\*h[j-1])

    return a, b, c, d

def evaluate\_spline(x, a, b, c, d, X\_):

    for i in range(len(x)-1):

        if x[i] <= X\_ <= x[i+1]:

            return a[i] + b[i]\*(X\_ - x[i]) + c[i]\*(X\_ - x[i])\*\*2 + d[i]\*(X\_ - x[i])\*\*3

    print("Точка X\* вне сплайна")

    return None

def plot\_spline(x, f, a, b, c, d, X\_=None, f\_star=None):

    xs = []

    ys = []

    step = 0.01

    cur = x[0]

    while cur <= x[-1]:

        xs.append(cur)

        ys.append(evaluate\_spline(x, a, b, c, d, cur))

        cur += step

    plt.figure(figsize=(8,6))

    plt.plot(x, f, 'o', label='Исходные узлы')

    plt.plot(xs, ys, '-', label='Кубический сплайн')

    if X\_ is not None and f\_star is not None:

        plt.plot(X\_, f\_star, 's', markersize=8, label=f'f({X\_:.2f}) = {f\_star:.5f}')

    plt.grid(True)

    plt.title('Кубический сплайн интерполяции')

    plt.xlabel('x')

    plt.ylabel('f(x)')

    plt.legend()

    plt.show()

x = [1.0, 1.9, 2.8, 3.7, 4.6]

f = [2.8069, 1.8279, 1.6091, 1.5713, 1.5663]

X\_ = 2.66666667  # X\*

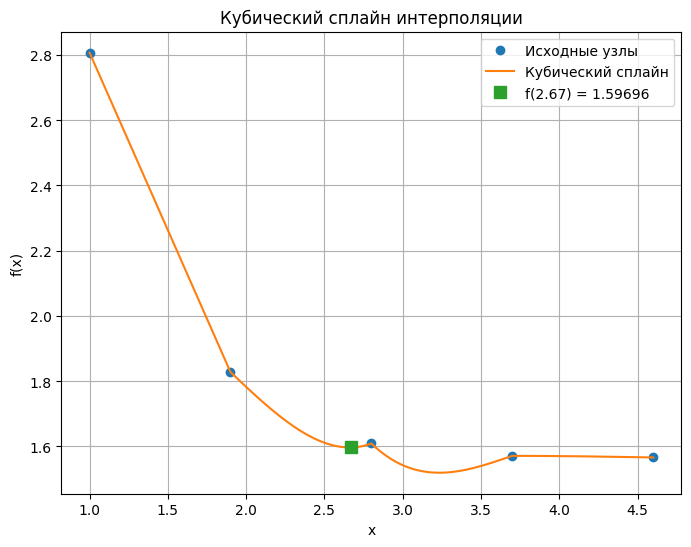
a, b, c, d = compute\_spline\_coeffs(x, f)

F\_ = evaluate\_spline(x, a, b, c, d, X\_)

print(f"f(X\*) = {F\_:.6f}")

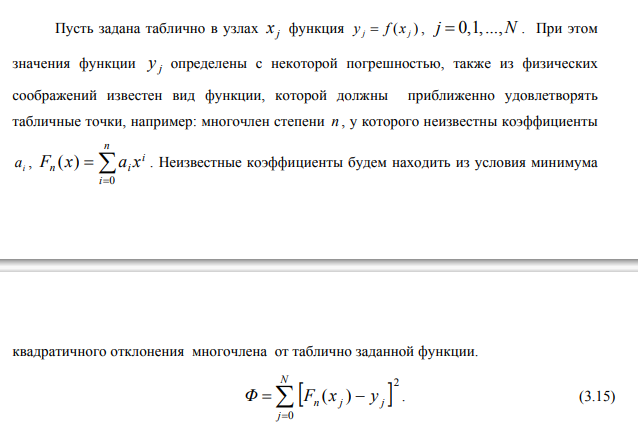
plot\_spline(x, f, a, b, c, d, X\_, F\_)

Результат:

f(X\*) = 1.596958

## 3.3 Метод наименьших квадратов

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



Условие:



Код программы:

import matplotlib.pyplot as plt

def gauss\_method(A, B):

    A = [row[:] for row in A]

    B = B[:]

    n = len(B)

    ## Нормализуем построчно

    for i in range(n):

        max = A[i][i]

        for j in range(i, n):

            A[i][j] /= max

        B[i] /= max

        ## приводим к верхнетреугольной матрице A, обнуляя столбцы

        for k in range(i + 1, n):

            factor = A[k][i]

            for j in range(i, n):

                A[k][j] -= factor \* A[i][j]

            B[k] -= factor \* B[i]

    X = [0] \* n

    for i in range(n - 1, -1, -1):

        X[i] = B[i]

        for j in range(i + 1, n):

            X[i] -= A[i][j] \* X[j]

    return X

def get\_linear\_coeffs(x, y):

    n = len(x)

    sx = sum(x)

    sx2 = sum(xi \*\* 2 for xi in x)

    sy = sum(y)

    sxy = sum(x[i] \* y[i] for i in range(n))

    A = [[n, sx],

         [sx, sx2]]

    B = [sy, sxy]

    return gauss\_method(A, B)

def get\_square\_coeffs(x, y):

    n = len(x)

    sx = sum(x)

    sx2 = sum(xi \*\* 2 for xi in x)

    sx3 = sum(xi \*\* 3 for xi in x)

    sx4 = sum(xi \*\* 4 for xi in x)

    sy = sum(y)

    sxy = sum(x[i] \* y[i] for i in range(n))

    sx2y = sum((x[i] \*\* 2) \* y[i] for i in range(n))

    A = [[n, sx, sx2],

         [sx, sx2, sx3],

         [sx2, sx3, sx4]]

    B = [sy, sxy, sx2y]

    return gauss\_method(A, B)

x = [1.0, 1.9, 2.8, 3.7, 4.6, 5.5]

y = [3.4142, 2.9818, 3.3095, 3.8184, 4.3599, 4.8318]

a0, a1 = get\_linear\_coeffs(x, y)

b0, b1, b2 = get\_square\_coeffs(x, y)

def f\_lin(x\_): return a0 + a1 \* x\_

def f\_sq(x\_):  return b0 + b1 \* x\_ + b2 \* x\_ \* x\_

print(f"Многочлен 1-й степени: f(x) = {a0:.5f} + {a1:.5f} \* x")

print(f"Многочлен 2-й степени: f(x) = {b0:.5f} + {b1:.5f} \* x + {b2:.5f} \* x^2")

error1 = sum((f\_lin(x[i]) - y[i]) \*\* 2 for i in range(len(x)))

error2 = sum((f\_sq(x[i]) - y[i]) \*\* 2 for i in range(len(x)))

print(f"Сумма квадратов ошибок для многочлена 1-й степени: {error1:.5f}")

print(f"Сумма квадратов ошибок для многочлена 2-й степени: {error2:.5f}")

# построим кривые вблизи диапазона узлов

lo, hi = min(x) - 0.2, max(x) + 0.2

x\_val = [lo + i \* (hi - lo) / 300 for i in range(301)]

y\_val1 = [f\_lin(xx) for xx in x\_val]

y\_val2 = [f\_sq(xx) for xx in x\_val]

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))

plt.scatter(x, y, color='black', label='Табличные данные', zorder=5)

plt.plot(x\_val, y\_val1, label='Многочлен 1-й степени', linewidth=2)

plt.plot(x\_val, y\_val2, label='Многочлен 2-й степени', linestyle='--', linewidth=2)

plt.xlabel('x'); plt.ylabel('y'); plt.title('Аппроксимация МНК')

plt.grid(True); plt.legend(); plt.show()

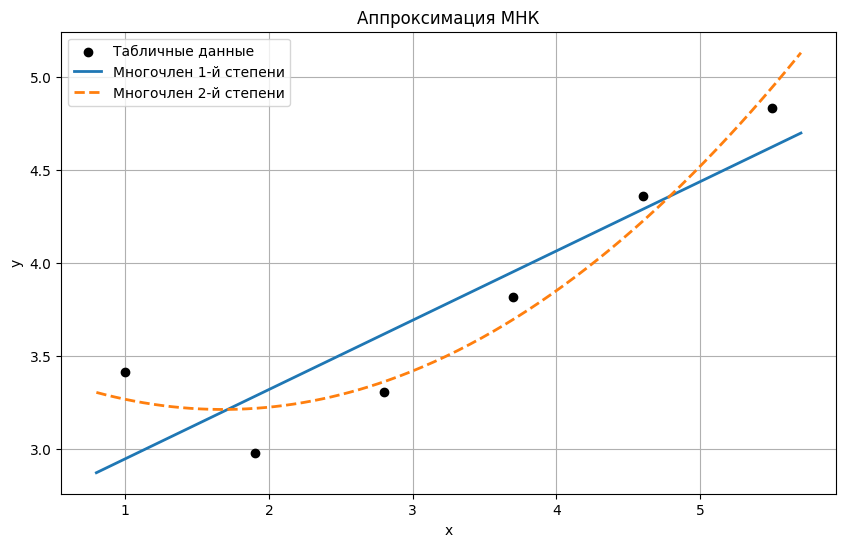
Результат:

Многочлен 1-й степени: f(x) = 2.57557 + 0.37242 \* x

Многочлен 2-й степени: f(x) = 3.54755 + -0.39805 \* x + 0.11853 \* x^2

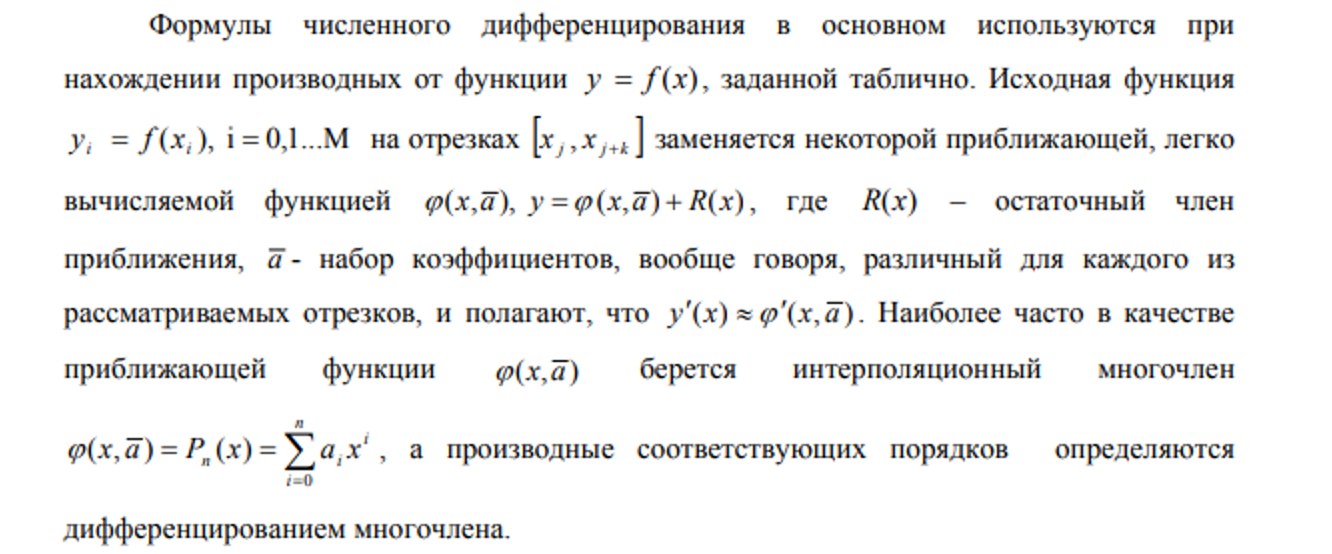
Сумма квадратов ошибок для многочлена 1-й степени: 0.47012

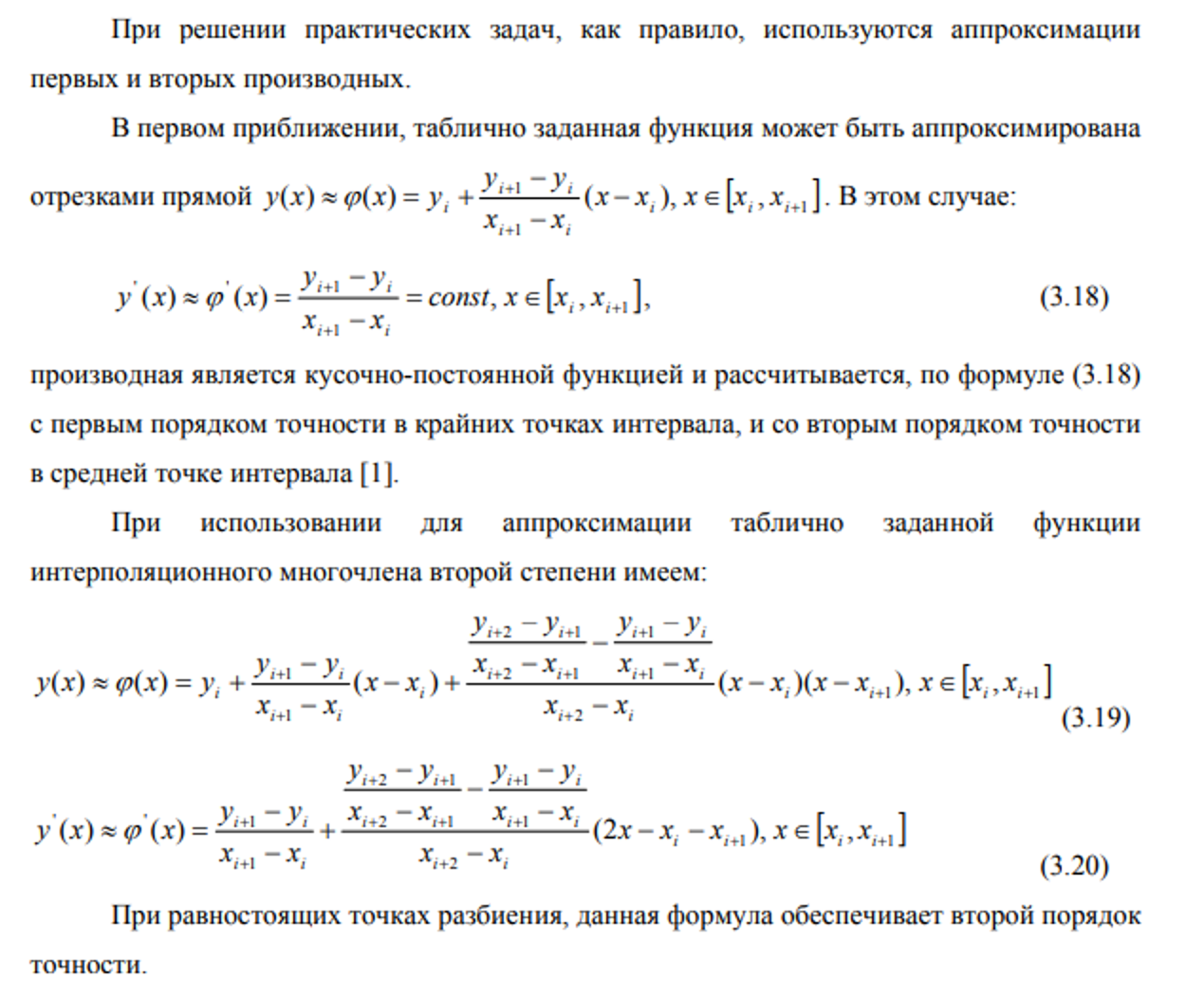
Сумма квадратов ошибок для многочлена 2-й степени: 0.12597

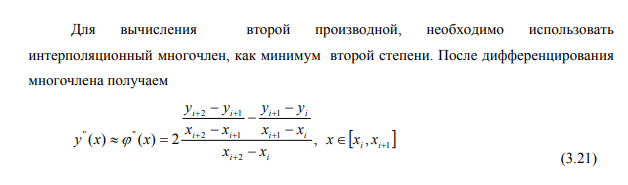


## 3.4 Численное дифференцирование

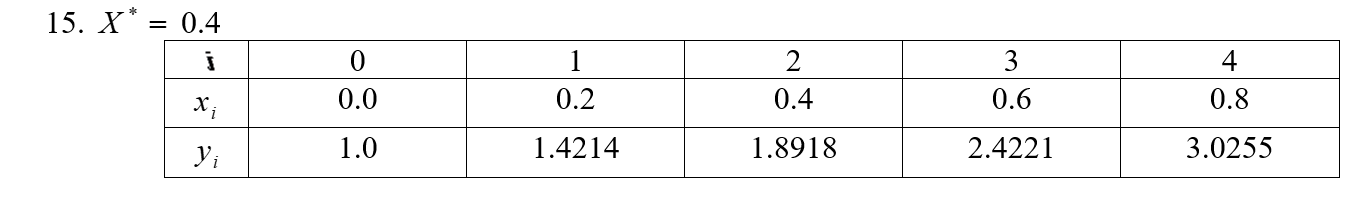
Вычислить первую и вторую производную от таблично заданной функции в точке .







Условие:



Код программы:

def get\_index(x, x0):

    for i in range(len(x)):

        if abs(x[i] - x0) < 1e-8:

            return i

    return -1

def first\_derivative(x, y, index\_x):

    if 0 < index\_x < len(x) - 1:

        print("Точка находится внутри отрезка: ")

        dx\_left = (y[index\_x] - y[index\_x - 1]) / (x[index\_x] - x[index\_x - 1])

        dx\_right = (y[index\_x + 1] - y[index\_x]) / (x[index\_x + 1] - x[index\_x])

        print(f"  Левосторонняя производная: {dx\_left:.6f}")

        print(f"  Правосторонняя производная: {dx\_right:.6f}")

        diff\_dy = (dx\_right - dx\_left) / (x[index\_x + 1] - x[index\_x - 1])

        derivative = dx\_left + diff\_dy \* (2 \* x[index\_x] - x[index\_x - 1] - x[index\_x])

        print(f"  Первая производная: {derivative:.6f} \n")

    elif index\_x == 0:

        print("Точка находится на левом крае:")

        dx\_right = (y[index\_x + 1] - y[index\_x]) / (x[index\_x + 1] - x[index\_x])

        print(f"  Правая производная: {dx\_right:.6f} \n")

    elif index\_x == len(x) - 1:

        print("Точка находится на правом крае: ")

        dx\_left = (y[index\_x] - y[index\_x - 1]) / (x[index\_x] - x[index\_x - 1])

        print(f"  Левая производная: {dx\_left:.6f} \n")

def second\_derivative\_standard(x, y, index\_x):

    if 0 < index\_x < len(x) - 1:

        h = x[index\_x + 1] - x[index\_x]  # шаг (у нас он равен 0.2)

        dx2 = (y[index\_x - 1] - 2 \* y[index\_x] + y[index\_x + 1]) / (h \*\* 2)

        print(f"Вторая производная (центральная): {dx2:.6f}")

    else:

        print("Вторая производная не определена на краях.")

def second\_derivative(x, y, index\_x):

    if 0 < index\_x < len(x) - 1:

        dx2 = 2 \* ((y[index\_x + 1] - y[index\_x]) / (x[index\_x + 1] - x[index\_x]) -

                   (y[index\_x] - y[index\_x - 1]) / (x[index\_x] - x[index\_x - 1])) / \

              (x[index\_x + 1] - x[index\_x - 1])

        print(f"Вторая производная: {dx2:.6f}")

    else:

        print("Вторая производная не определена на краях.")

# --- данные из таблицы варианта 15 ---

x  = [0.0, 0.2, 0.4, 0.6, 0.8]

y  = [1.0, 1.4214, 1.8918, 2.4221, 3.0255]

x0 = 0.4  # X\*

index\_x = get\_index(x, x0)

if index\_x != -1:

    first\_derivative(x, y, index\_x)

    second\_derivative\_standard(x, y, index\_x)

    second\_derivative(x, y, index\_x)

else:

    print("Точка не найдена в таблице")

Результат:

Точка находится внутри отрезка:

Левосторонняя производная: 2.352000

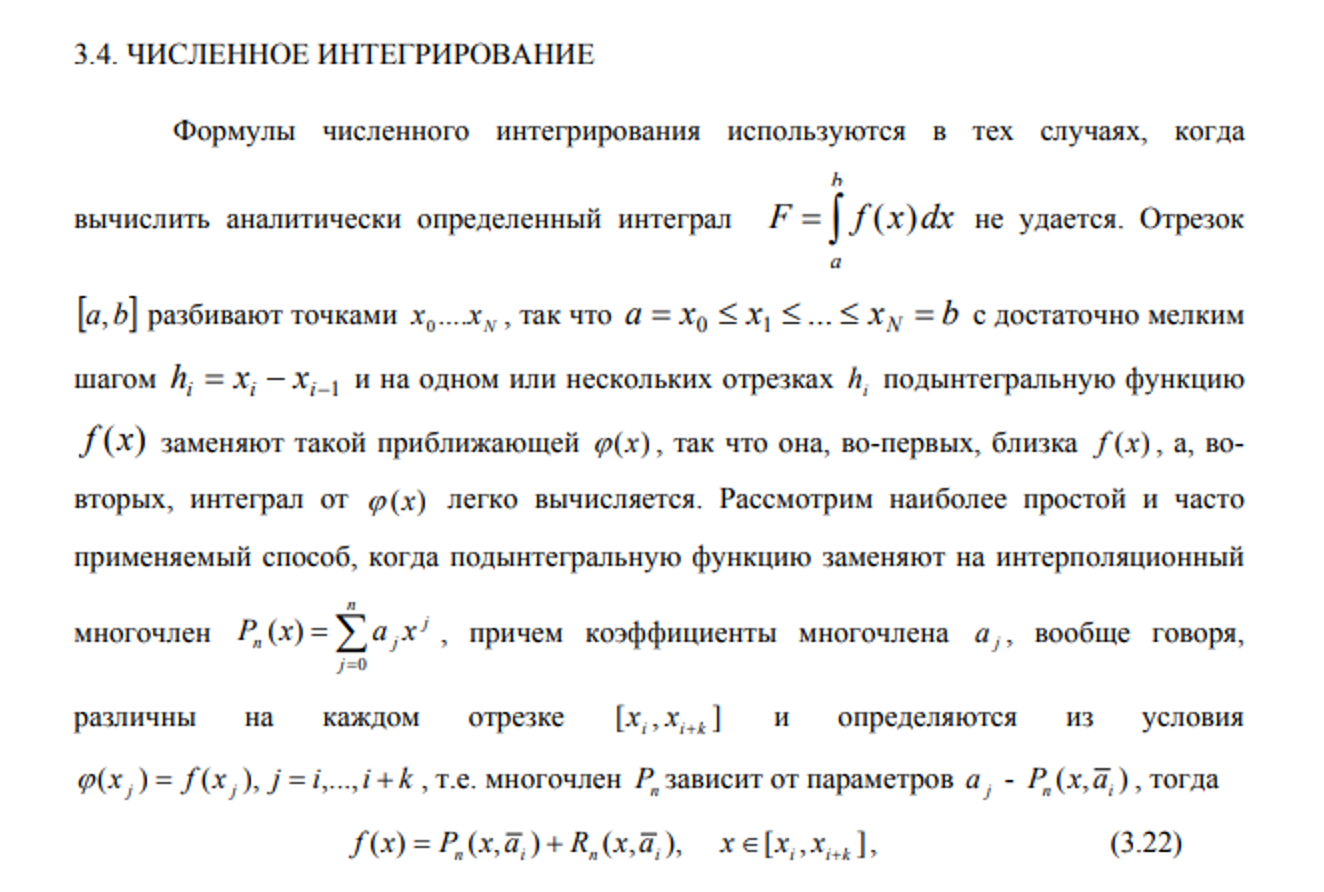
Правосторонняя производная: 2.651500

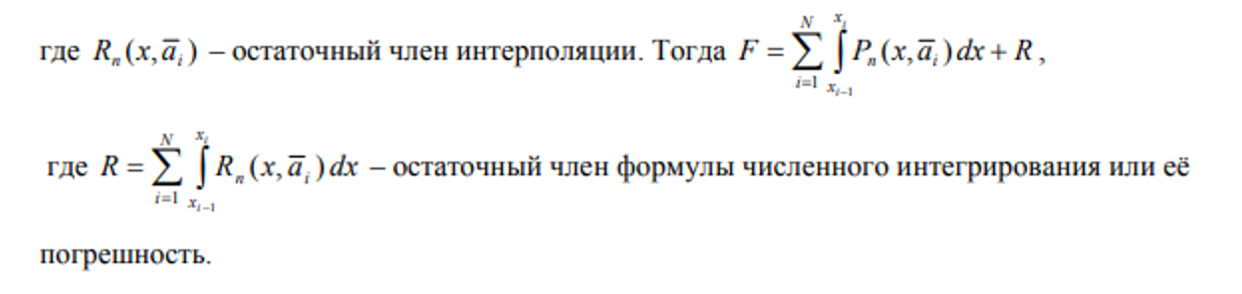
Первая производная: 2.501750

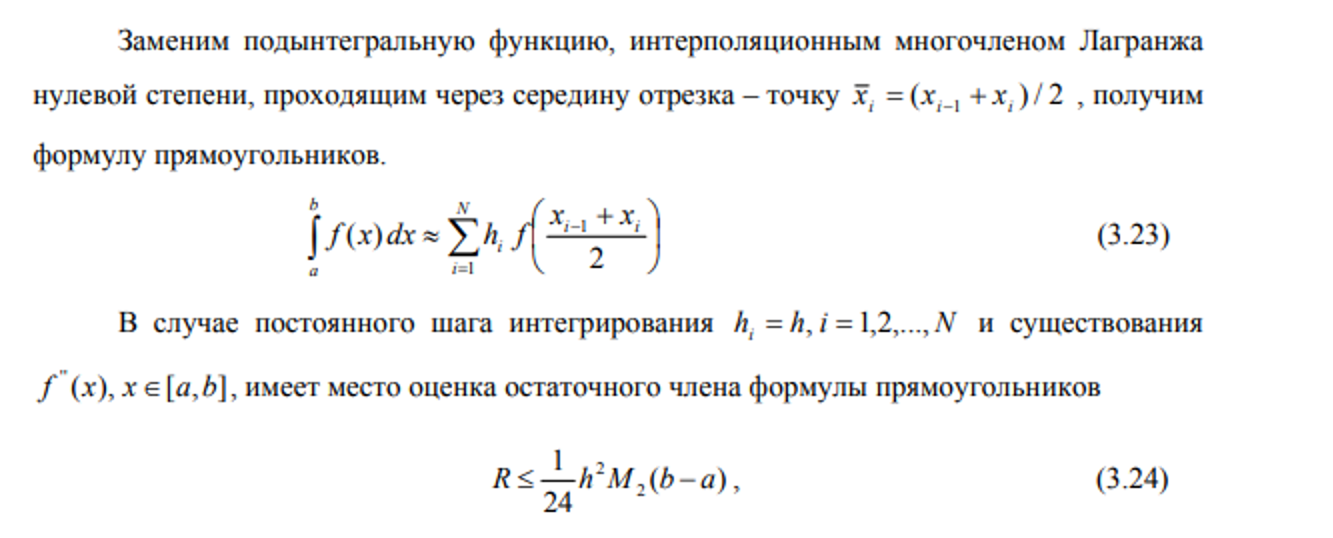
Вторая производная (центральная): 1.497500

Вторая производная: 1.497500

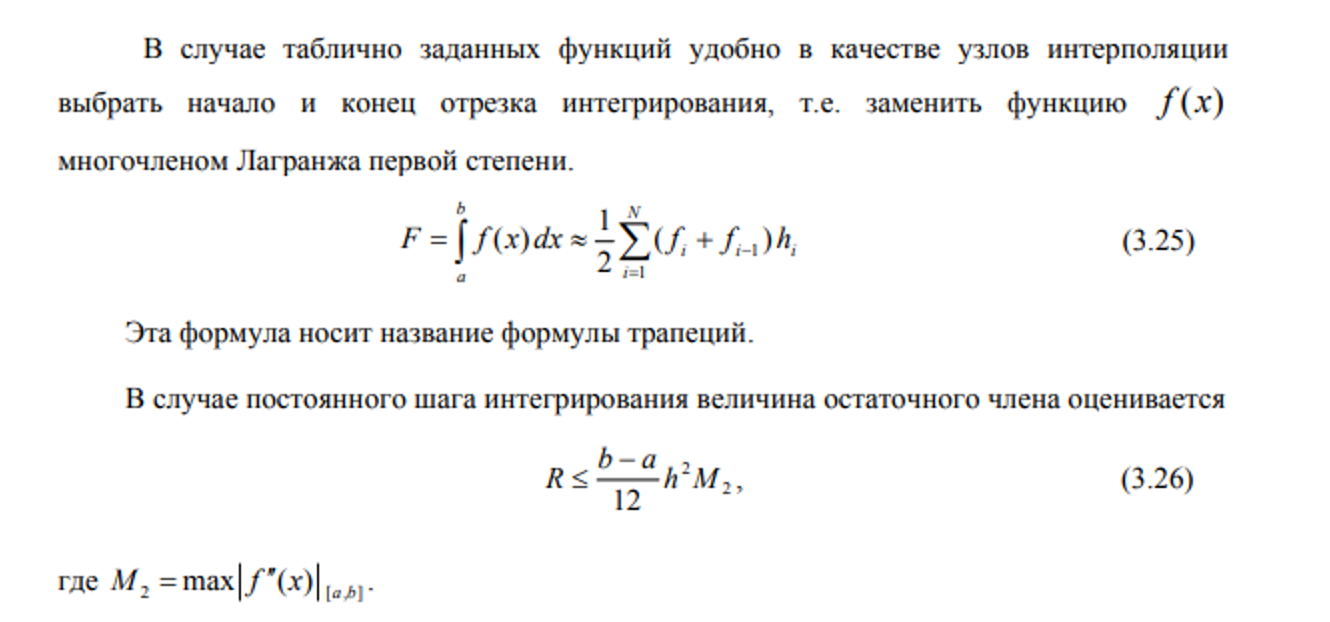
## 3.5 Численное интегрирование

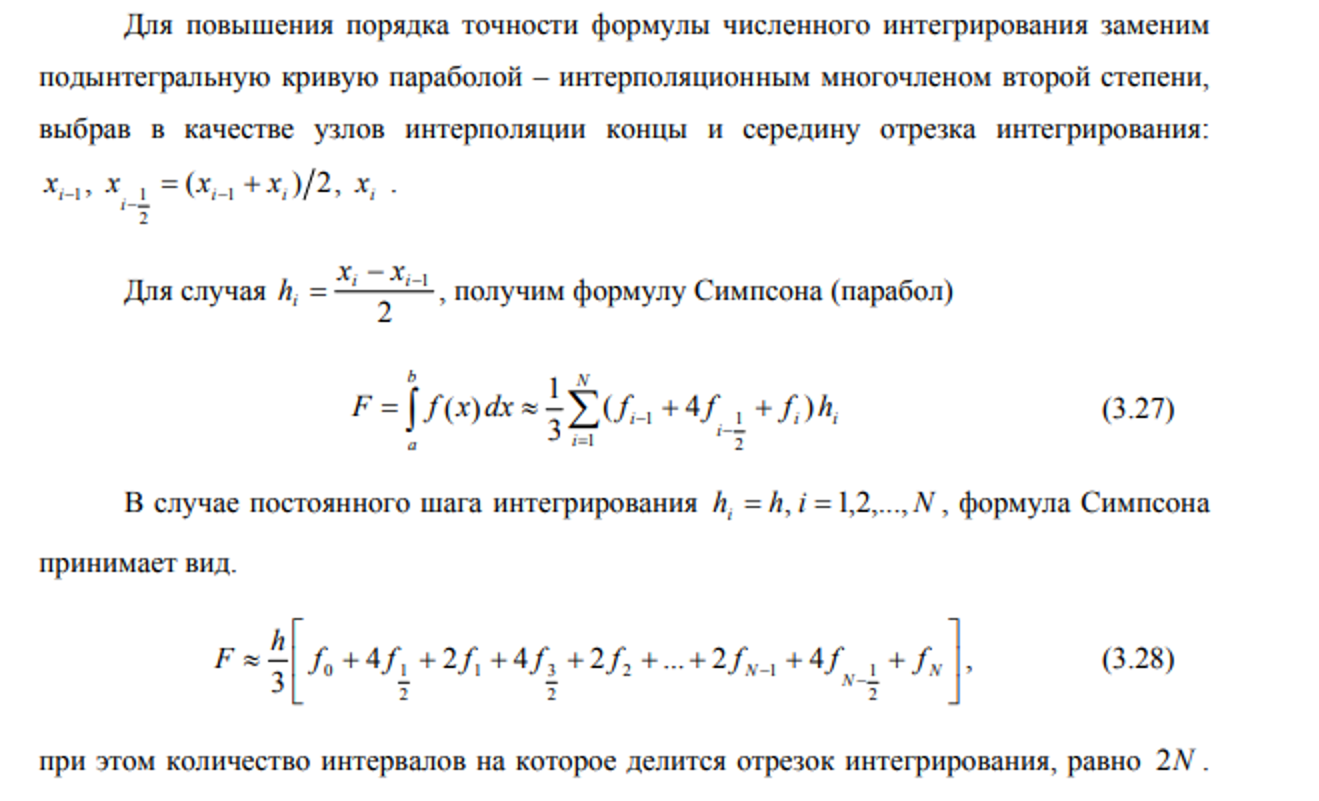
Вычислить определенный интеграл , методами прямоугольников, трапеций, Симпсона с шагами . Оценить погрешность вычислений, используя Ме­тод Рунге-Ромберга: 

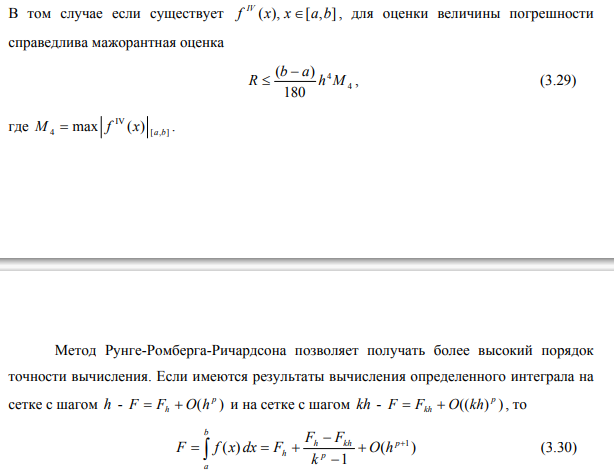












Условие:



Код программы:

def f(x):

    return x / (x\*\*4 + 81)   # y = x/(x^4+81)

# Метод прямоугольников (серединных)

def rectangle\_method(f, a, b, h):

    n = int((b - a) / h)

    s = 0.0

    for i in range(n):

        x\_mid = (a + i\*h) + h/2

        s += f(x\_mid)

    return s \* h

def trapezoidal\_method(f, a, b, h):

    n = int((b - a) / h)

    s = (f(a) + f(b)) / 2

    for i in range(1, n):

        xi = a + i\*h

        s += f(xi)

    return s \* h

def simpson\_method(f, a, b, h):

    n = int((b - a) / h)

    if n % 2 != 0:

        n += 1

        h = (b - a) / n

    s = f(a) + f(b)

    for i in range(1, n):

        xi = a + i\*h

        s += (4 if i % 2 else 2) \* f(xi)

    return s \* h / 3

def runge\_romberg\_corrected(I1, I2, p):

    return I2 + (I2 - I1) / (2\*\*p - 1)

def runge\_romberg(I1, I2, p):

    return abs(I2 - I1) / (2\*\*p - 1)

def main():

    a, b = 0.0, 2.0

    h1, h2 = 0.5, 0.25

    I\_rect\_h1 = rectangle\_method(f, a, b, h1)

    I\_rect\_h2 = rectangle\_method(f, a, b, h2)

    rect\_corr  = runge\_romberg\_corrected(I\_rect\_h1, I\_rect\_h2, 2)

    err\_rect   = runge\_romberg(I\_rect\_h1, I\_rect\_h2, 2)

    I\_trap\_h1 = trapezoidal\_method(f, a, b, h1)

    I\_trap\_h2 = trapezoidal\_method(f, a, b, h2)

    trap\_corr  = runge\_romberg\_corrected(I\_trap\_h1, I\_trap\_h2, 2)

    err\_trap   = runge\_romberg(I\_trap\_h1, I\_trap\_h2, 2)

    I\_simp\_h1 = simpson\_method(f, a, b, h1)

    I\_simp\_h2 = simpson\_method(f, a, b, h2)

    simp\_corr  = runge\_romberg\_corrected(I\_simp\_h1, I\_simp\_h2, 4)

    err\_simp   = runge\_romberg(I\_simp\_h1, I\_simp\_h2, 4)

    print(f"Прямоугольники: I(h1)={I\_rect\_h1:.6f}, I(h2)={I\_rect\_h2:.6f}, уточнённое={rect\_corr:.6f}, погр. RR={err\_rect:.6f}")

    print(f"Трапеции:       I(h1)={I\_trap\_h1:.6f}, I(h2)={I\_trap\_h2:.6f}, уточнённое={trap\_corr:.6f}, погр. RR={err\_trap:.6f}")

    print(f"Симпсон:        I(h1)={I\_simp\_h1:.6f}, I(h2)={I\_simp\_h2:.6f}, уточнённое={simp\_corr:.6f}, погр. RR={err\_simp:.6f}")

if \_\_name\_\_ == '\_\_main\_\_':

    main()

Результат:

Прямоугольники: I(h1)=0.023326, I(h2)=0.023258, уточнённое=0.023235, погр. RR=0.000023

Трапеции: I(h1)=0.023051, I(h2)=0.023189, уточнённое=0.023235, погр. RR=0.000046

Симпсон: I(h1)=0.023233, I(h2)=0.023235, уточнённое=0.023235, погр. RR=0.000000

# **4 Лабораторная работа**