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.

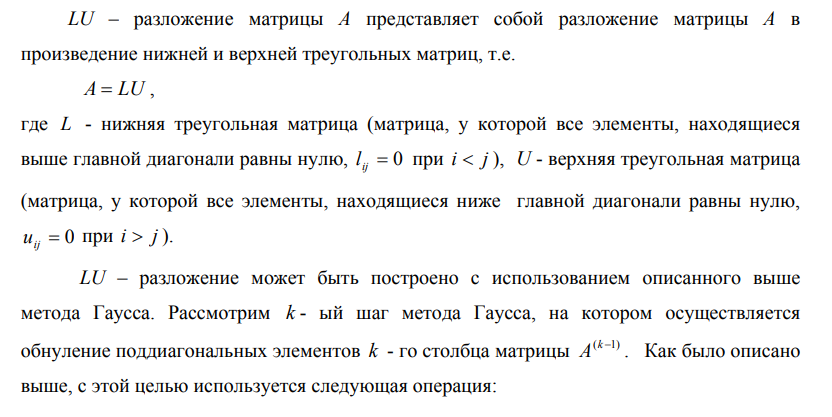
Группа: М8О-312Б-22

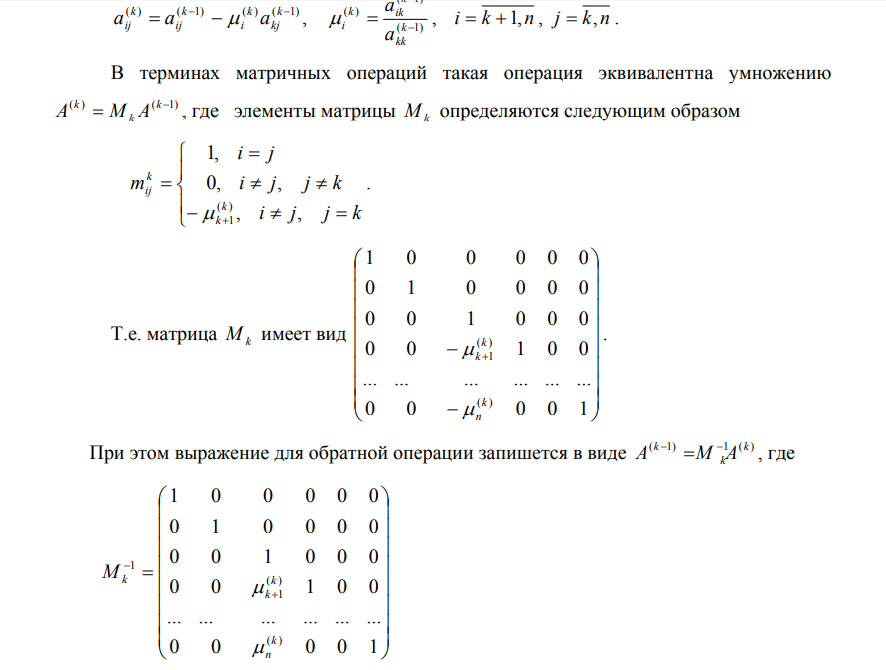
Руководитель: Демидова О. Л.

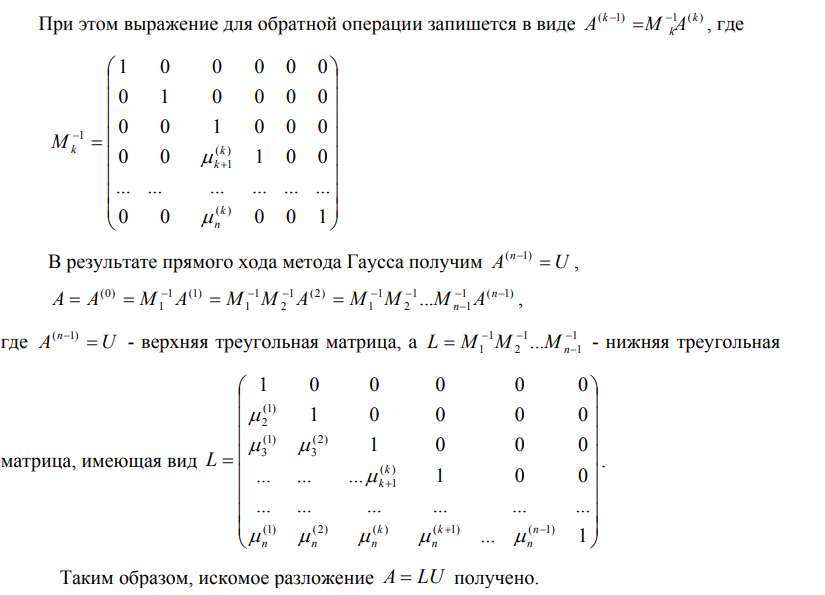
Москва 2025

Лабораторная работа №1

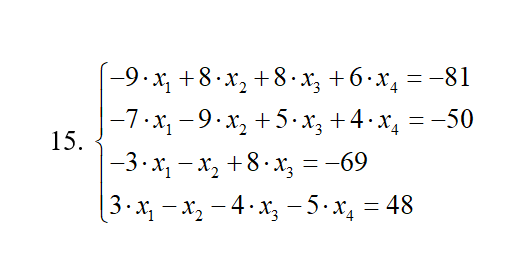
* 1. Реализовать алгоритм LU - разложения матриц (с выбором главного элемента) в виде программы. Используя разработанное программное обеспечение, решить систему линейных алгебраических уравнений (СЛАУ). Для матрицы СЛАУ вычислить определитель и обратную матрицу.







Условие:



Код программы:

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 determinant(A):

    n = len(A)

    det = 1

    for i in range(n):

        det \*= A[i][i]

    return det

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 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 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]

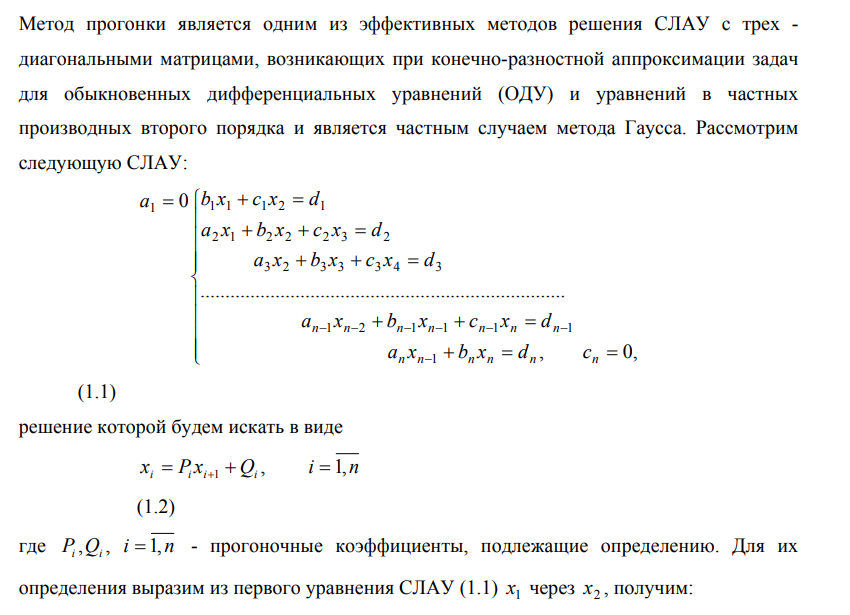
[0.05732018985031034, -0.06133625410733845, -0.009127418765972981, 0.019715224534501648]

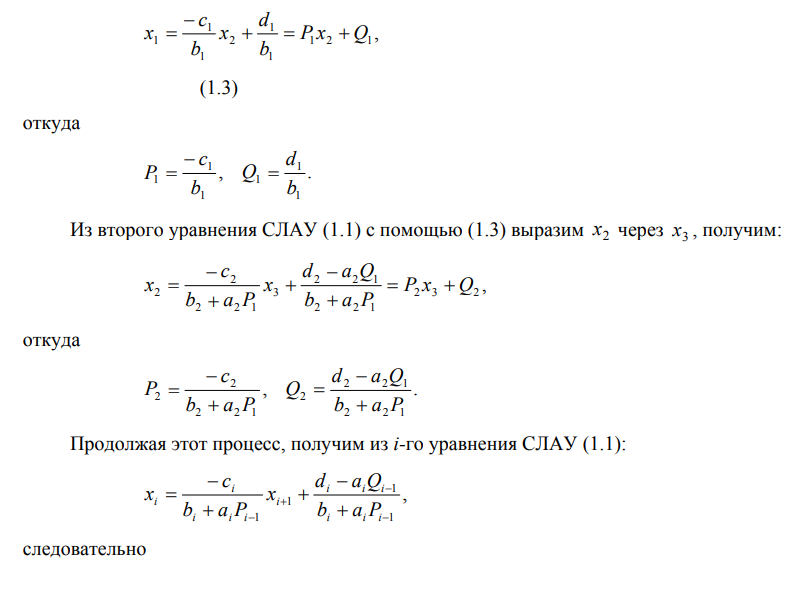
[-0.04527199707922598, -0.04709748083242059, 0.15370573201898502, -0.09200438116100766]

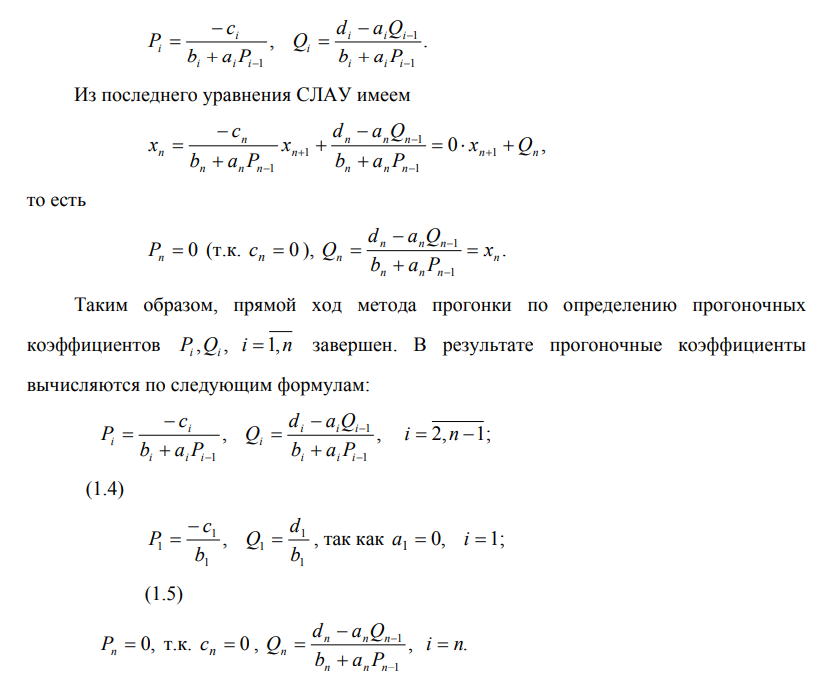
[-0.059145673603504915, -0.01314348302300109, -0.07338444687842278, -0.28148959474260676]

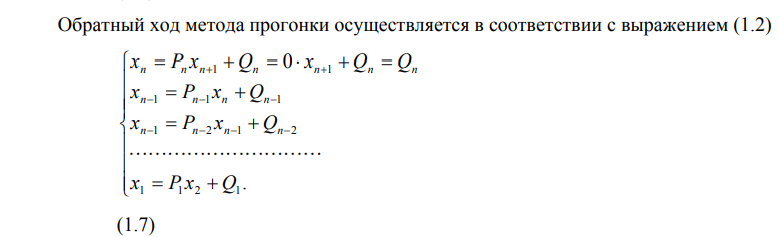
Проверка решения (Ax = b): True

* 1. 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.

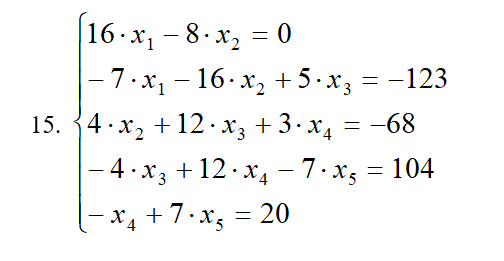








Условие:



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

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]

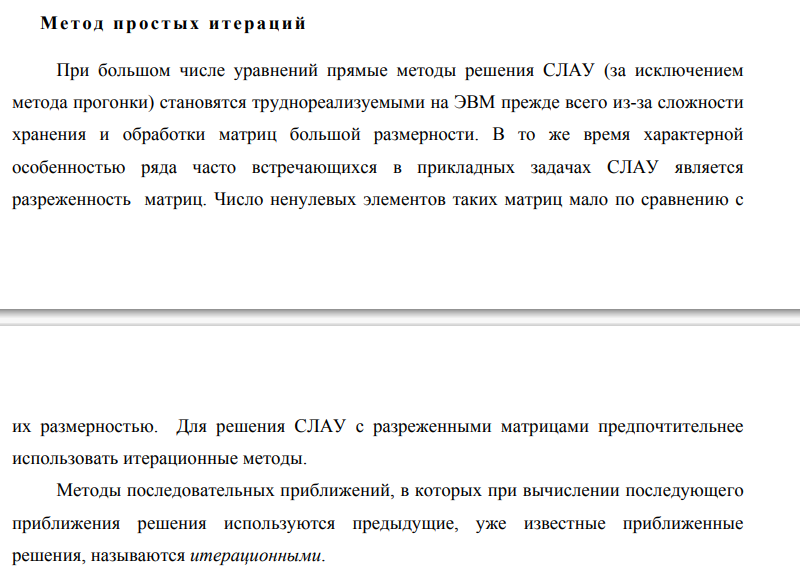
solution = run\_through\_algorithm(A, B)

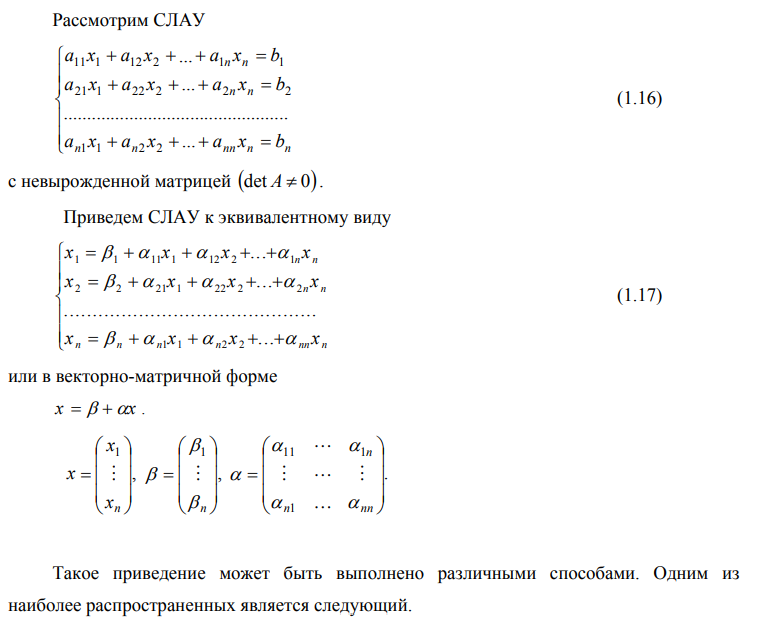
print("Рещение СЛАУ методом прогонки :", solution)

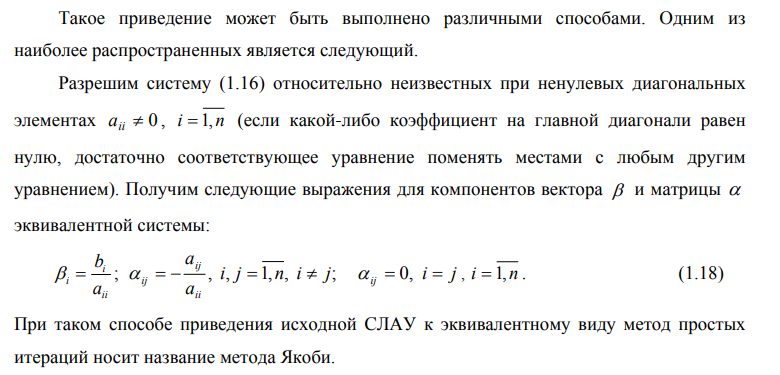
Результат:

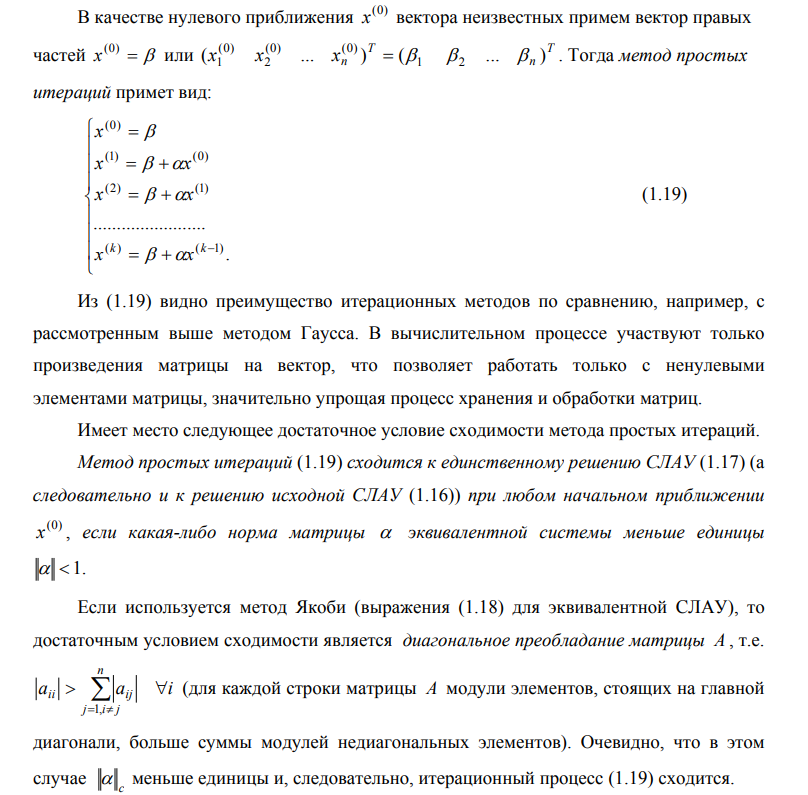
Рещение СЛАУ методом прогонки: [2.0, 4.0, -9.0, 8.0, 4.0]

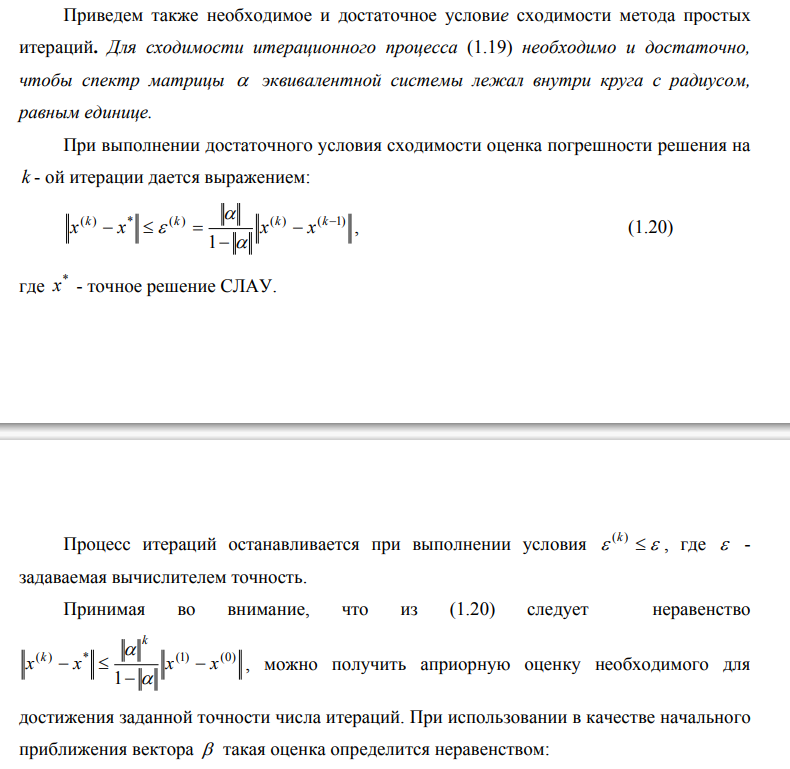
* 1. Реализовать метод простых итераций и метод Зейделя в виде программ, задавая в качестве входных данных матрицу системы, вектор правых частей и точность вычислений. Используя разработанное программное обеспечение, решить СЛАУ. Проанализировать количество итераций, необходимое для достижения заданной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

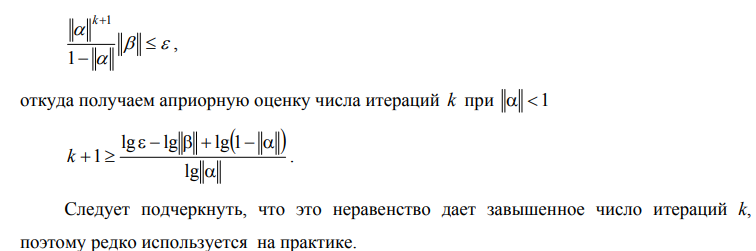




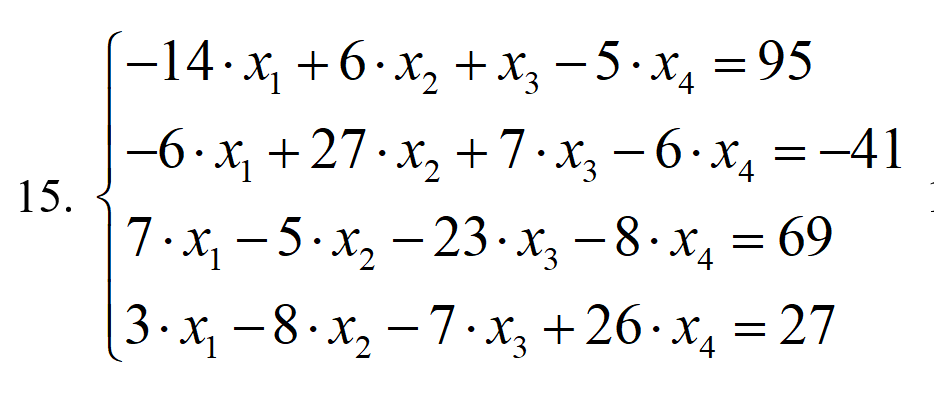








Условие:



Код программы:

def check\_matrix(A):

    n = len(A)

    flag = False

    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)

        if diagonal <= row\_sum:

            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)

                if diagonal <= column\_sum:

                    return False

                elif diagonal > column\_sum:

                    flag = True

        elif diagonal > row\_sum:

            flag = True

    return flag

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

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

    n = len(b)

    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()

    iteration = 0

    iteration\_history = []

    residual\_history = []

    while iteration < max\_iterations:

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

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

        iteration\_history.append(iteration + 1)

        residual\_history.append(residual)

        if residual <= tolerance:

            # Проверка, что A \* X ≈ b вручную

            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")

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

        X0 = X.copy()

        iteration += 1

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

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()

    iteration = 0

    iteration\_history = []

    residual\_history = []

    while iteration < max\_iterations:

        for i in range(n):

            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))

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

        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)

        iteration\_history.append(iteration + 1)

        residual\_history.append(max\_residual)

        if max\_residual < tolerance:

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

        iteration += 1

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

# Система уравнений

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]

print("Матрица A:", A)

print("Вектор B:", B)

# Метод простых итераций

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

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)

print("Метод простых итераций:")

print("Решение:", solution\_s\_i)

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

# Метод Зейделя

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

print("\nМетод Зейделя:")

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

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

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]

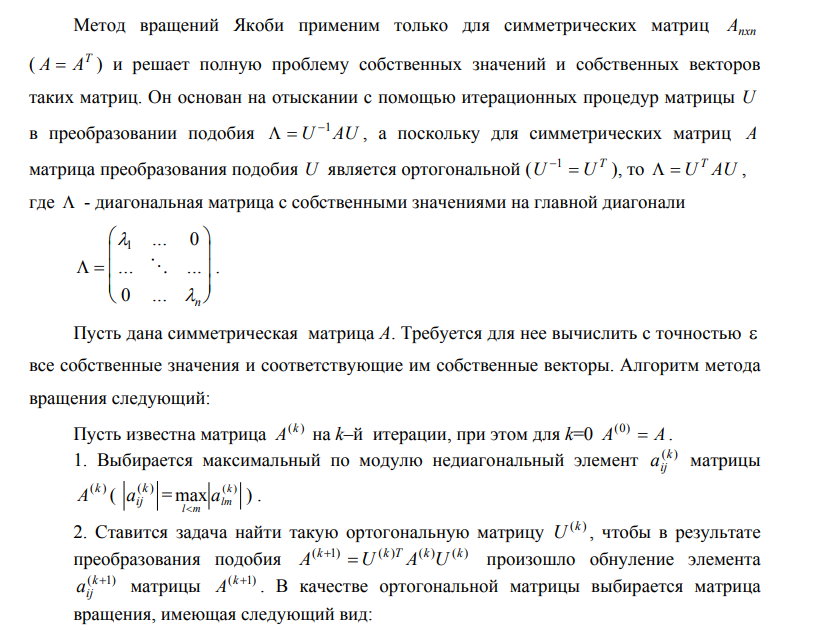
Количество итераций: 48

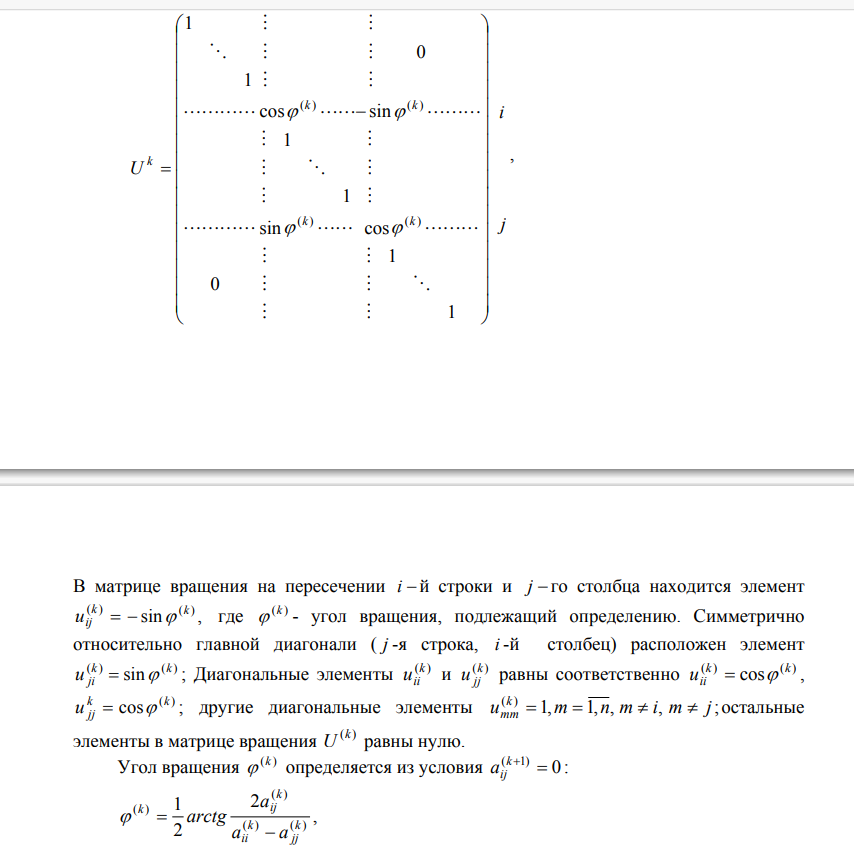
Метод Зейделя:

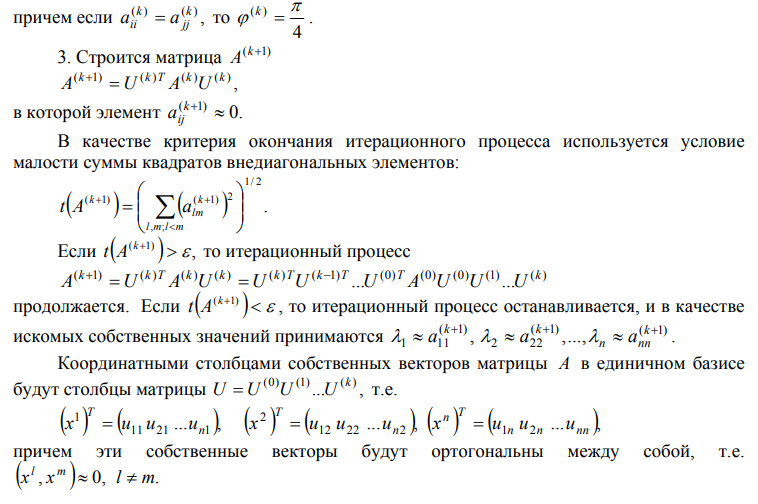
Решение: [-7.999999999996591, -1.9999999999993985, -4.999999999999361, -3.634699379080512e-14]

Количество итераций: 17

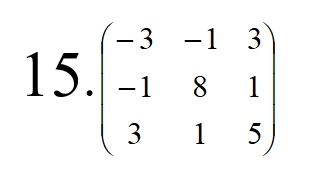
* 1. Реализовать метод вращений в виде программы, задавая в качестве входных данных матрицу и точность вычислений. Используя разработанное программное обеспечение, найти собственные значения и собственные векторы симметрических матриц. Проанализировать зависимость погрешности вычислений от числа итераций.







Условие:



Код программы:

import math

def create\_eye\_matrix(n):

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

def transpose(matrix):

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

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

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

    """Восстанавливает матрицу A из собственных значений и векторов."""

    n = len(eigen\_values)

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

    for i in range(n):

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

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

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

    return A\_reconstructed

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

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

    n = len(A)

    eigenvectors = create\_eye\_matrix(n)

    iteration = 0

    iterations = []

    while iteration < max\_iter:

        max\_off\_diag = 0

        p, q = 0, 0

        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)

        if max\_off\_diag < precision:

            break

        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)

        s = math.sin(phi)

        R = create\_eye\_matrix(n)

        R[p][p] = c

        R[p][q] = -s

        R[q][p] = s

        R[q][q] = c

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

        eigenvectors = matrix\_multiply(eigenvectors, R)

        iteration += 1

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

    return eigenvalues, eigenvectors, iterations

# === Основной блок ===

try:

    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])

    if not check\_symmetry(A):

        raise ValueError("Матрица не является симметричной.")

    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)}")

    # Восстановим матрицу A по собственным значениям и векторам

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

    print("\Проверка A:")

    for row in A\_reconstructed:

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

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

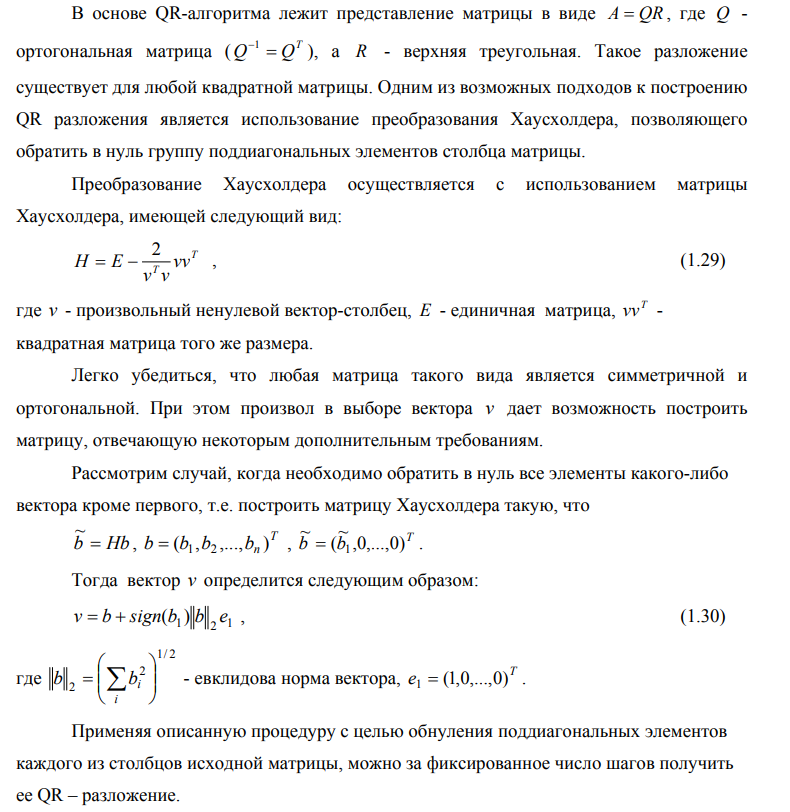
\Проверка A:

[-3.0, -1.0, 3.0]

[-1.0, 8.0, 1.0]

[3.0, 1.0, 5.0]

* 1. 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:

def multiply\_matrix(A, B):

    if len(A[0]) != len(B):

        raise ValueError("Невозможно умножить матрицы такого размера")

    rows\_A = len(A)

    cols\_B = len(B[0])

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

    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

def transpose\_matrix(A):

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

def identity\_matrix(n):

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

def qr\_decomposition(A):

    n = len(A)

    Q = identity\_matrix(n)

    A\_k = [row.copy() for row in A]

    for i in range(n - 1):

        v = [0] \* n

        sum\_sq = 0

        for j in range(i, n):

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

        norm = sum\_sq \*\* 0.5

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

        v[i] = A\_k[i][i] + sign \* norm

        for j in range(i + 1, n):

            v[j] = A\_k[j][i]

        v\_transpose = [[x] for x in v]

        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))

        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)

    return Q, A\_k

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

    A\_k = [row.copy() for row in A]

    for \_ in range(max\_iter):

        converged = True

        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

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

        A\_k = multiply\_matrix(R, Q)

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

    return eigenvalues

matrix = [

    [1, 7, -1],

    [-2, 2, -2],

    [9, -7, 3]

]

eigenvalues = calculate\_eigenvalues(matrix)

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

Результат:

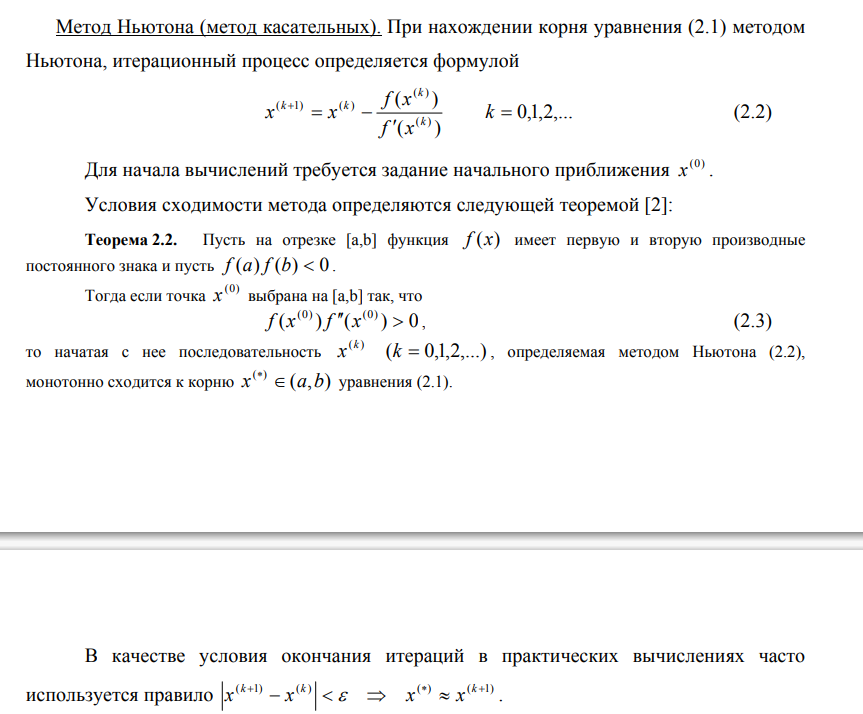
Полученные собственные значения: [1.6475929460379108, 6.622176049434543, -2.269768995472492]

Лабораторная работа №2

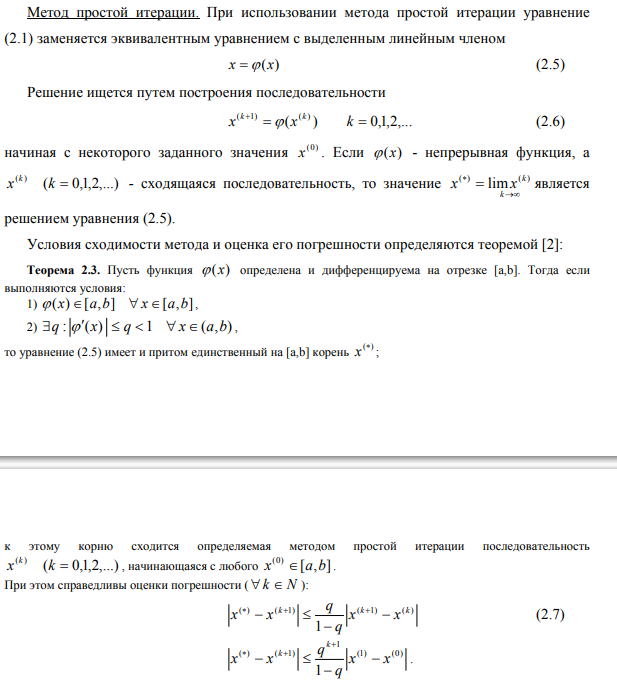
2.1. Реализовать методы простой итерации и Ньютона решения нелинейных уравнений в виде программ, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения найти положительный корень нелинейного уравнения (начальное приближение определить графически).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

Условие:

****1) Метод Ньютона



2)Метод простой итерации



Код программы:

import numpy as np

import matplotlib.pyplot as plt

# Построение графика функции

x = np.linspace(0, 2, 400)

y = np.sin(x) - x\*\*2 + 1

# Создаем окно с двумя графиками

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

# График функции

ax1.plot(x, y, label='sin(x) - x² + 1', color='blue')

ax1.axhline(0, color='black', linewidth=0.5)

ax1.grid(True)

ax1.legend()

ax1.set\_title('График функции sin(x) - x² + 1')

ax1.set\_xlabel('x')

ax1.set\_ylabel('y')

# Метод простой итерации

def simple\_iteration(phi, phi\_deriv, x0, tol=1e-6, max\_iter=1000):

    # Проверка условия сходимости |φ'(x)| < 1 в окрестности начального приближения

    if abs(phi\_deriv(x0)) >= 1:

        raise ValueError("Условие сходимости метода простой итерации не выполнено: |φ'(x)| >= 1")

    x = x0

    errors = []

    for i in range(max\_iter):

        x\_new = phi(x)

        error = abs(x\_new - x)

        errors.append(error)

        if error < tol:

            return x\_new, errors

        x = x\_new

    return x, errors

# Метод Ньютона

def newton\_method(f, f\_deriv, f\_double\_deriv, x0, tol=1e-6, max\_iter=1000):

    # Проверка условия сходимости: f'(x) != 0 и f''(x) существует

    if f\_deriv(x0) == 0:

        raise ValueError("Условие сходимости метода Ньютона не выполнено: f'(x) = 0")

    x = x0

    errors = []

    for i in range(max\_iter):

        fx = f(x)

        if abs(fx) < tol:

            return x, errors

        dx = -fx / f\_deriv(x)

        x\_new = x + dx

        errors.append(abs(dx))

        x = x\_new

    return x, errors

# Определение функций

phi = lambda x: np.sqrt(np.sin(x) + 1)

phi\_deriv = lambda x: 0.5 \* (np.cos(x) / np.sqrt(np.sin(x) + 1))

f = lambda x: np.sin(x) - x\*\*2 + 1

f\_deriv = lambda x: np.cos(x) - 2 \* x

f\_double\_deriv = lambda x: -np.sin(x) - 2

# Начальное приближение из графика

x0 = 1.4

# Решение методом простой итерации

try:

    solution\_si, errors\_si = simple\_iteration(phi, phi\_deriv, x0, tol=1e-6)

    print(f"Метод простой итерации: корень = {solution\_si:.6f}, итераций = {len(errors\_si)}")

except ValueError as e:

    print(f"Метод простой итерации: {e}")

    solution\_si = None

# Решение методом Ньютона

try:

    solution\_newton, errors\_newton = newton\_method(f, f\_deriv, f\_double\_deriv, x0, tol=1e-6)

    print(f"Метод Ньютона: корень = {solution\_newton:.6f}, итераций = {len(errors\_newton)}")

except ValueError as e:

    print(f"Метод Ньютона: {e}")

    solution\_newton = None

# Отображение найденных корней на графике функции

if solution\_si is not None:

    ax1.scatter(solution\_si, f(solution\_si), color='red', label=f'Корень (итерация): {solution\_si:.6f}')

if solution\_newton is not None:

    ax1.scatter(solution\_newton, f(solution\_newton), color='green', label=f'Корень (Ньютон): {solution\_newton:.6f}')

ax1.legend()

# График зависимости погрешности

if solution\_si is not None and solution\_newton is not None:

    ax2.semilogy(range(1, len(errors\_si)+1), errors\_si, 'o-', label='Простая итерация', color='red')

    ax2.semilogy(range(1, len(errors\_newton)+1), errors\_newton, 's-', label='Ньютона', color='green')

    ax2.set\_xlabel('Номер итерации')

    ax2.set\_ylabel('Погрешность (лог. шкала)')

    ax2.set\_title('Сходимость методов')

    ax2.legend()

    ax2.grid(True)

# Показываем графики

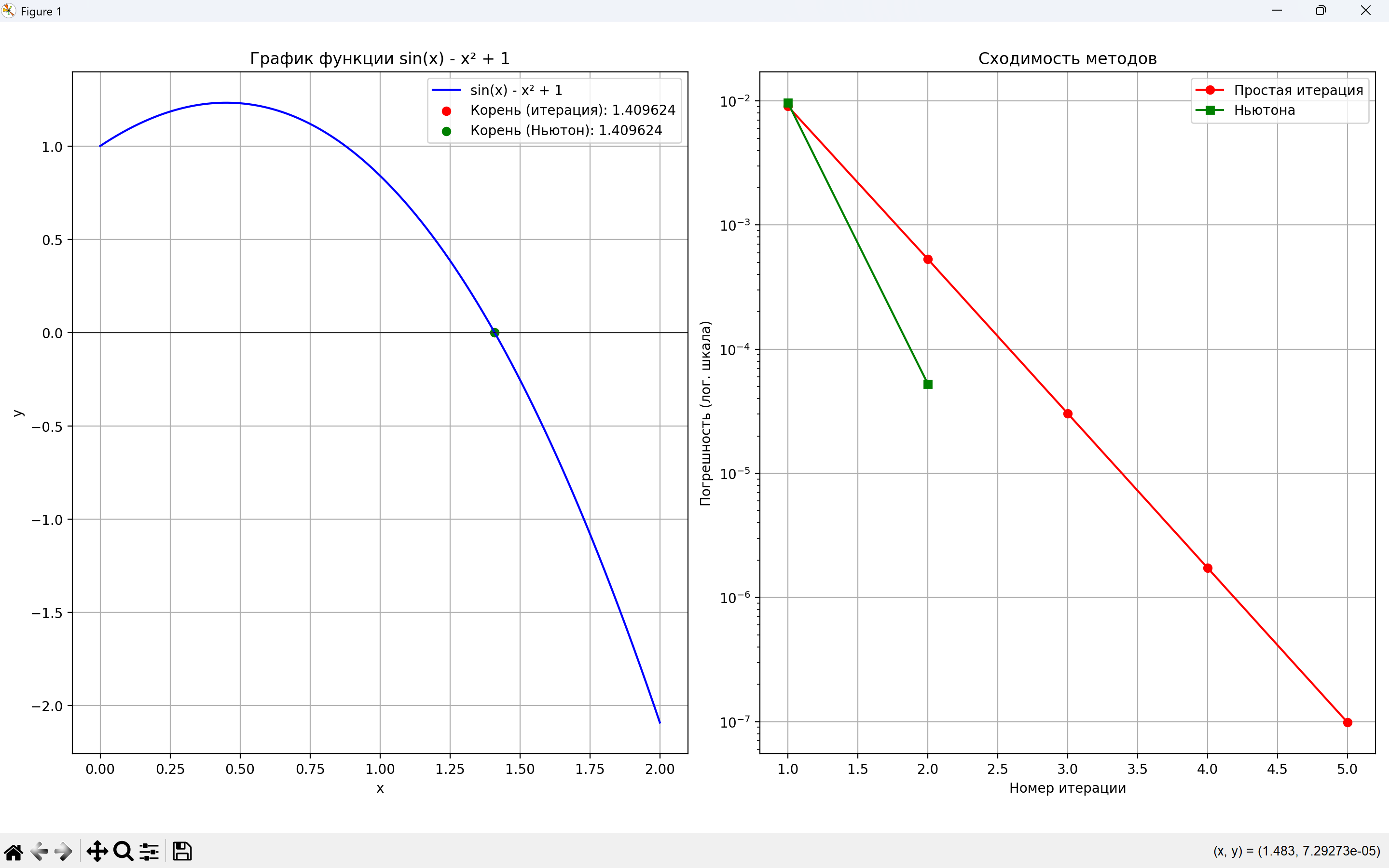
plt.tight\_layout()

plt.show()

Результат:

Метод простой итерации: корень = 1.409624, итераций = 5

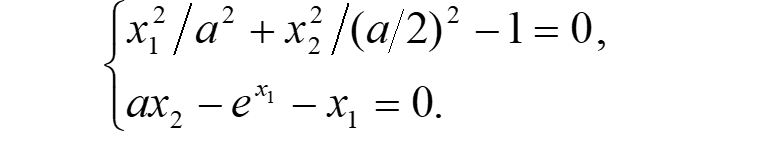
Метод Ньютона: корень = 1.409624, итераций = 2



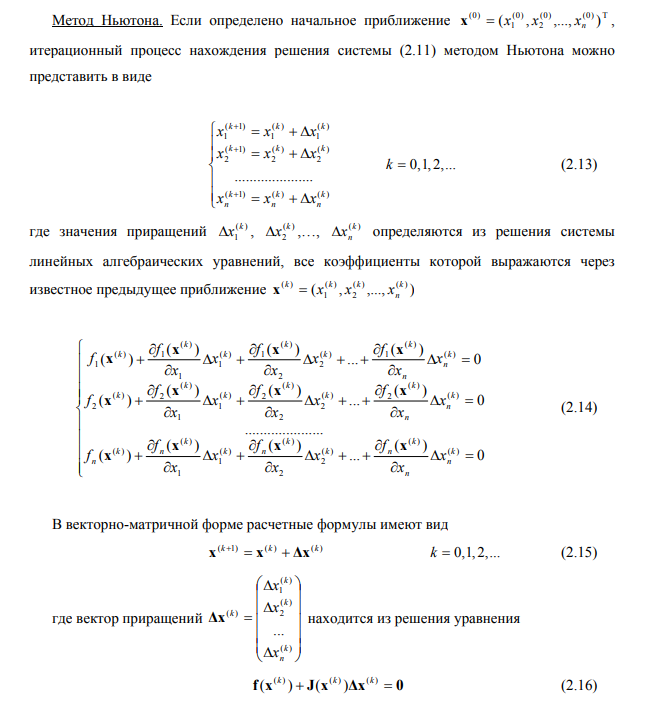
2.2. Реализовать методы простой итерации и Ньютона решения систем нелинейных уравнений в виде программного кода, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения решить систему нелинейных уравнений (при наличии нескольких решений найти то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

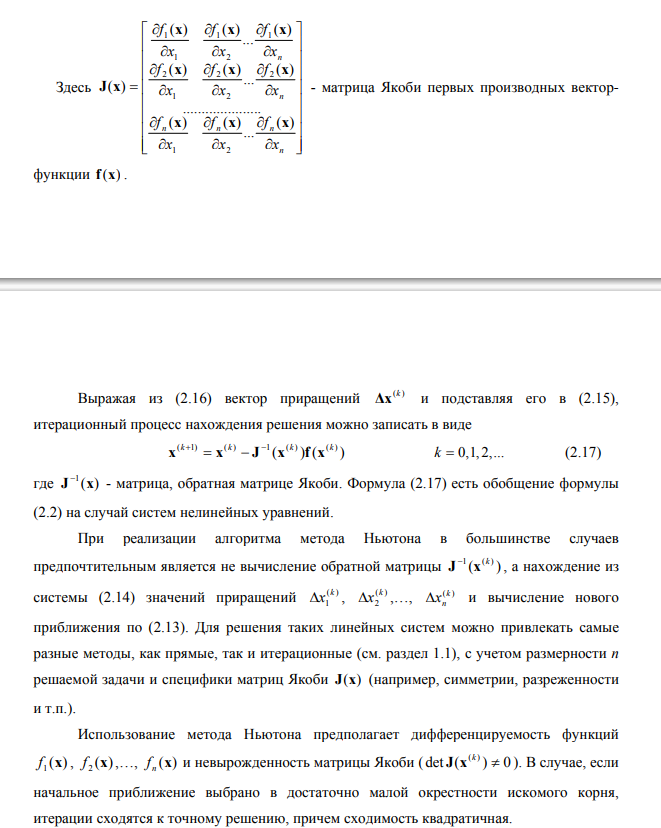
Условие:

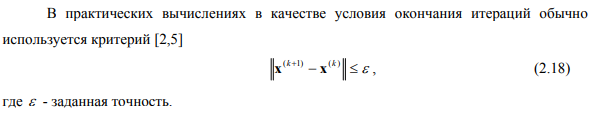
a = 4



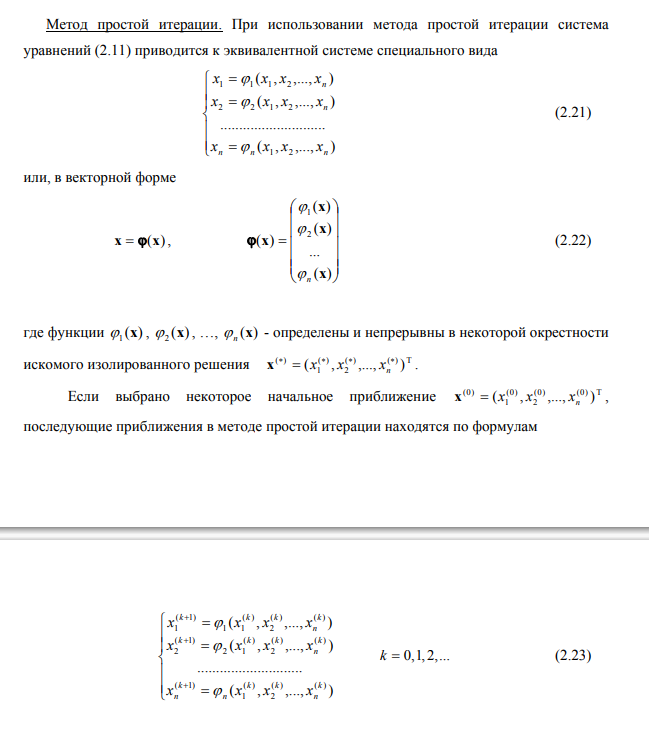
1) Метод Ньютона

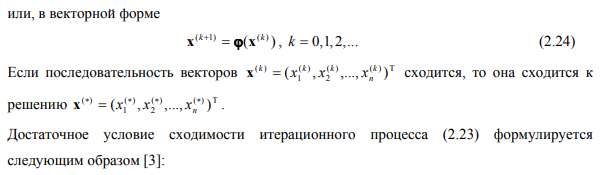






2)Метод простой итерации





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: