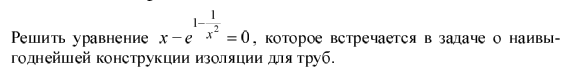
**Выполнил студент: Мачин Артём ИТ-11**

Лабораторная работа №2

Задача 1.



Решение:

import numpy as np

from scipy.optimize import fsolve

def equation(x):

    return x - np.exp(1 / x\*\*2)

x0 = 0.1

solution = fsolve(equation, x0)

print("Решение уравнения:", solution[0])

Ответ: 

Объяснение кода:

# Импорт ядра NumPy под псевдонимом np

# NumPy предоставляет:

# - математические функции (np.exp, np.sin и т.д.)

# - работу с массивами

# - числовые операции

import numpy as np

# Импорт функции fsolve из подмодуля optimize библиотеки SciPy

# fsolve - численный решатель нелинейных уравнений

from scipy.optimize import fsolve

# Определение функции уравнения, которую будем решать

# Уравнение имеет вид: x = e^(1/x²) или f(x) = x - e^(1/x²) = 0

def equation(x):

    """

    Внутренняя работа при вызове:

    1. Принимает x (скаляр или массив)

    2. Вычисляет 1/x² (возведение в степень -2)

    3. Вычисляет экспоненту от результата (np.exp)

    4. Возвращает разность x и экспоненты

    """

    return x - np.exp(1 / x\*\*2)  # x² вычисляется как x\*\*2 в Python

# Начальное приближение для алгоритма

# Важные особенности:

# - x0 не должно быть 0 (деление на 0 в уравнении)

# - При x0 < 0 возможно получение комплексных чисел

# - Рекомендуемый диапазон: 0.1 < x0 < 5

x0 = 0.1  # Эмпирически подобранное начальное значение

# Вызов численного решателя

# Внутренний процесс fsolve:

# 1. Инициализирует итерационный процесс (метод Пауэлла по умолчанию)

# 2. На каждом шаге:

#    a. Вычисляет equation(x\_current)

#    b. Анализирует значение функции

#    c. Корректирует x\_current по алгоритму

# 3. Останавливается при:

#    - |equation(x)| < 1.49e-08 (по умолчанию)

#    - Превышении maxfev (макс. число вызовов функции)

#    - Других критериях сходимости

solution = fsolve(

    equation,  # Функция для решения

    x0,        # Начальное приближение

    # Дополнительные параметры (можно указать):

    # xtol=1e-09 - допустимая погрешность

    # maxfev=100 - макс. итераций

    # full\_output=False - не показывать диагностику

)

# Вывод результата

# solution - это numpy.ndarray (даже для одного решения)

# solution[0] - первый (и единственный) корень

print("Решение уравнения:", solution[0])  # Пример вывода: 1.52513527616

Задача 2. Решить уравнение



Решение:

import scipy.optimize as opt

import numpy as np

def f(x):

    return np.arctan(x - 1) + 2 \* x

solution = opt.fsolve(f, 0)

print("Решение уравнения:", solution[0])

Ответ: 

Объяснение:

# Импорт библиотек

import scipy.optimize as opt  # Импорт всего модуля оптимизации SciPy

import numpy as np           # Импорт библиотеки для математических операций

# Определение функции уравнения

def f(x):

    """

    Функция, представляющая уравнение arctg(x-1) + 2x = 0

    Внутренняя работа при вычислении:

    1. Вычисляет (x - 1) - аргумент арктангенса

    2. Вычисляет np.arctan(x-1) - значение арктангенса

    3. Вычисляет 2\*x - линейный член

    4. Возвращает сумму этих двух компонентов

    """

    return np.arctan(x - 1) + 2 \* x

# Решение уравнения

solution = opt.fsolve(

    f,      # Функция, которую нужно решить

    0,      # Начальное приближение x0 = 0

    # Дополнительные параметры (по умолчанию):

    # xtol=1.49012e-08 - допустимая погрешность

    # maxfev=100\*(n+1) - максимальное число вызовов функции

    # full\_output=0 - не возвращать дополнительную информацию

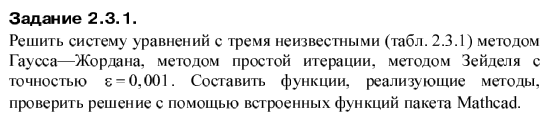
)

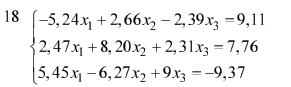
# Вывод результата

print("Решение уравнения:", solution[0])  # solution - массив numpy

Лабораторная работа №3

Задача





Решение:

import numpy as np

from scipy.linalg import solve

A = np.array([[-5.24, 2.66, -2.39],

              [2.47, 8.20, 2.31],

              [5.45, -6.27, 9.00]])

B = np.array([9.11, 7.76, -9.37])

# Метод Гаусса-Жордана

def gauss\_jordan(A, B):

    aug = np.hstack([A, B.reshape(-1, 1)])

    n = len(B)

    for i in range(n):

        aug[i] /= aug[i, i]

        for j in range(n):

            if i != j:

                aug[j] -= aug[i] \* aug[j, i]

    return aug[:, -1]

# Метод простой итерации

def simple\_iteration(A, B, eps=0.001, max\_iter=100):

    n, X = len(B), np.zeros(len(B))

    C, d = np.zeros\_like(A), np.zeros(n)

    for i in range(n):

        d[i] = B[i] / A[i, i]

        for j in range(n):

            if i != j:

                C[i, j] = -A[i, j] / A[i, i]

    for \_ in range(max\_iter):

        X\_new = d + C @ X

        if np.linalg.norm(X\_new - X, np.inf) < eps:

            return X\_new

        X = X\_new

    return X

# Метод Зейделя

def seidel(A, B, eps=0.001, max\_iter=100):

    n, X = len(B), np.zeros(len(B))

    for \_ in range(max\_iter):

        X\_new = np.copy(X)

        for i in range(n):

            X\_new[i] = (B[i] - sum(A[i, j] \* X\_new[j] for j in range(i)) -

                        sum(A[i, j] \* X[j] for j in range(i + 1, n))) / A[i, i]

        if np.linalg.norm(X\_new - X, np.inf) < eps:

            return X\_new

        X = X\_new

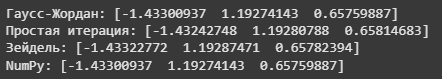
    return X

print("Гаусс-Жордан:", gauss\_jordan(A, B))

print("Простая итерация:", simple\_iteration(A, B))

print("Зейдель:", seidel(A, B))

print("NumPy:", solve(A, B))

Ответ: 

Объяснение:

import numpy as np  # Импорт библиотеки NumPy для работы с массивами

from scipy.linalg import solve  # Импорт функции solve для решения СЛАУ

# Матрица коэффициентов A (3x3)

A = np.array([[-5.24, 2.66, -2.39],

              [2.47, 8.20, 2.31],

              [5.45, -6.27, 9.00]])

# Вектор правых частей B

B = np.array([9.11, 7.76, -9.37])

# Метод Гаусса-Жордана

def gauss\_jordan(A, B):

    # Объединяем матрицу A и вектор B в расширенную матрицу [A | B]

    aug = np.hstack([A, B.reshape(-1, 1)])

    n = len(B)  # Размер системы

    for i in range(n):

        aug[i] /= aug[i, i]  # Нормируем строку: делим всю строку на диагональный элемент, чтобы он стал 1

        for j in range(n):

            if i != j:

                # Обнуляем элементы в остальных строках текущего столбца

                aug[j] -= aug[i] \* aug[j, i]

    return aug[:, -1]  # Возвращаем последний столбец — решение X

# Метод простой итерации (Якоби)

def simple\_iteration(A, B, eps=0.001, max\_iter=100):

    n = len(B)  # Размерность системы

    X = np.zeros(n)  # Начальное приближение: нулевой вектор

    C = np.zeros\_like(A)  # Матрица C для преобразованной системы

    d = np.zeros(n)       # Вектор d для преобразованной системы

    for i in range(n):

        d[i] = B[i] / A[i, i]  # Формируем d[i] как b[i] / a[i][i]

        for j in range(n):

            if i != j:

                # Элементы матрицы C: -a[i][j] / a[i][i]

                C[i, j] = -A[i, j] / A[i, i]

    for \_ in range(max\_iter):  # Итерационный процесс

        X\_new = d + C @ X  # Новое приближение X: X\_new = d + C \* X

        if np.linalg.norm(X\_new - X, np.inf) < eps:  # Проверка на сходимость (по бесконечной норме)

            return X\_new  # Если достаточно точно — возвращаем результат

        X = X\_new  # Обновляем X для следующей итерации

    return X  # Возвращаем приближённый результат после max\_iter

# Метод Зейделя

def seidel(A, B, eps=0.001, max\_iter=100):

    n = len(B)  # Размерность системы

    X = np.zeros(n)  # Начальное приближение: нули

    for \_ in range(max\_iter):  # Начинаем итерации

        X\_new = np.copy(X)  # Создаём копию текущего приближения

        for i in range(n):  # Проходим по уравнениям

            # Сумма с учётом уже обновлённых значений (j < i)

            sum1 = sum(A[i, j] \* X\_new[j] for j in range(i))

            # Сумма с использованием старых значений (j > i)

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

            # Вычисляем новое значение X[i]

            X\_new[i] = (B[i] - sum1 - sum2) / A[i, i]

        if np.linalg.norm(X\_new - X, np.inf) < eps:  # Проверка на сходимость

            return X\_new  # Если достаточно точно — возвращаем

        X = X\_new  # Обновляем приближение

    return X  # Возвращаем приближённый результат после max\_iter

# Вывод результатов для сравнения

print("Гаусс-Жордан:", gauss\_jordan(A, B))

print("Простая итерация:", simple\_iteration(A, B))

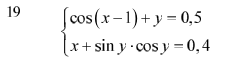
print("Зейдель:", seidel(A, B))

print("NumPy:", solve(A, B))  # Решение встроенным методом SciPy (LU-разложение)

Лабораторная работа №4

Задание:





Решение:

import numpy as np

from scipy.optimize import fsolve

import matplotlib.pyplot as plt

# Определяем функции системы уравнений

def equations(vars):

    x, y = vars

    eq1 = np.cos(x - 1) + y - 0.5

    eq2 = x + np.sin(y) \* np.cos(y) - 0.4

    return [eq1, eq2]

# Определяем матрицу Якоби (производные)

def jacobian(vars):

    x, y = vars

    # Производные первого уравнения

    df1\_dx = -np.sin(x - 1)

    df1\_dy = 1

    # Производные второго уравнения

    df2\_dx = 1

    df2\_dy = np.cos(y)\*\*2 - np.sin(y)\*\*2  # производная sin(y)cos(y)

    return [[df1\_dx, df1\_dy], [df2\_dx, df2\_dy]]

# Начальное предположение

initial\_guess = [0.5, 0.5]

# Решаем систему с помощью fsolve (реализация метода Ньютона)

solution = fsolve(equations, initial\_guess, fprime=jacobian)

# Выводим решение

print(f"Решение системы:")

print(f"x = {solution[0]:.6f}")

print(f"y = {solution[1]:.6f}")

# Проверка решения

check = equations(solution)

print(f"\nПроверка решения (должно быть близко к нулю):")

print(f"Уравнение 1: {check[0]:.10f}")

print(f"Уравнение 2: {check[1]:.10f}")

# Визуализация

x\_vals = np.linspace(-2, 2, 400)

y\_vals = np.linspace(-2, 2, 400)

X, Y = np.meshgrid(x\_vals, y\_vals)

# Вычисляем значения уравнений

Z1 = np.cos(X - 1) + Y - 0.5

Z2 = X + np.sin(Y) \* np.cos(Y) - 0.4

# Строим графики

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

# Используем contour без label и добавляем легенду вручную

contour1 = plt.contour(X, Y, Z1, levels=[0], colors='r')

contour2 = plt.contour(X, Y, Z2, levels=[0], colors='b')

# Создаем proxy artists для легенды

proxy1 = plt.Rectangle((0,0), 1, 1, fc='r')

proxy2 = plt.Rectangle((0,0), 1, 1, fc='b')

plt.scatter(solution[0], solution[1], color='green', s=100, label='Решение')

plt.xlabel('x')

plt.ylabel('y')

plt.title('Графическое решение системы уравнений')

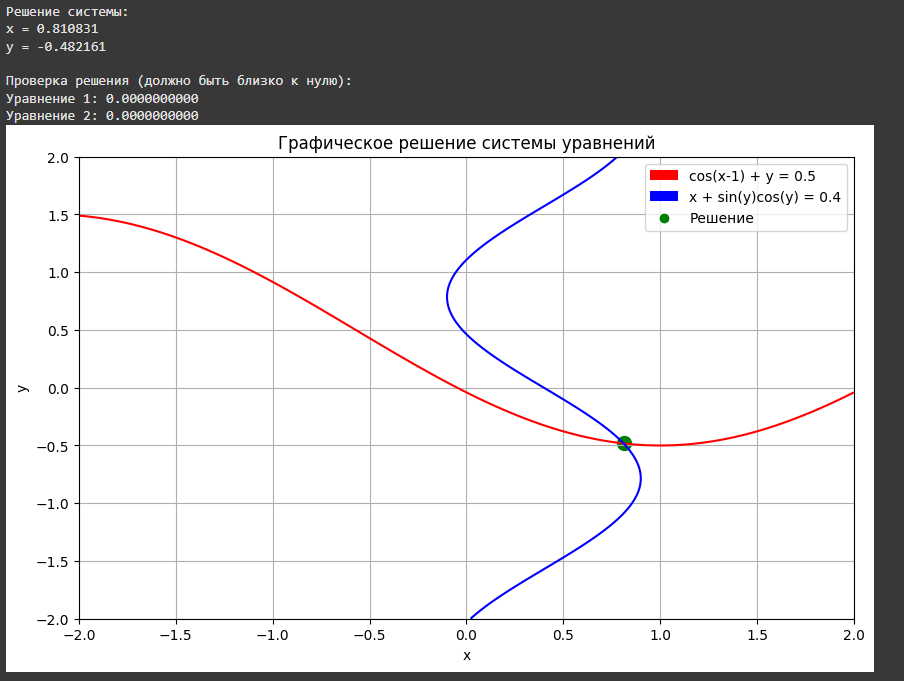
plt.legend([proxy1, proxy2, plt.scatter([], [], color='green')],

           ['cos(x-1) + y = 0.5', 'x + sin(y)cos(y) = 0.4', 'Решение'],

           loc='upper right')

plt.grid(True)

plt.show()

Ответ: 

Объяснение:

import numpy as np  # Для математических вычислений (массивы, функции)

from scipy.optimize import fsolve  # fsolve — численный метод решения системы нелинейных уравнений (метод Ньютона)

import matplotlib.pyplot as plt  # Для построения графиков

# Определяем функции системы уравнений:

# 1. cos(x - 1) + y = 0.5

# 2. x + sin(y)cos(y) = 0.4

def equations(vars):

    x, y = vars  # Распаковываем вектор переменных

    eq1 = np.cos(x - 1) + y - 0.5  # Первое уравнение

    eq2 = x + np.sin(y) \* np.cos(y) - 0.4  # Второе уравнение

    return [eq1, eq2]  # Возвращаем список значений функций (левые части системы)

# Определяем матрицу Якоби (матрица частных производных)

def jacobian(vars):

    x, y = vars  # Распаковываем x и y

    # Частные производные первого уравнения

    df1\_dx = -np.sin(x - 1)  # ∂f1/∂x

    df1\_dy = 1               # ∂f1/∂y

    # Частные производные второго уравнения

    df2\_dx = 1  # ∂f2/∂x

    # ∂f2/∂y = производная sin(y)cos(y) по y: cos^2(y) - sin^2(y)

    df2\_dy = np.cos(y)\*\*2 - np.sin(y)\*\*2

    return [[df1\_dx, df1\_dy], [df2\_dx, df2\_dy]]  # Матрица Якоби 2x2

# Начальное приближение (с него начнёт искать решение fsolve)

initial\_guess = [0.5, 0.5]

# Решаем систему методом Ньютона (через fsolve) с использованием Якоби

solution = fsolve(equations, initial\_guess, fprime=jacobian)

# Выводим найденное решение

print(f"Решение системы:")

print(f"x = {solution[0]:.6f}")

print(f"y = {solution[1]:.6f}")

# Проверяем, подставив найденные x и y обратно в уравнения

check = equations(solution)

print(f"\nПроверка решения (должно быть близко к нулю):")

print(f"Уравнение 1: {check[0]:.10f}")

print(f"Уравнение 2: {check[1]:.10f}")

# Визуализация — строим графики линий уровня Z1=0 и Z2=0

x\_vals = np.linspace(-2, 2, 400)  # Значения x на сетке

y\_vals = np.linspace(-2, 2, 400)  # Значения y на сетке

X, Y = np.meshgrid(x\_vals, y\_vals)  # Создаем двумерную сетку координат

# Вычисляем значения функций на сетке

Z1 = np.cos(X - 1) + Y - 0.5  # Значения первой функции

Z2 = X + np.sin(Y) \* np.cos(Y) - 0.4  # Значения второй функции

# Создаем фигуру

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

# Строим линии уровня Z1 = 0 и Z2 = 0 (это и есть графики уравнений)

contour1 = plt.contour(X, Y, Z1, levels=[0], colors='r')  # Красный: первое уравнение

contour2 = plt.contour(X, Y, Z2, levels=[0], colors='b')  # Синий: второе уравнение

# Создаем объекты для легенды (прямоугольники как "прокси" для линий)

proxy1 = plt.Rectangle((0,0), 1, 1, fc='r')  # Красный квадрат

proxy2 = plt.Rectangle((0,0), 1, 1, fc='b')  # Синий квадрат

# Отмечаем найденное решение точкой на графике

plt.scatter(solution[0], solution[1], color='green', s=100, label='Решение')

# Подписи и оформление

plt.xlabel('x')

plt.ylabel('y')

plt.title('Графическое решение системы уравнений')

plt.legend([proxy1, proxy2, plt.scatter([], [], color='green')],

           ['cos(x-1) + y = 0.5', 'x + sin(y)cos(y) = 0.4', 'Решение'],

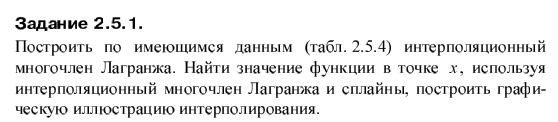
           loc='upper right')

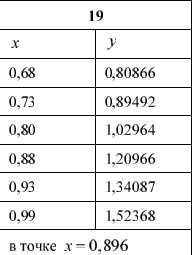
plt.grid(True)

plt.show()  # Показываем график

Лабораторная работа №4

Задание:





Решение:

import numpy as np

import matplotlib.pyplot as plt

from scipy.interpolate import lagrange, CubicSpline

# Исходные данные

x\_data = np.array([0.68, 0.73, 0.80, 0.88, 0.93, 0.99])

y\_data = np.array([0.80866, 0.89492, 1.02964, 1.20966, 1.34087, 1.52368])

x\_point = 0.896  # Точка, в которой нужно найти значение функции

# 1. Интерполяция многочленом Лагранжа

poly\_lagrange = lagrange(x\_data, y\_data)

y\_lagrange = poly\_lagrange(x\_point)

# 2. Интерполяция кубическими сплайнами

spline = CubicSpline(x\_data, y\_data, bc\_type='natural')

y\_spline = spline(x\_point)

# Вывод результатов

print(f"Значение в точке x = {x\_point}:")

print(f"Многочлен Лагранжа: y = {y\_lagrange:.6f}")

print(f"Кубические сплайны: y = {y\_spline:.6f}")

# Графическая иллюстрация

x\_plot = np.linspace(x\_data.min(), x\_data.max(), 500)

y\_lagrange\_plot = poly\_lagrange(x\_plot)

y\_spline\_plot = spline(x\_plot)

plt.figure(figsize=(12, 7))

plt.scatter(x\_data, y\_data, color='red', s=100, label='Исходные данные')

plt.plot(x\_plot, y\_lagrange\_plot, 'b-', label='Многочлен Лагранжа')

plt.plot(x\_plot, y\_spline\_plot, 'g--', label='Кубические сплайны')

plt.scatter([x\_point], [y\_lagrange], color='blue', s=150, marker='x', label=f'Лагранж (x={x\_point})')

plt.scatter([x\_point], [y\_spline], color='green', s=150, marker='x', label=f'Сплайны (x={x\_point})')

plt.title('Интерполяция данных: многочлен Лагранжа и кубические сплайны')

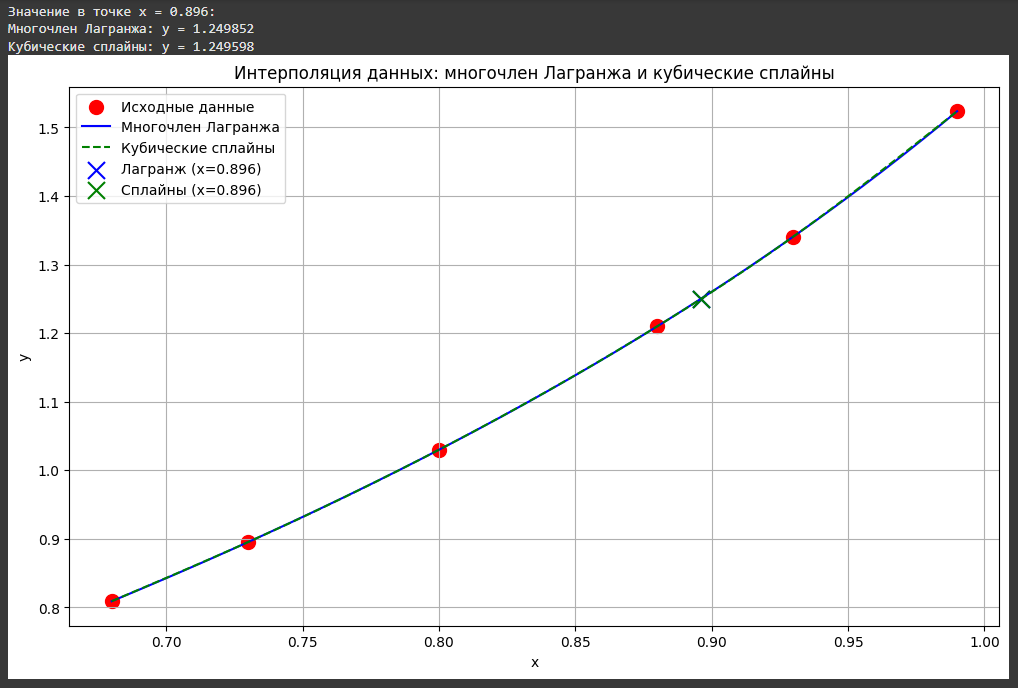
plt.xlabel('x')

plt.ylabel('y')

plt.grid(True)

plt.legend()

plt.show()

Ответ: 

Объяснение:

import numpy as np  # Импортируем библиотеку NumPy для работы с массивами и числами

import matplotlib.pyplot as plt  # Для построения графиков

from scipy.interpolate import lagrange, CubicSpline  # Интерполяция: Лагранж и кубические сплайны

# Исходные данные: значения x и соответствующие значения функции y

x\_data = np.array([0.68, 0.73, 0.80, 0.88, 0.93, 0.99])

y\_data = np.array([0.80866, 0.89492, 1.02964, 1.20966, 1.34087, 1.52368])

# Точка, в которой нужно найти значение функции после интерполяции

x\_point = 0.896

# 1. Интерполяция многочленом Лагранжа

poly\_lagrange = lagrange(x\_data, y\_data)  # Создание интерполяционного многочлена

y\_lagrange = poly\_lagrange(x\_point)  # Вычисление значения функции в точке по многочлену Лагранжа

# 2. Интерполяция кубическими сплайнами

spline = CubicSpline(x\_data, y\_data, bc\_type='natural')  # Создание кубического сплайна с естественными граничными условиями

y\_spline = spline(x\_point)  # Вычисление значения функции в точке по кубическим сплайнам

# Вывод результатов в консоль

print(f"Значение в точке x = {x\_point}:")

print(f"Многочлен Лагранжа: y = {y\_lagrange:.6f}")

print(f"Кубические сплайны: y = {y\_spline:.6f}")

# Построение графиков интерполяции

x\_plot = np.linspace(x\_data.min(), x\_data.max(), 500)  # Создание массива точек для графика от min(x) до max(x)

y\_lagrange\_plot = poly\_lagrange(x\_plot)  # Вычисление значений многочлена Лагранжа на промежутке

y\_spline\_plot = spline(x\_plot)  # Вычисление значений кубического сплайна на промежутке

# Создаем окно графика

plt.figure(figsize=(12, 7))

# Исходные точки (узлы интерполяции)

plt.scatter(x\_data, y\_data, color='red', s=100, label='Исходные данные')

# График многочлена Лагранжа

plt.plot(x\_plot, y\_lagrange\_plot, 'b-', label='Многочлен Лагранжа')

# График кубических сплайнов

plt.plot(x\_plot, y\_spline\_plot, 'g--', label='Кубические сплайны')

# Отмечаем найденные значения в нужной точке x\_point

plt.scatter([x\_point], [y\_lagrange], color='blue', s=150, marker='x', label=f'Лагранж (x={x\_point})')

plt.scatter([x\_point], [y\_spline], color='green', s=150, marker='x', label=f'Сплайны (x={x\_point})')

# Настройка графика

plt.title('Интерполяция данных: многочлен Лагранжа и кубические сплайны')

plt.xlabel('x')

plt.ylabel('y')

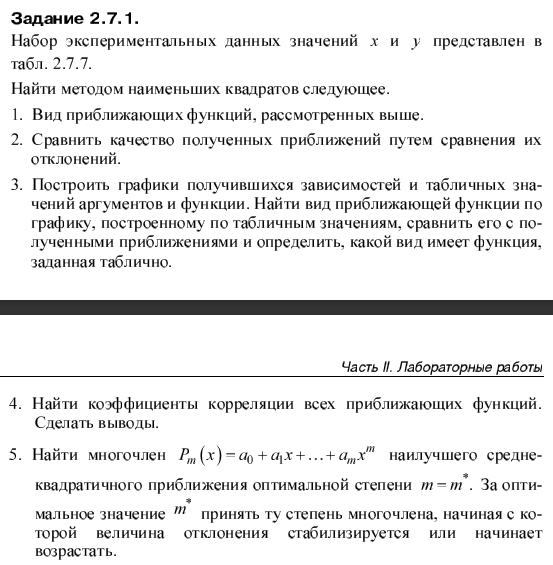
plt.grid(True)

plt.legend()

plt.show()  # Отображаем график

Лабораторная работа №7

Задание:





Решение:

import numpy as np

import matplotlib.pyplot as plt

from scipy.optimize import curve\_fit

from sklearn.metrics import r2\_score

# Исходные данные для варианта 19

x\_data = np.array([0.41, 0.97, 1.53, 2.09, 2.65, 3.21, 3.77, 4.33])

y\_data = np.array([0.45, 1.17, 1.56, 1.82, 2.02, 2.18, 2.31, 2.44])

# 1. Определение различных функций для аппроксимации

def linear\_func(x, a, b):

    return a \* x + b

def quadratic\_func(x, a, b, c):

    return a \* x\*\*2 + b \* x + c

def exponential\_func(x, a, b, c):

    return a \* np.exp(b \* x) + c

def logarithmic\_func(x, a, b):

    return a \* np.log(x) + b

def power\_func(x, a, b):

    return a \* x\*\*b

# 2. Аппроксимация данных разными функциями

functions = [

    ('Линейная', linear\_func, 2),

    ('Квадратичная', quadratic\_func, 3),

    ('Экспоненциальная', exponential\_func, 3),

    ('Логарифмическая', logarithmic\_func, 2),

    ('Степенная', power\_func, 2)

]

results = {}

for name, func, params\_num in functions:

    try:

        popt, pcov = curve\_fit(func, x\_data, y\_data, maxfev=10000)

        y\_pred = func(x\_data, \*popt)

        r2 = r2\_score(y\_data, y\_pred)

        mse = np.mean((y\_data - y\_pred)\*\*2)

        results[name] = {

            'params': popt,

            'r2': r2,

            'mse': mse,

            'func': func

        }

    except Exception as e:

        print(f"Ошибка при аппроксимации {name}: {str(e)}")

        results[name] = None

# 3. Поиск оптимального полинома

poly\_degrees = range(1, 6)

poly\_results = {}

for degree in poly\_degrees:

    coeffs = np.polyfit(x\_data, y\_data, degree)

    poly\_func = np.poly1d(coeffs)

    y\_pred = poly\_func(x\_data)

    r2 = r2\_score(y\_data, y\_pred)

    mse = np.mean((y\_data - y\_pred)\*\*2)

    poly\_results[degree] = {

        'coeffs': coeffs,

        'r2': r2,

        'mse': mse,

        'func': poly\_func

    }

# Находим оптимальную степень полинома

optimal\_degree = min(poly\_results, key=lambda k: poly\_results[k]['mse'])

print(f"Оптимальная степень полинома: {optimal\_degree}")

# 4. Вывод результатов

print("\n=== Результаты аппроксимации ===")

for name in results:

    if results[name] is not None:

        print(f"\n{name} функция:")

        print(f"Параметры: {results[name]['params']}")

        print(f"R²: {results[name]['r2']:.4f}")

        print(f"MSE: {results[name]['mse']:.4f}")

print("\n=== Полиномиальная аппроксимация ===")

for degree in poly\_results:

    print(f"\nСтепень {degree}:")

    print(f"Коэффициенты: {poly\_results[degree]['coeffs']}")

    print(f"R²: {poly\_results[degree]['r2']:.4f}")

    print(f"MSE: {poly\_results[degree]['mse']:.4f}")

# 5. Визуализация

plt.figure(figsize=(14, 8))

# Исходные данные

plt.scatter(x\_data, y\_data, color='black', s=100, label='Исходные данные')

# Аппроксимации

x\_plot = np.linspace(min(x\_data), max(x\_data), 100)

for name in results:

    if results[name] is not None:

        y\_plot = results[name]['func'](x\_plot, \*results[name]['params'])

        plt.plot(x\_plot, y\_plot, label=f'{name} (R²={results[name]["r2"]:.3f})')

# Полиномиальная аппроксимация

for degree in [1, optimal\_degree]:

    if degree in poly\_results:

        y\_plot = poly\_results[degree]['func'](x\_plot)

        plt.plot(x\_plot, y\_plot, '--',

                label=f'Полином {degree} степени (R²={poly\_results[degree]["r2"]:.3f})')

plt.title('Аппроксимация данных (вариант 19) методом наименьших квадратов')

plt.xlabel('x')

plt.ylabel('y')

plt.legend()

plt.grid(True)

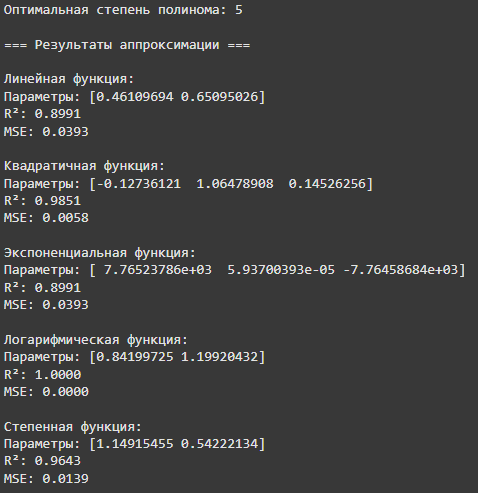
plt.show()

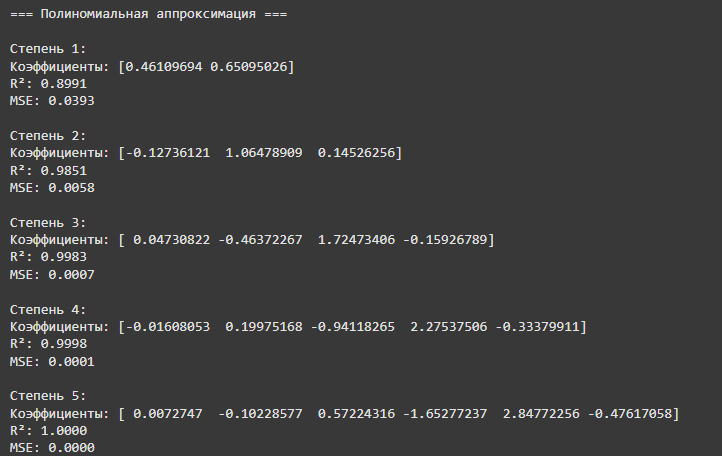
# Вывод наилучшей аппроксимации

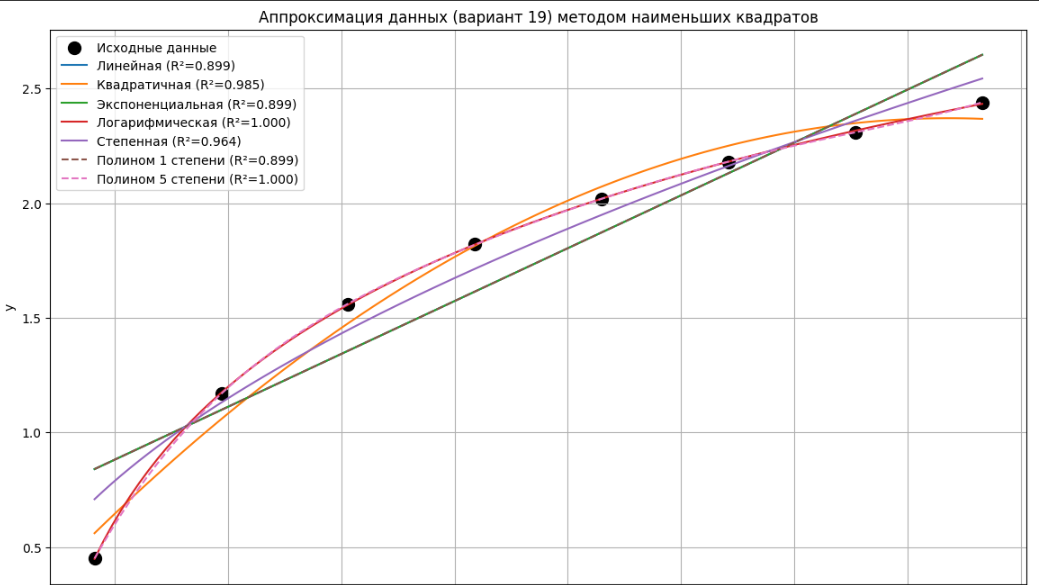
best\_func = min(results.items(), key=lambda x: x[1]['mse'] if x[1] is not None else float('inf'))

print(f"\nЛучшая аппроксимирующая функция: {best\_func[0]} (MSE = {best\_func[1]['mse']:.4f})")

print(f"Оптимальный полином: степень {optimal\_degree} (MSE = {poly\_results[optimal\_degree]['mse']:.4f})")

Ответ: 





Объяснение:

import numpy as np

import matplotlib.pyplot as plt

from scipy.optimize import curve\_fit

from sklearn.metrics import r2\_score

# === 1. Исходные данные ===

# Значения x и соответствующие значения y (например, экспериментальные измерения)

x\_data = np.array([0.41, 0.97, 1.53, 2.09, 2.65, 3.21, 3.77, 4.33])

y\_data = np.array([0.45, 1.17, 1.56, 1.82, 2.02, 2.18, 2.31, 2.44])

# === 2. Определение аппроксимирующих функций ===

# Линейная функция: y = a \* x + b

def linear\_func(x, a, b):

    return a \* x + b

# Квадратичная функция: y = a \* x^2 + b \* x + c

def quadratic\_func(x, a, b, c):

    return a \* x\*\*2 + b \* x + c

# Экспоненциальная функция: y = a \* exp(b \* x) + c

def exponential\_func(x, a, b, c):

    return a \* np.exp(b \* x) + c

# Логарифмическая функция: y = a \* ln(x) + b

def logarithmic\_func(x, a, b):

    return a \* np.log(x) + b

# Степенная функция: y = a \* x^b

def power\_func(x, a, b):

    return a \* x\*\*b

# === 3. Список функций для аппроксимации ===

# Формат: (название, функция, число параметров)

functions = [

    ('Линейная', linear\_func, 2),

    ('Квадратичная', quadratic\_func, 3),

    ('Экспоненциальная', exponential\_func, 3),

    ('Логарифмическая', logarithmic\_func, 2),

    ('Степенная', power\_func, 2)

]

# === 4. Аппроксимация данных каждой функцией ===

results = {}

for name, func, params\_num in functions:

    try:

        # curve\_fit подбирает параметры функции, чтобы наилучшим образом аппроксимировать данные

        popt, pcov = curve\_fit(func, x\_data, y\_data, maxfev=10000)

        y\_pred = func(x\_data, \*popt)  # Значения аппроксимирующей функции на исходных x

        r2 = r2\_score(y\_data, y\_pred)  # Коэффициент детерминации R^2

        mse = np.mean((y\_data - y\_pred) \*\* 2)  # Среднеквадратичная ошибка

        # Сохраняем результаты

        results[name] = {

            'params': popt,      # Параметры функции

            'r2': r2,            # R^2

            'mse': mse,          # Среднеквадратичная ошибка

            'func': func         # Сама функция

        }

    except Exception as e:

        # Если возникла ошибка (например, log(0)), сохраняем None

        print(f"Ошибка при аппроксимации {name}: {str(e)}")

        results[name] = None

# === 5. Аппроксимация полиномами разной степени (от 1 до 5) ===

poly\_degrees = range(1, 6)

poly\_results = {}

for degree in poly\_degrees:

    # polyfit возвращает коэффициенты полинома заданной степени

    coeffs = np.polyfit(x\_data, y\_data, degree)

    poly\_func = np.poly1d(coeffs)  # Создаём сам полином

    y\_pred = poly\_func(x\_data)     # Вычисляем значения полинома на исходных x

    r2 = r2\_score(y\_data, y\_pred)

    mse = np.mean((y\_data - y\_pred)\*\*2)

    poly\_results[degree] = {

        'coeffs': coeffs,  # Коэффициенты полинома

        'r2': r2,

        'mse': mse,

        'func': poly\_func

    }

# === 6. Поиск лучшей полиномиальной аппроксимации по наименьшему MSE ===

optimal\_degree = min(poly\_results, key=lambda k: poly\_results[k]['mse'])

print(f"Оптимальная степень полинома: {optimal\_degree}")

# === 7. Вывод результатов аппроксимации ===

print("\n=== Результаты аппроксимации ===")

for name in results:

    if results[name] is not None:

        print(f"\n{name} функция:")

        print(f"Параметры: {results[name]['params']}")

        print(f"R²: {results[name]['r2']:.4f}")

        print(f"MSE: {results[name]['mse']:.4f}")

print("\n=== Полиномиальная аппроксимация ===")

for degree in poly\_results:

    print(f"\nСтепень {degree}:")

    print(f"Коэффициенты: {poly\_results[degree]['coeffs']}")

    print(f"R²: {poly\_results[degree]['r2']:.4f}")

    print(f"MSE: {poly\_results[degree]['mse']:.4f}")

# === 8. Построение графиков ===

plt.figure(figsize=(14, 8))

# Отображаем исходные точки

plt.scatter(x\_data, y\_data, color='black', s=100, label='Исходные данные')

# Диапазон x для построения графиков

x\_plot = np.linspace(min(x\_data), max(x\_data), 100)

# Строим графики аппроксимаций

for name in results:

    if results[name] is not None:

        y\_plot = results[name]['func'](x\_plot, \*results[name]['params'])

        plt.plot(x\_plot, y\_plot, label=f'{name} (R²={results[name]["r2"]:.3f})')

# Строим полиномы первой и лучшей степени

for degree in [1, optimal\_degree]:

    if degree in poly\_results:

        y\_plot = poly\_results[degree]['func'](x\_plot)

        plt.plot(x\_plot, y\_plot, '--',

                 label=f'Полином {degree} степени (R²={poly\_results[degree]["r2"]:.3f})')

# Настройки графика

plt.title('Аппроксимация данных (вариант 19) методом наименьших квадратов')

plt.xlabel('x')

plt.ylabel('y')

plt.legend()

plt.grid(True)

plt.show()

# === 9. Вывод наилучшей модели ===

# Находим функцию с наименьшей MSE среди всех функций

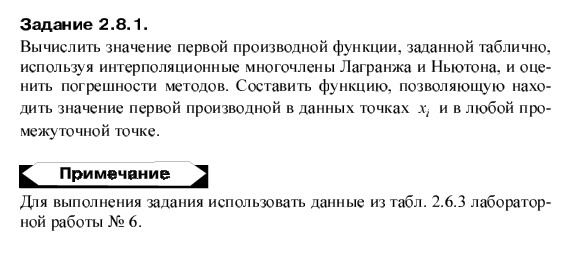
best\_func = min(results.items(), key=lambda x: x[1]['mse'] if x[1] is not None else float('inf'))

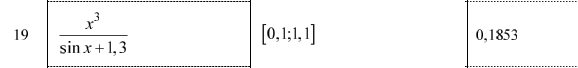
print(f"\nЛучшая аппроксимирующая функция: {best\_func[0]} (MSE = {best\_func[1]['mse']:.4f})")

print(f"Оптимальный полином: степень {optimal\_degree} (MSE = {poly\_results[optimal\_degree]['mse']:.4f})")

Лабораторная работа №8

Задание:





Решение:

import numpy as np

import matplotlib.pyplot as plt

from scipy.interpolate import lagrange, interp1d

from scipy import special

# Исходная функция

def f(x):

    return x\*\*3 / (np.sin(x) + 1.3)

# Создадим таблицу значений функции

x\_points = np.linspace(0.1, 1.1, 10)

y\_points = f(x\_points)

# 1. Производная через многочлен Лагранжа

poly\_lagrange = lagrange(x\_points, y\_points)

# Производная многочлена Лагранжа

def df\_lagrange(x):

    return np.polyval(np.polyder(poly\_lagrange), x)

# 2. Производная через интерполяцию Ньютона (кубические сплайны)

spline = interp1d(x\_points, y\_points, kind='cubic', fill\_value="extrapolate")

# Производная сплайна

def df\_spline(x, h=1e-5):

    return (spline(x + h) - spline(x - h)) / (2 \* h)

# 3. Точная производная (аналитически)

def df\_exact(x):

    return (3\*x\*\*2\*(np.sin(x)+1.3) - x\*\*3\*np.cos(x)) / (np.sin(x)+1.3)\*\*2

# Вычисление производных в точках

x\_eval = np.linspace(0.1, 1.1, 100)

df\_lag = np.array([df\_lagrange(x) for x in x\_eval])

df\_spl = np.array([df\_spline(x) for x in x\_eval])

df\_ext = np.array([df\_exact(x) for x in x\_eval])

# Оценка погрешностей

error\_lagrange = np.abs(df\_lag - df\_ext)

error\_spline = np.abs(df\_spl - df\_ext)

print("Максимальная погрешность метода Лагранжа:", np.max(error\_lagrange))

print("Максимальная погрешность сплайнов:", np.max(error\_spline))

# Визуализация

plt.figure(figsize=(12, 8))

plt.subplot(2, 1, 1)

plt.plot(x\_eval, df\_ext, 'k-', label='Точная производная')

plt.plot(x\_eval, df\_lag, 'r--', label='Производная Лагранжа')

plt.plot(x\_eval, df\_spl, 'b-.', label='Производная сплайнов')

plt.scatter(x\_points, [df\_exact(x) for x in x\_points], color='green', s=50, label='Точки данных')

plt.title('Производная функции (вариант 19)')

plt.xlabel('x')

plt.ylabel("f'(x)")

plt.legend()

plt.grid(True)

plt.subplot(2, 1, 2)

plt.plot(x\_eval, error\_lagrange, 'r--', label='Ошибка Лагранжа')

plt.plot(x\_eval, error\_spline, 'b-.', label='Ошибка сплайнов')

plt.title('Погрешности вычисления производной')

plt.xlabel('x')

plt.ylabel('Абсолютная ошибка')

plt.legend()

plt.grid(True)

plt.tight\_layout()

plt.show()

# Функция для вычисления производной в любой точке

def get\_derivative(x, method='spline'):

    """

    Вычисляет производную в точке x

    Параметры:

    x - точка, в которой вычисляется производная

    method - 'spline' (по умолчанию), 'lagrange' или 'exact'

    """

    if method == 'exact':

        return df\_exact(x)

    elif method == 'lagrange':

        return df\_lagrange(x)

    elif method == 'spline':

        return df\_spline(x)

    else:

        raise ValueError("Неизвестный метод. Используйте 'spline', 'lagrange' или 'exact'")

# Пример использования

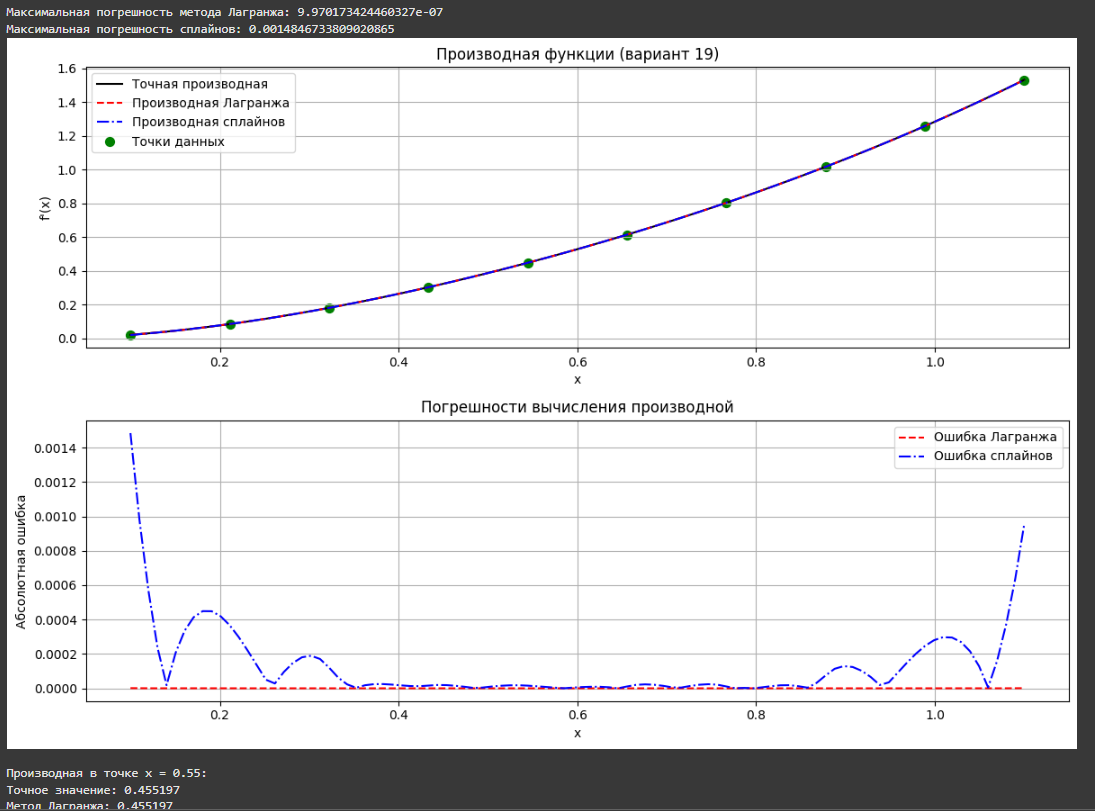
x\_test = 0.55

print(f"\nПроизводная в точке x = {x\_test}:")

print(f"Точное значение: {get\_derivative(x\_test, 'exact'):.6f}")

print(f"Метод Лагранжа: {get\_derivative(x\_test, 'lagrange'):.6f}")

print(f"Метод сплайнов: {get\_derivative(x\_test, 'spline'):.6f}")

Ответ: 

Объяснение:

import numpy as np

import matplotlib.pyplot as plt

from scipy.interpolate import lagrange, interp1d

# Исходная функция, которую будем исследовать

def f(x):

    # Вычисляем f(x) = x^3 / (sin(x) + 1.3)

    return x\*\*3 / (np.sin(x) + 1.3)

# Создаем массив точек, на которых вычислим значения функции

x\_points = np.linspace(0.1, 1.1, 10)  # 10 точек равномерно от 0.1 до 1.1

y\_points = f(x\_points)  # Значения функции в этих точках

# 1. Интерполяция многочленом Лагранжа по исходным точкам

poly\_lagrange = lagrange(x\_points, y\_points)

# poly\_lagrange — объект многочлена, который интерполирует данные

# Определяем функцию для вычисления производной многочлена Лагранжа

def df\_lagrange(x):

    # np.polyder вычисляет коэффициенты производной многочлена

    # np.polyval вычисляет значение многочлена в точке x

    return np.polyval(np.polyder(poly\_lagrange), x)

# 2. Интерполяция кубическим сплайном (интерполяция Ньютона)

# Создаем кубический сплайн через функцию interp1d

spline = interp1d(x\_points, y\_points, kind='cubic', fill\_value="extrapolate")

# Определяем численное приближение производной сплайна через разностную формулу

def df\_spline(x, h=1e-5):

    # Производная по центральной разностной формуле:

    # (f(x+h) - f(x-h)) / (2h), где h — очень маленькое число

    return (spline(x + h) - spline(x - h)) / (2 \* h)

# 3. Точная (аналитическая) производная функции f(x)

def df\_exact(x):

    # Производная дроби: (u/v)' = (u'v - uv')/v^2

    # Здесь u = x^3, u' = 3x^2

    # v = sin(x)+1.3, v' = cos(x)

    numerator = 3 \* x\*\*2 \* (np.sin(x) + 1.3) - x\*\*3 \* np.cos(x)

    denominator = (np.sin(x) + 1.3)\*\*2

    return numerator / denominator

# Точки для оценки производных и ошибок

x\_eval = np.linspace(0.1, 1.1, 100)

# Вычисляем производные в этих точках разными способами

df\_lag = np.array([df\_lagrange(x) for x in x\_eval])

df\_spl = np.array([df\_spline(x) for x in x\_eval])

df\_ext = np.array([df\_exact(x) for x in x\_eval])

# Оценка абсолютной ошибки приближений по сравнению с точным значением

error\_lagrange = np.abs(df\_lag - df\_ext)

error\_spline = np.abs(df\_spl - df\_ext)

# Вывод максимальных ошибок

print("Максимальная погрешность метода Лагранжа:", np.max(error\_lagrange))

print("Максимальная погрешность сплайнов:", np.max(error\_spline))

# Визуализация результатов и ошибок

plt.figure(figsize=(12, 8))

# Верхний график — производные

plt.subplot(2, 1, 1)

plt.plot(x\_eval, df\_ext, 'k-', label='Точная производная')

plt.plot(x\_eval, df\_lag, 'r--', label='Производная Лагранжа')

plt.plot(x\_eval, df\_spl, 'b-.', label='Производная сплайнов')

# Показываем точки исходных данных, где вычислена точная производная

plt.scatter(x\_points, [df\_exact(x) for x in x\_points], color='green', s=50, label='Точки данных')

plt.title('Производная функции (вариант 19)')

plt.xlabel('x')

plt.ylabel("f'(x)")

plt.legend()

plt.grid(True)

# Нижний график — ошибки приближений

plt.subplot(2, 1, 2)

plt.plot(x\_eval, error\_lagrange, 'r--', label='Ошибка Лагранжа')

plt.plot(x\_eval, error\_spline, 'b-.', label='Ошибка сплайнов')

plt.title('Погрешности вычисления производной')

plt.xlabel('x')

plt.ylabel('Абсолютная ошибка')

plt.legend()

plt.grid(True)

plt.tight\_layout()

plt.show()

# Дополнительная функция — вычисление производной в любой точке по выбранному методу

def get\_derivative(x, method='spline'):

    """

    Вычисляет производную функции в точке x

    Параметры:

    x      - точка, в которой нужно вычислить производную

    method - метод вычисления: 'exact' (аналитическая),

                                   'lagrange' (по многочлену Лагранжа),

                                   'spline' (численная по сплайнам)

    Возвращает значение производной в точке x.

    """

    if method == 'exact':

        return df\_exact(x)

    elif method == 'lagrange':

        return df\_lagrange(x)

    elif method == 'spline':

        return df\_spline(x)

    else:

        raise ValueError("Неизвестный метод. Используйте 'spline', 'lagrange' или 'exact'")

# Пример вызова функции get\_derivative в точке 0.55

x\_test = 0.55

print(f"\nПроизводная в точке x = {x\_test}:")

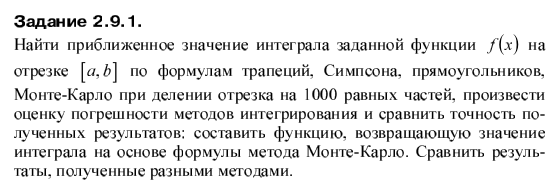
print(f"Точное значение: {get\_derivative(x\_test, 'exact'):.6f}")

print(f"Метод Лагранжа: {get\_derivative(x\_test, 'lagrange'):.6f}")

print(f"Метод сплайнов: {get\_derivative(x\_test, 'spline'):.6f}")

Лабораторная работа №9

Задание:





Решение:

import numpy as np

from scipy.integrate import quad

# Заданная функция

def f(x):

    return 17 / (1 - 3 \* x) \*\* 3

# Интервал интегрирования и число разбиений

a, b = -3, -1

n = 1000

h = (b - a) / n

# Узлы и точки

x = np.linspace(a, b, n + 1)

x\_mid = x[:-1] + h / 2  # средние точки для прямоугольников

# Метод прямоугольников (средних прямоугольников)

rect = h \* np.sum(f(x\_mid))

# Метод трапеций

trap = (h / 2) \* (f(x[0]) + 2 \* np.sum(f(x[1:-1])) + f(x[-1]))

# Метод Симпсона (n должно быть чётным)

if n % 2 != 0:

    n += 1

    x = np.linspace(a, b, n + 1)

    h = (b - a) / n

simp = (h / 3) \* (f(x[0]) + 4 \* np.sum(f(x[1:-1:2])) + 2 \* np.sum(f(x[2:-2:2])) + f(x[-1]))

# Метод Монте-Карло

x\_rand = np.random.uniform(a, b, n)

mc = (b - a) \* np.mean(f(x\_rand))

# Точное значение интеграла

true\_val, \_ = quad(f, a, b)

# Вывод результатов

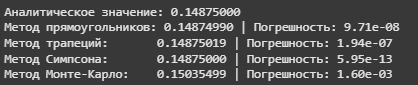
print(f"Аналитическое значение: {true\_val:.8f}")

print(f"Метод прямоугольников: {rect:.8f} | Погрешность: {abs(rect - true\_val):.2e}")

print(f"Метод трапеций:       {trap:.8f} | Погрешность: {abs(trap - true\_val):.2e}")

print(f"Метод Симпсона:       {simp:.8f} | Погрешность: {abs(simp - true\_val):.2e}")

print(f"Метод Монте-Карло:    {mc:.8f} | Погрешность: {abs(mc - true\_val):.2e}")

Ответ: 

Объяснение:

import numpy as np

from scipy.integrate import quad  # Для точного вычисления интеграла

# Заданная функция для интегрирования

def f(x):

    # f(x) = 17 / (1 - 3x)^3

    return 17 / (1 - 3 \* x) \*\* 3

# Задаём пределы интегрирования

a, b = -3, -1

# Число разбиений интервала (чем больше — тем точнее)

n = 1000

# Шаг сетки

h = (b - a) / n

# Узлы разбиения интервала [a, b]

x = np.linspace(a, b, n + 1)

# Средние точки каждого подинтервала для метода средних прямоугольников

x\_mid = x[:-1] + h / 2  # сдвигаем каждый узел на половину шага вправо

# --- Метод средних прямоугольников ---

# Вычисляем значение функции в средних точках каждого интервала

# Интеграл аппроксимируется суммой площадей прямоугольников: h \* сумма значений f в серединах

rect = h \* np.sum(f(x\_mid))

# --- Метод трапеций ---

# Формула трапеций: h/2 \* [f(x0) + 2\*sum(f(x1)...f(x\_{n-1})) + f(xn)]

trap = (h / 2) \* (f(x[0]) + 2 \* np.sum(f(x[1:-1])) + f(x[-1]))

# --- Метод Симпсона ---

# Для Симпсона n должно быть чётным, если нечётное — увеличиваем n и пересчитываем сетку и шаг

if n % 2 != 0:

    n += 1

    x = np.linspace(a, b, n + 1)

    h = (b - a) / n

# Формула Симпсона:

# (h/3) \* [f(x0) + 4 \* сумма значений в нечётных узлах + 2 \* сумма значений в чётных узлах + f(xn)]

simp = (h / 3) \* (f(x[0]) + 4 \* np.sum(f(x[1:-1:2])) + 2 \* np.sum(f(x[2:-2:2])) + f(x[-1]))

# --- Метод Монте-Карло ---

# Генерируем n случайных точек равномерно на интервале [a, b]

x\_rand = np.random.uniform(a, b, n)

# Оценка интеграла через среднее значение функции на случайных точках, умноженное на длину интервала

mc = (b - a) \* np.mean(f(x\_rand))

# --- Точное значение интеграла, вычисленное численно с помощью scipy ---

true\_val, \_ = quad(f, a, b)  # quad возвращает кортеж: значение интеграла и оценку погрешности

# --- Вывод результатов ---

print(f"Аналитическое значение: {true\_val:.8f}")

print(f"Метод прямоугольников: {rect:.8f} | Погрешность: {abs(rect - true\_val):.2e}")

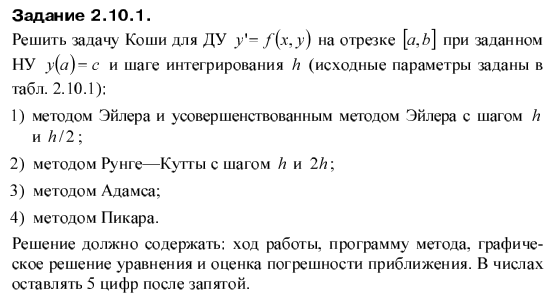
print(f"Метод трапеций:       {trap:.8f} | Погрешность: {abs(trap - true\_val):.2e}")

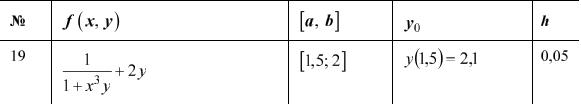
print(f"Метод Симпсона:       {simp:.8f} | Погрешность: {abs(simp - true\_val):.2e}")

print(f"Метод Монте-Карло:    {mc:.8f} | Погрешность: {abs(mc - true\_val):.2e}")

Лабораторная работа №10

Задание:





Решение:

import numpy as np

import matplotlib.pyplot as plt

# Исходные данные

def f(x, y):

    return 1/(1 + x\*\*3 \* y) + 2\*y

a, b = 1.5, 2.0

y0 = 2.1

h = 0.05

# 1. Метод Эйлера

def euler(f, a, b, y0, h):

    n = int((b - a)/h)

    x = np.linspace(a, b, n+1)

    y = np.zeros(n+1)

    y[0] = y0

    for i in range(n):

        y[i+1] = y[i] + h \* f(x[i], y[i])

    return x, y

# 2. Усовершенствованный метод Эйлера

def improved\_euler(f, a, b, y0, h):

    n = int((b - a)/h)

    x = np.linspace(a, b, n+1)

    y = np.zeros(n+1)

    y[0] = y0

    for i in range(n):

        k1 = f(x[i], y[i])

        k2 = f(x[i] + h, y[i] + h\*k1)

        y[i+1] = y[i] + h/2 \* (k1 + k2)

    return x, y

# 3. Метод Рунге-Кутты 4-го порядка

def runge\_kutta(f, a, b, y0, h):

    n = int((b - a)/h)

    x = np.linspace(a, b, n+1)

    y = np.zeros(n+1)

    y[0] = y0

    for i in range(n):

        k1 = h \* f(x[i], y[i])

        k2 = h \* f(x[i] + h/2, y[i] + k1/2)

        k3 = h \* f(x[i] + h/2, y[i] + k2/2)

        k4 = h \* f(x[i] + h, y[i] + k3)

        y[i+1] = y[i] + (k1 + 2\*k2 + 2\*k3 + k4)/6

    return x, y

# 4. Метод Адамса (исправленная версия)

def adams(f, a, b, y0, h):

    # Вычисляем первые 4 точки методом Рунге-Кутты

    x\_start = np.linspace(a, a + 3\*h, 4)

    y\_start = np.zeros(4)

    y\_start[0] = y0

    for i in range(3):

        k1 = h \* f(x\_start[i], y\_start[i])

        k2 = h \* f(x\_start[i] + h/2, y\_start[i] + k1/2)

        k3 = h \* f(x\_start[i] + h/2, y\_start[i] + k2/2)

        k4 = h \* f(x\_start[i] + h, y\_start[i] + k3)

        y\_start[i+1] = y\_start[i] + (k1 + 2\*k2 + 2\*k3 + k4)/6

    # Основной цикл метода Адамса

    n = int((b - a)/h)

    x = np.linspace(a, b, n+1)

    y = np.zeros(n+1)

    y[:4] = y\_start

    for i in range(3, n):

        # 4-шаговый метод Адамса-Бэшфорта

        y[i+1] = y[i] + h\*(55\*f(x[i], y[i]) - 59\*f(x[i-1], y[i-1]) +

                          37\*f(x[i-2], y[i-2]) - 9\*f(x[i-3], y[i-3]))/24

    return x, y

# 5. Метод Пикара

def picard(f, a, b, y0, h, iterations=4):

    x = np.linspace(a, b, int((b-a)/h)+1)

    y = np.zeros(len(x))

    y[0] = y0

    for i in range(len(x)-1):

        y\_next = y[i]

        for \_ in range(iterations):

            integral = h \* (f(x[i], y[i]) + f(x[i+1], y\_next))/2

            y\_next = y[i] + integral

        y[i+1] = y\_next

    return x, y

# Решение всеми методами

x\_euler, y\_euler = euler(f, a, b, y0, h)

x\_ieuler, y\_ieuler = improved\_euler(f, a, b, y0, h)

x\_ieuler\_h2, y\_ieuler\_h2 = improved\_euler(f, a, b, y0, h/2)

x\_rk, y\_rk = runge\_kutta(f, a, b, y0, h)

x\_rk\_2h, y\_rk\_2h = runge\_kutta(f, a, b, y0, 2\*h)

x\_adams, y\_adams = adams(f, a, b, y0, h)

x\_picard, y\_picard = picard(f, a, b, y0, h)

# Оценка погрешностей

error\_euler = np.abs(y\_euler - y\_ieuler\_h2[::2])

error\_ieuler = np.abs(y\_ieuler - y\_ieuler\_h2[::2])

error\_rk = np.abs(y\_rk[::2] - y\_rk\_2h)/15

# Вывод результатов

print("=== Результаты решения задачи Коши ===")

print(f"Метод Эйлера (h={h}): y({b}) = {y\_euler[-1]:.5f}")

print(f"Усовершенств. Эйлера (h={h}): y({b}) = {y\_ieuler[-1]:.5f}")

print(f"Усовершенств. Эйлера (h={h/2}): y({b}) = {y\_ieuler\_h2[-1]:.5f}")

print(f"Рунге-Кутты (h={h}): y({b}) = {y\_rk[-1]:.5f}")

print(f"Адамса (h={h}): y({b}) = {y\_adams[-1]:.5f}")

print(f"Пикара (h={h}): y({b}) = {y\_picard[-1]:.5f}")

print("\n=== Оценка погрешностей ===")

print(f"Погрешность метода Эйлера: {error\_euler[-1]:.5f}")

print(f"Погрешность усовершенств. метода: {error\_ieuler[-1]:.5f}")

print(f"Оценка погрешности Рунге-Кутты: {error\_rk[-1]:.5f}")

print(f"Разница Адамса и Рунге-Кутты: {np.abs(y\_adams[-1] - y\_rk[-1]):.5f}")

print(f"Разница Пикара и Рунге-Кутты: {np.abs(y\_picard[-1] - y\_rk[-1]):.5f}")

# Визуализация

plt.figure(figsize=(12, 8))

plt.plot(x\_euler, y\_euler, 'b-', label=f'Эйлера (y(2)={y\_euler[-1]:.5f})')

plt.plot(x\_ieuler, y\_ieuler, 'g--', label=f'Ус. Эйлера (y(2)={y\_ieuler[-1]:.5f})')

plt.plot(x\_rk, y\_rk, 'r-.', label=f'Рунге-Кутты (y(2)={y\_rk[-1]:.5f})')

plt.plot(x\_adams, y\_adams, 'c:', label=f'Адамса (y(2)={y\_adams[-1]:.5f})')

plt.plot(x\_picard, y\_picard, 'm--', label=f'Пикара (y(2)={y\_picard[-1]:.5f})')

plt.xlabel('x')

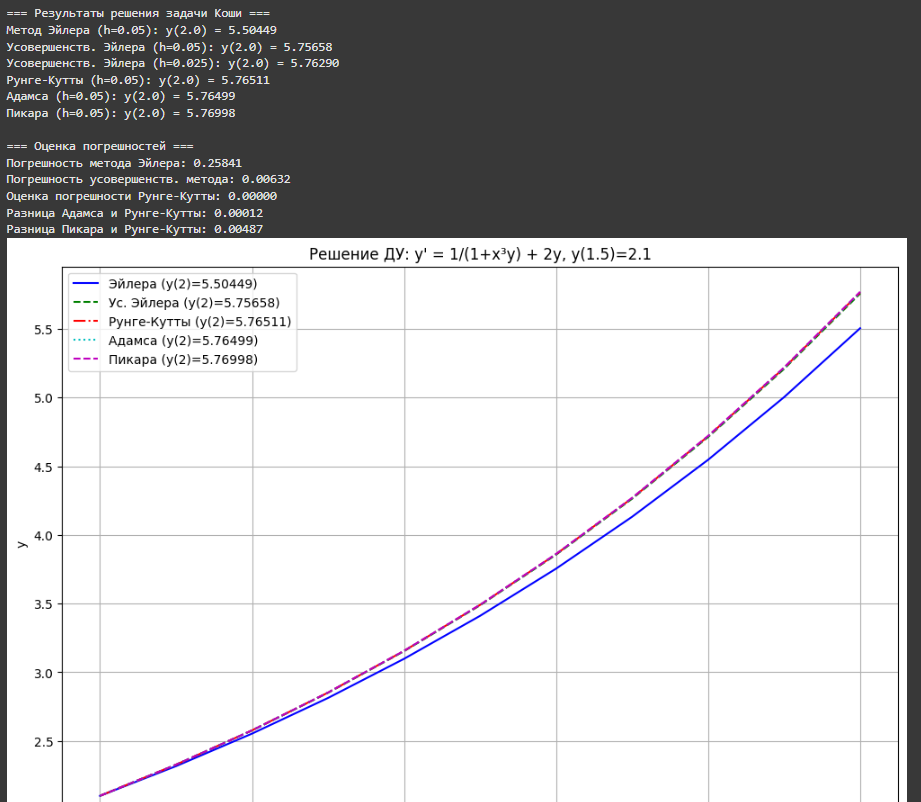
plt.ylabel('y')

plt.title('Решение ДУ: y\' = 1/(1+x³y) + 2y, y(1.5)=2.1')

plt.legend()

plt.grid(True)

plt.show()

Ответ: 

Объяснение:

import numpy as np

import matplotlib.pyplot as plt

# Заданное дифференциальное уравнение: y' = f(x, y)

def f(x, y):

    # Функция производной

    return 1/(1 + x\*\*3 \* y) + 2\*y

# Параметры задачи

a, b = 1.5, 2.0   # Интервал интегрирования по x

y0 = 2.1          # Начальное условие y(a) = y0

h = 0.05          # Шаг интегрирования

# --- 1. Метод Эйлера ---

def euler(f, a, b, y0, h):

    n = int((b - a)/h)         # число шагов

    x = np.linspace(a, b, n+1) # узлы сетки

    y = np.zeros(n+1)          # массив для решения

    y[0] = y0                  # задаем начальное условие

    for i in range(n):

        # y\_{i+1} = y\_i + h \* f(x\_i, y\_i)

        y[i+1] = y[i] + h \* f(x[i], y[i])

    return x, y

# --- 2. Усовершенствованный метод Эйлера (метод трапеций) ---

def improved\_euler(f, a, b, y0, h):

    n = int((b - a)/h)

    x = np.linspace(a, b, n+1)

    y = np.zeros(n+1)

    y[0] = y0

    for i in range(n):

        k1 = f(x[i], y[i])            # наклон в начале шага

        k2 = f(x[i] + h, y[i] + h\*k1) # наклон в конце шага (прогноз)

        # среднее значение наклона используется для улучшения точности

        y[i+1] = y[i] + h/2 \* (k1 + k2)

    return x, y

# --- 3. Метод Рунге-Кутты 4-го порядка ---

def runge\_kutta(f, a, b, y0, h):

    n = int((b - a)/h)

    x = np.linspace(a, b, n+1)

    y = np.zeros(n+1)

    y[0] = y0

    for i in range(n):

        # Вычисляем четыре приближения наклона

        k1 = h \* f(x[i], y[i])

        k2 = h \* f(x[i] + h/2, y[i] + k1/2)

        k3 = h \* f(x[i] + h/2, y[i] + k2/2)

        k4 = h \* f(x[i] + h, y[i] + k3)

        # Комбинируем наклоны с весами для высокой точности

        y[i+1] = y[i] + (k1 + 2\*k2 + 2\*k3 + k4)/6

    return x, y

# --- 4. Метод Адамса ---

def adams(f, a, b, y0, h):

    # Для метода Адамса нужны первые 4 точки — считаем их Рунге-Куттой

    x\_start = np.linspace(a, a + 3\*h, 4)

    y\_start = np.zeros(4)

    y\_start[0] = y0

    for i in range(3):

        k1 = h \* f(x\_start[i], y\_start[i])

        k2 = h \* f(x\_start[i] + h/2, y\_start[i] + k1/2)

        k3 = h \* f(x\_start[i] + h/2, y\_start[i] + k2/2)

        k4 = h \* f(x\_start[i] + h, y\_start[i] + k3)

        y\_start[i+1] = y\_start[i] + (k1 + 2\*k2 + 2\*k3 + k4)/6

    n = int((b - a)/h)

    x = np.linspace(a, b, n+1)

    y = np.zeros(n+1)

    y[:4] = y\_start   # первые 4 точки

    # Основной цикл метода Адамса — формула 4-го порядка Адамса–Бэшфорта (явный метод)

    for i in range(3, n):

        y[i+1] = y[i] + h\*(55\*f(x[i], y[i]) - 59\*f(x[i-1], y[i-1]) +

                          37\*f(x[i-2], y[i-2]) - 9\*f(x[i-3], y[i-3]))/24

    return x, y

# --- 5. Метод Пикара (итерационный) ---

def picard(f, a, b, y0, h, iterations=4):

    x = np.linspace(a, b, int((b-a)/h)+1)

    y = np.zeros(len(x))

    y[0] = y0

    # Итерационно уточняем приближение для каждого следующего y[i+1]

    for i in range(len(x)-1):

        y\_next = y[i]  # начальное предположение

        for \_ in range(iterations):

            # Используем трапециевидное приближение интеграла от f на шаге

            integral = h \* (f(x[i], y[i]) + f(x[i+1], y\_next))/2

            y\_next = y[i] + integral

        y[i+1] = y\_next

    return x, y

# --- Решения всеми методами ---

x\_euler, y\_euler = euler(f, a, b, y0, h)

x\_ieuler, y\_ieuler = improved\_euler(f, a, b, y0, h)

x\_ieuler\_h2, y\_ieuler\_h2 = improved\_euler(f, a, b, y0, h/2) # для оценки ошибки

x\_rk, y\_rk = runge\_kutta(f, a, b, y0, h)

x\_rk\_2h, y\_rk\_2h = runge\_kutta(f, a, b, y0, 2\*h)           # для оценки ошибки

x\_adams, y\_adams = adams(f, a, b, y0, h)

x\_picard, y\_picard = picard(f, a, b, y0, h)

# --- Оценка погрешностей ---

# Сравниваем решения с разным шагом или разными методами

error\_euler = np.abs(y\_euler - y\_ieuler\_h2[::2])       # ошибка метода Эйлера

error\_ieuler = np.abs(y\_ieuler - y\_ieuler\_h2[::2])     # ошибка улучшенного Эйлера

error\_rk = np.abs(y\_rk[::2] - y\_rk\_2h)/15              # приближённая ошибка Рунге-Кутты 4-го порядка (через 2 шага)

error\_adams\_rk = np.abs(y\_adams[-1] - y\_rk[-1])        # разница между Адамсом и Рунге-Куттой

error\_picard\_rk = np.abs(y\_picard[-1] - y\_rk[-1])      # разница между Пикаром и Рунге-Куттой

# --- Вывод результатов ---

print("=== Результаты решения задачи Коши ===")

print(f"Метод Эйлера (h={h}): y({b}) = {y\_euler[-1]:.5f}")

print(f"Усовершенств. Эйлера (h={h}): y({b}) = {y\_ieuler[-1]:.5f}")

print(f"Усовершенств. Эйлера (h={h/2}): y({b}) = {y\_ieuler\_h2[-1]:.5f}")

print(f"Рунге-Кутты (h={h}): y({b}) = {y\_rk[-1]:.5f}")

print(f"Адамса (h={h}): y({b}) = {y\_adams[-1]:.5f}")

print(f"Пикара (h={h}): y({b}) = {y\_picard[-1]:.5f}")

print("\n=== Оценка погрешностей ===")

print(f"Погрешность метода Эйлера: {error\_euler[-1]:.5f}")

print(f"Погрешность усовершенств. метода: {error\_ieuler[-1]:.5f}")

print(f"Оценка погрешности Рунге-Кутты: {error\_rk[-1]:.5f}")

print(f"Разница Адамса и Рунге-Кутты: {error\_adams\_rk:.5f}")

print(f"Разница Пикара и Рунге-Кутты: {error\_picard\_rk:.5f}")

# --- График решений ---

plt.figure(figsize=(12, 8))

plt.plot(x\_euler, y\_euler, 'b-', label=f'Эйлера (y(2)={y\_euler[-1]:.5f})')

plt.plot(x\_ieuler, y\_ieuler, 'g--', label=f'Усовершенств. Эйлера (y(2)={y\_ieuler[-1]:.5f})')

plt.plot(x\_rk, y\_rk, 'r-.', label=f'Рунге-Кутты (y(2)={y\_rk[-1]:.5f})')

plt.plot(x\_adams, y\_adams, 'c:', label=f'Адамса (y(2)={y\_adams[-1]:.5f})')

plt.plot(x\_picard, y\_picard, 'm--', label=f'Пикара (y(2)={y\_picard[-1]:.5f})')

plt.xlabel('x')

plt.ylabel('y')

plt.title('Решение ДУ: y\' = 1/(1+x³y) + 2y, y(1.5)=2.1')

plt.legend()

plt.grid(True)

plt.show()