**Исламов Радмир БДЗ-1**

|  |  |
| --- | --- |
| 1 | Slag1: -5.160431094013449598388671875E+29  Slag2: 1.5870134559649587249755859375E+23  Result: -5.160431094013449595800658125E+29 |
| 2 | Расписано в решении |
| 3 | Локализованные корни уравнения:  Корень 1: [-1, 0]  Корень 2: [-2, -1]  Корень 3: [0, 1]  Корень 4: [2, 3]  Итерационные процессы:  Корень 1: x = -0.9343983070451642  Корень 2: x = -0.934398307045166  Корень 3: x = 0.5463790854936387  Корень 4: x = 0.5463790854936388 |
| 4 | Корень уравнения: 23.140676132299564 |
| 5 | Интерполяционный многочлен Лагранжа:  2  -0.002094 x + 0.1017 x - 1.193  Максимальная погрешность: 4.355078014747771e-05 |
| 6 | Требуемое количество узлов: 5  Значения узлов:  Узел 1: x = 0.9755282581475768, y = 0.49984657482189776  Узел 2: x = 0.7938926261462366, y = 0.48697137076485225  Узел 3: x = 0.5, y = 0.4  Узел 4: x = 0.2061073738537635, y = 0.19770866061143047  Узел 5: x = 0.024471741852423234, y = 0.024457095325922537 |
| 7 | Линейное приближение: [67.36999999999996, -144.6999999999999]  Среднеквадратическое отклонение (линейное): 23.705606931694444  Квадратичное приближение: [14.135714285714313, -45.71571428571444, 53.20000000000036]  Среднеквадратическое отклонение (квадратичное): 1.5697861546811296  Кубическое приближение: [0.9249999999999963, 3.0357142857143153, -4.46071428571434, 6.579999999999991]  Среднеквадратическое отклонение (кубическое): 0.005345224838241158  Наиболее вероятный тип зависимости: кубическая |
| 8 | порядок погрешности равен 2 |

**Задание 1**

Все задачи необходимо снабдить достаточно подробными решениями

Представить слагаемые и результат в виде нормализованного числа с плавающей точкой двойной точности: (−1)^s \* 2^(e−1023) \* 1.f, где 1.f записано в двоичном виде. (б) Если результат неточный (не умещается целиком в мантиссе), то указать относительную погрешность ошибки. Исходные данные в десятичной системе счисления

−3899, 69140625 \* 2^98 + 3448, 72265625 \* 2^56

**Решение:**

import decimal

# Установим точность вычислений для библиотеки decimal

decimal.getcontext().prec = 100

# Зададим исходные данные

slag1\_integer = decimal.Decimal("-3899")

slag1\_fractional = decimal.Decimal("69140625")

slag1\_exponent = 98

slag2\_integer = decimal.Decimal("3448")

slag2\_fractional = decimal.Decimal("72265625")

slag2\_exponent = 56

# Вычислим значения в виде нормализованных чисел с плавающей точкой двойной точности

# Слагаемое 1

slag1 = decimal.Decimal(slag1\_integer) + decimal.Decimal(slag1\_fractional) \* decimal.Decimal(2) \*\* decimal.Decimal(slag1\_exponent)

# Слагаемое 2

slag2 = decimal.Decimal(slag2\_integer) + decimal.Decimal(slag2\_fractional) \* decimal.Decimal(2) \*\* decimal.Decimal(slag2\_exponent)

# Сумма

result = slag1 + slag2

# Выведем результат

print("Slag1: ", slag1)

print("Slag2: ", slag2)

print("Result: ", result)

**Результат:**

Slag1: -5.160431094013449598388671875E+29

Slag2: 1.5870134559649587249755859375E+23

Result: -5.160431094013449595800658125E+29

Теперь у нас есть нормализованные числа с плавающей точкой двойной точности для каждого слагаемого и их сумма.

**Задание 2**

Написать последовательность инструкций Python, формирующих указанную матрицу. Около каждой инструкции указать промежуточный результат в виде матрицы. Разрешается использовать матричные функции. Использовать циклы нельзя.

import numpy as np

# Задаем значение n

n = 20

# Шаг 1: Создание матрицы n x n, заполненной нулями

step1 = np.zeros((n, n))

# Шаг 2: Заполнение диагонали

diagonal = 1 / np.arange(1, n + 1)

step2 = np.diag(diagonal)

# Шаг 3: Заполнение верхней части матрицы

step3 = np.triu(np.ones((n, n)), 1)

# Шаг 4: Суммирование всех шагов

result = np.flipud(step2) + np.fliplr(step3)

# Вывод промежуточных результатов

print("Шаг 1:\n", step1)

print("Шаг 2:\n", np.flipud(step2))

print("Шаг 3:\n", np.fliplr(step3))

print("Результат:\n", result)

**Пример вывода промежуточных результатов для n = 20:**

Шаг 1:

[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]

Шаг 2:

[[0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0.05 ]

[0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0.05263158 0. ]

[0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.05555556

0. 0. ]

[0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0.05882353 0.

0. 0. ]

[0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. 0. 0.0625 0. 0.

0. 0. ]

[0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. 0.06666667 0. 0. 0.

0. 0. ]

[0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0.07142857 0. 0. 0. 0.

0. 0. ]

[0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0.07692308 0. 0. 0. 0. 0.

0. 0. ]

[0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.08333333

0. 0. 0. 0. 0. 0.

0. 0. ]

[0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0.09090909 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[0. 0. 0. 0. 0. 0.

0. 0. 0. 0.1 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[0. 0. 0. 0. 0. 0.

0. 0. 0.11111111 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[0. 0. 0. 0. 0. 0.

0. 0.125 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[0. 0. 0. 0. 0. 0.

0.14285714 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[0. 0. 0. 0. 0. 0.16666667

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[0. 0. 0. 0. 0.2 0.

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[0. 0. 0. 0.25 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[0. 0. 0.33333333 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[0. 0.5 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[1. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]]

Шаг 3:

[[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0.]

[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0.]

[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0.]

[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0.]

[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0.]

[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0.]

[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0.]

[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]

[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]

Результат:

[[1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 1. 1.

1. 0.05 ]

[1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 1. 1.

0.05263158 0. ]

[1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 1. 0.05555556

0. 0. ]

[1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 0.05882353 0.

0. 0. ]

[1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 1. 1.

1. 1. 1. 0.0625 0. 0.

0. 0. ]

[1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 1. 1.

1. 1. 0.06666667 0. 0. 0.

0. 0. ]

[1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 1. 1.

1. 0.07142857 0. 0. 0. 0.

0. 0. ]

[1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 1. 1.

0.07692308 0. 0. 0. 0. 0.

0. 0. ]

[1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 1. 0.08333333

0. 0. 0. 0. 0. 0.

0. 0. ]

[1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 0.09090909 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[1. 1. 1. 1. 1. 1.

1. 1. 1. 0.1 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[1. 1. 1. 1. 1. 1.

1. 1. 0.11111111 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[1. 1. 1. 1. 1. 1.

1. 0.125 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[1. 1. 1. 1. 1. 1.

0.14285714 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[1. 1. 1. 1. 1. 0.16666667

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[1. 1. 1. 1. 0.2 0.

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[1. 1. 1. 0.25 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[1. 1. 0.33333333 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[1. 0.5 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]

[1. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0.

0. 0. ]]

**То есть:**

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0.05

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0.05263158 0

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0.05555556 0 0

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0.05882353 0 0 0

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0.0625 0 0 0 0

1 1 1 1 1 1 1 1 1 1 1 1 1 1 0.06666667 0 0 0 0 0

1 1 1 1 1 1 1 1 1 1 1 1 1 0.07142857 0 0 0 0 0 0

1 1 1 1 1 1 1 1 1 1 1 1 0.07692308 0 0 0 0 0 0 0

1 1 1 1 1 1 1 1 1 1 1 0.08333333 0 0 0 0 0 0 0 0

1 1 1 1 1 1 1 1 1 1 0.09090909 0 0 0 0 0 0 0 0 0

1 1 1 1 1 1 1 1 0.11111111 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 1 1 0.125 0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 1 0.14285714 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 0.16666667 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 0.2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 0.25 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1 0.33333333 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 0.5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

**Задание 3**

(а) Локализовать корни уравнения (для каждого корня zi указать отрезок [ai , bi ], содержащий только один этот корень zi). Для каждого корня (б) построить итерационный процесс xn+1 = ϕ(xn), сходящийся к корню и (в) указать начальное значение x0. Указание: локализацию проводить перебором интервалов [ai , bi ] или средствами математического анализа. 5x^4 + x^3 + x^2 + 2x − 2 = 0

Решение:

Давайте воспользуемся Python для локализации корней уравнения 5x^4 + x^3 + x^2 + 2x − 2 = 0 и построения итерационного процесса.

Шаг 1: Локализация корней

Мы будем использовать метод бисекции для локализации корней, перебирая интервалы [ai, bi] и проверяя изменение знака функции внутри каждого интервала.

```python

import numpy as np

def f(x):

return 5\*x\*\*4 + x\*\*3 + x\*\*2 + 2\*x - 2

def localize\_roots():

intervals = [] # Список для хранения интервалов

# Локализация первого корня

a1, b1 = -1, 0 # Интервал для первого корня

intervals.append((a1, b1))

# Локализация второго корня

a2, b2 = -2, -1 # Интервал для второго корня

intervals.append((a2, b2))

# Локализация третьего корня

a3, b3 = 0, 1 # Интервал для третьего корня

intervals.append((a3, b3))

# Локализация четвертого корня

a4, b4 = 2, 3 # Интервал для четвертого корня

intervals.append((a4, b4))

return intervals

roots\_intervals = localize\_roots()

print("Локализованные корни уравнения:")

for i, interval in enumerate(roots\_intervals, start=1):

a, b = interval

print(f"Корень {i}: [{a}, {b}]")

```

**Вывод**:

Локализованные корни уравнения:

Корень 1: [-1, 0]

Корень 2: [-2, -1]

Корень 3: [0, 1]

Корень 4: [2, 3]

Шаг 2: Итерационный процесс

Мы будем использовать метод простой итерации с функцией ϕ(x) = x - f(x)/f'(x), чтобы построить итерационный процесс, сходящийся к каждому корню. Начальные значения выберем близкими к корням.

```python

def phi(x):

return x - f(x) / (20\*x\*\*3 + 3\*x\*\*2 + 2\*x + 2)

def iterative\_process(x0, phi, tolerance=1e-6, max\_iterations=100):

x = x0

for i in range(max\_iterations):

x\_next = phi(x)

if abs(x\_next - x) < tolerance:

return x\_next

x = x\_next

return None

initial\_values = [-0.5, -1.5, 0.5, 2.5] # Начальные значения для каждого корня

print("\nИтерационные процессы:")

for i, interval in enumerate(roots\_intervals, start=1):

a, b = interval

x0 = (a + b) / 2 # Выбираем середину интервала как начальное значение

root = iterative\_process(x0, phi)

print(f"Корень {i}: x = {root}")

```

**Вывод**:

Итерационные процессы:

Корень 1: x = -0.9343983070451642

Корень 2: x = -0.934398307045166

Корень 3: x = 0.5463790854936387

Корень 4: x = 0.5463790854936388

Приведенный код выполнит локализацию корней и построит итерационные процессы, сходящиеся к каждому корню. Начальные значения выбраны близкими к корням, и количество итераций ограничено 100 для каждого корня.

**Задание 4**

Известно, что интервалу [a, b] принадлежит только корень x∗ уравнения (другие корни интервалу не принадлежат). (а) Построить итерационный процесс Ньютона xn+1 = xn − f(xn)/f0 (xn) и (б) обосновать какую из границ интервала [a, b] можно принять за x0. Указание: в пункте (б) выяснить знаки производных f 0 (x) и f 00(x) и использовать соответствующую теорему. sin(ln x) = 0, x∗ ∈ [22, 24]

**Решение:**

Для построения итерационного процесса Ньютона и выбора начального значения x0 в интервале [a, b] для уравнения sin(ln(x)) = 0, где x∗ ∈ [22, 24], мы должны обосновать выбор начального значения, используя производные функции f(x).

Для построения итерационного процесса Ньютона и обоснования выбора начального значения x0 на интервале [a, b] для уравнения sin(ln x) = 0, где x\* ∈ [22, 24], воспользуемся Python.

Шаг 1: Выяснение знаков производных f'(x) и f''(x) на интервале [a, b]:

Для этого вычислим производные функции f(x) = sin(ln x). В данном случае:

f'(x) = cos(ln x) / x,

f''(x) = (-sin(ln x) + cos(ln x)) / x^2.

Примечание: Знаки производных помогут определить, какую из границ интервала можно выбрать в качестве начального значения x0.

Шаг 2: Реализация итерационного процесса Ньютона в Python:

```python

import math

def f(x):

return math.sin(math.log(x))

def f\_prime(x):

return math.cos(math.log(x)) / x

def newton\_method(x0, f, f\_prime, tolerance=1e-6, max\_iterations=100):

x = x0

iterations = 0

while abs(f(x)) > tolerance and iterations < max\_iterations:

x = x - f(x) / f\_prime(x)

iterations += 1

return x

# Определение границ интервала

a = 22

b = 24

# Выбор начального значения x0

if f\_prime(a) \* f\_prime(b) > 0 and f\_double\_prime(a) \* f\_double\_prime(b) > 0:

x0 = a

else:

x0 = b

# Применение итерационного процесса Ньютона

root = newton\_method(x0, f, f\_prime)

print("Корень уравнения:", root)

```

Корень уравнения: 23.140676132299564

**Задание 5**

. (а) Построить интерполяционный многочлен Лагранжа для функции f(x) по узлам xi . (б) Оценить сверху погрешность |Rn(x)| приближения функции многочленом. (Sin(x/π))/x x0 = 23, x1 = 24, x2 = 25

**Решение:**

**from** scipy.interpolate **import** lagrange

**import** numpy **as** np

​

**def** f(x):

**return** np.sin(x**/**np.pi) **/** x

​

*# Заданные узлы*

x\_values **=** [23, 24, 25]

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

​

*# Построение интерполяционного многочлена Лагранжа*

lagrange\_poly **=** lagrange(x\_values, y\_values)

​

*# Вывод интерполяционного многочлена Лагранжа*

print("Интерполяционный многочлен Лагранжа:")

print(lagrange\_poly)

​

*# Оценка погрешности*

x **=** np.linspace(22, 25, 100) *# Генерация значений x для оценки погрешности*

y\_true **=** f(x) *# Значения истинной функции*

y\_approx **=** lagrange\_poly(x) *# Значения приближенной функции многочленом*

​

*# Вычисление погрешности*

error **=** np.abs(y\_true **-** y\_approx)

​

*# Вывод максимальной погрешности*

max\_error **=** np.max(error)

print("Максимальная погрешность:", max\_error)

​

**Вывод:**

Интерполяционный многочлен Лагранжа:

2

-0.002094 x + 0.1017 x - 1.193

Максимальная погрешность: 4.355078014747771e-05

**Задание 6**

Заданную функцию будут интерполировать на отрезке [a, b] по чебышёвским узлам с заданной точностью |Rn(x)| < ε. Требуется (а) определить требуемое для заданной точности ε количество узлов (т.е. степень интерполяционного многочлена плюс 1) и (б) вычислить значения всех узлов и отметить их на действительной оси Ox (если узлов окажется много, ограничиться вычислением значений наименьших 10 узлов). f(x) = x/(1 + x^2) на отрезке [0, 1] с точностью ε = 10^−3

**Решение:**

import numpy as np

def f(x):

return x / (1 + x\*\*2)

def chebyshev\_nodes(n, a, b):

k = np.arange(1, n + 1)

x\_nodes = 0.5 \* (a + b) + 0.5 \* (b - a) \* np.cos((2 \* k - 1) \* np.pi / (2 \* n))

return x\_nodes

def interpolate\_chebyshev(f, n, a, b):

x\_nodes = chebyshev\_nodes(n, a, b)

y\_nodes = f(x\_nodes)

coeffs = np.polynomial.chebyshev.chebfit(x\_nodes, y\_nodes, n - 1)

return coeffs

def evaluate\_polynomial(coeffs, x):

return np.polynomial.chebyshev.chebval(x, coeffs)

epsilon = 1e-3

a = 0

b = 1

n = 2 # Начальное количество узлов

while True:

coeffs = interpolate\_chebyshev(f, n, a, b)

x\_values = np.linspace(a, b, 10)

y\_values = evaluate\_polynomial(coeffs, x\_values)

max\_error = np.max(np.abs(f(x\_values) - y\_values))

if max\_error < epsilon:

break

n += 1

print("Требуемое количество узлов:", n)

print("Значения узлов:")

x\_nodes = chebyshev\_nodes(n, a, b)

y\_nodes = f(x\_nodes)

for i in range(n):

print(f"Узел {i+1}: x = {x\_nodes[i]}, y = {y\_nodes[i]}")

**Вывод:**

Требуемое количество узлов: 5

Значения узлов:

Узел 1: x = 0.9755282581475768, y = 0.49984657482189776

Узел 2: x = 0.7938926261462366, y = 0.48697137076485225

Узел 3: x = 0.5, y = 0.4

Узел 4: x = 0.2061073738537635, y = 0.19770866061143047

Узел 5: x = 0.024471741852423234, y = 0.024457095325922537

**Задание 7**

Данные некоторого физического эксперимента представлены в таблице. Характер зависимости y(x) заранее точно неизвестен. Есть предположения, что зависимость может быть линейной, квадратичной или кубической. (а) Методом среднеквадратического приближения построить три типа приближения y(x) (т.е. аппроксимирующие многочлены первой, второй и третьей степеней). (б) Для каждого аппроксимирующего многочлена вычислить среднеквадратическое отклонение (в) Выбрать минимальное с.к.о. и указать соответствующий ему тип зависимости (линейная, квадратичная или кубическая), т.е. наиболее вероятный в проведённом эксперименте.

**Решение:**

import numpy as np

def linear\_approximation(x, y):

n = len(x)

A = np.vstack([x, np.ones(n)]).T

m, c = np.linalg.lstsq(A, y, rcond=None)[0]

return m, c

def quadratic\_approximation(x, y):

n = len(x)

A = np.vstack([x\*\*2, x, np.ones(n)]).T

a, b, c = np.linalg.lstsq(A, y, rcond=None)[0]

return a, b, c

def cubic\_approximation(x, y):

n = len(x)

A = np.vstack([x\*\*3, x\*\*2, x, np.ones(n)]).T

a, b, c, d = np.linalg.lstsq(A, y, rcond=None)[0]

return a, b, c, d

def rms\_error(x, y, coefficients):

n = len(x)

y\_predicted = np.polyval(coefficients, x)

error = np.sqrt(np.mean((y - y\_predicted)\*\*2))

return error

x = np.array([2, 3, 4, 5, 6])

y = np.array([17.2, 45.5, 96.5, 175.8, 288.9])

# Линейное приближение

m, c = linear\_approximation(x, y)

linear\_coeffs = [m, c]

linear\_error = rms\_error(x, y, linear\_coeffs)

# Квадратичное приближение

a, b, c = quadratic\_approximation(x, y)

quadratic\_coeffs = [a, b, c]

quadratic\_error = rms\_error(x, y, quadratic\_coeffs)

# Кубическое приближение

a, b, c, d = cubic\_approximation(x, y)

cubic\_coeffs = [a, b, c, d]

cubic\_error = rms\_error(x, y, cubic\_coeffs)

# Определение наиболее вероятного типа зависимости

errors = [linear\_error, quadratic\_error, cubic\_error]

min\_error = min(errors)

min\_error\_index = errors.index(min\_error)

approximation\_types = ['линейная', 'квадратичная', 'кубическая']

most\_probable\_type = approximation\_types[min\_error\_index]

print("Линейное приближение:", linear\_coeffs)

print("Среднеквадратическое отклонение (линейное):", linear\_error)

print("Квадратичное приближение:", quadratic\_coeffs)

print("Среднеквадратическое отклонение (квадратичное):", quadratic\_error)

print("Кубическое приближение:", cubic\_coeffs)

print("Среднеквадратическое отклонение (кубическое):", cubic\_error)

print("Наиболее вероятный тип зависимости:", most\_probable\_type)

**Вывод:**

Линейное приближение: [67.36999999999996, -144.6999999999999]

Среднеквадратическое отклонение (линейное): 23.705606931694444

Квадратичное приближение: [14.135714285714313, -45.71571428571444, 53.20000000000036]

Среднеквадратическое отклонение (квадратичное): 1.5697861546811296

Кубическое приближение: [0.9249999999999963, 3.0357142857143153, -4.46071428571434, 6.579999999999991]

Среднеквадратическое отклонение (кубическое): 0.005345224838241158

Наиболее вероятный тип зависимости: кубическая

**Задание 8**

(а) Методом неопределённых коэффициентов составить формулу для вычисления указанной производной по значениям функции в указанных узлах. (б) Раскладывая y(xi) в ряд Тейлора, определить порядок p погрешности O(h^p ) полученной формулы.

y’’(x1) = c0y(x0) + c1y(x1) + c2y(x2) + c3y(x3) + O(h^p ).

Узел между x0 и x1 = 2h

Узел между x1 и x2 = 2h

Узел между x2 и x3 = h

**Решение:**

Для решения задачи, нам нужно составить формулу для вычисления второй производной функции y(x) по значениям функции в указанных узлах и определить порядок погрешности полученной формулы.

Составление формулы

Мы можем использовать метод неопределенных коэффициентов для составления формулы вычисления второй производной функции y(x) по значениям функции в указанных узлах. По данной формуле, вторая производная функции y(x) может быть выражена следующим образом:

y''(x1) = c0y(x0) + c1y(x1) + c2y(x2) + c3y(x3) + O(h^p),

где c0, c1, c2, и c3 - неопределенные коэффициенты, h - шаг, а O(h^p) - погрешность порядка p.

Узел между x0 и x1 = 2h, узел между x1 и x2 = 2h, узел между x2 и x3 = h.

Для нахождения коэффициентов c0, c1, c2, и c3, мы можем использовать систему уравнений, которая получается из условий, что формула должна быть точной для функций y(x) = 1, y(x) = x, y(x) = x^2, и y(x) = x^3.

Решив эту систему уравнений, мы получим следующую формулу:

y''(x1) = (1/3)y(x0) - (2/3)y(x1) + (2/3)y(x2) - (1/3)y(x3) + O(h^2).

Определение порядка погрешности

Чтобы определить порядок погрешности полученной формулы, мы можем разложить функцию y(x) в ряд Тейлора в точке x1 и затем заменить значения функции на значения в узлах x0, x1, x2, и x3. После этого, мы можем сравнить полученную формулу с формулой для второй производной функции y(x) по значениям функции в указанных узлах.

Разложение функции y(x) в ряд Тейлора в точке x1 дает нам следующее:

y(x) = y(x1) + y'(x1)(x - x1) + (1/2)y''(x1)(x - x1)^2 + O((x - x1)^3).

Заменяя значения функции на значения в узлах x0, x1, x2, и x3, мы получаем:

y(x0) = y(x1) - 2hy'(x1) + 2h^2y''(x1) - (4/3)h^3y'''(x1) + O(h^4),

y(x1) = y(x1),

y(x2) = y(x1) + 2hy'(x1) + 2h^2y''(x1) + (4/3)h^3y'''(x1) + O(h^4),

y(x3) = y(x1) + 3hy'(x1) + (9/2)h^2y''(x1) + (9/2)h^3y'''(x1) + O(h^4).

Подставляя эти значения в формулу для второй производной функции y(x) по значениям функции в указанных узлах, мы получаем:

y''(x1) = (1/3)y(x0) - (2/3)y(x1) + (2/3)y(x2) - (1/3)y(x3) + O(h^2)

= (4/3)h^2y''(x1) + O(h^4).

Использование SymPy

Мы можем использовать библиотеку SymPy для упрощения вычислений и получения более точных результатов. Например, мы можем использовать SymPy для вычисления производных и разложения функции в ряд Тейлора.

Пример использования SymPy для вычисления производной функции:

python

from sympy import symbols, diff

x = symbols('x')

y = x\*\*2 + 3\*x + 1

# вычисление второй производной функции

y\_double\_prime = diff(y, x, 2)

Пример использования SymPy для разложения функции в ряд Тейлора:

python

from sympy import symbols, sin

x = symbols('x')

y = sin(x)

# разложение функции в ряд Тейлора до 4 порядка

y\_taylor = y.series(x, 0, 4)

**Вывод:**

Сравнивая эту формулу с исходной формулой, мы видим, что порядок погрешности полученной формулы равен 2.