1 程序 1

1.1 题目描述

给定一个函数 $\Psi(x)$，该函数表示一个递归和的形式，形式如下：

$$\Psi(x) = \sum_{k=1}^{\infty} \frac{1}{k(k+x)}$$

需要实现一个算法来计算这个函数的近似值，要求截断误差在 10^{-6} 内，给定一系列输入 x，输出 x 和 $\Psi(x)$ 的值。

1.2 算法描述

本程序使用逐项累加的方式计算 $\Psi(x)$ 的值，直到当前项的绝对值小于指定的误差阈值。算法步骤如下：

1. 初始化变量 $k = 1$ 和 sum_psi = 0.0，分别表示和的当前项数和累计的 $\Psi(x)$ 值。
2. 计算当前项值，term = $\frac{1}{k(k+x)}$，并将其加到 sum_psi 中。
3. 如果当前项的绝对值大于误差阈值，继续计算下一项，更新 k 值。
4. 返回最终的 sum_psi 作为 $\Psi(x)$ 的值。

1.3 程序代码

```
import numpy as np

# 定义计算 Psi(x) 的函数
def compute_psi(x, error_threshold=1e-6):
k = 1  # 初始化项数
sum_psi = 0.0  # 初始化 Psi(x) 的值
term = 1.0  # 初始化项的值, 确保第一次循环进入

# 当当前项的绝对值大于误差阈值时继续累加
while abs(term) > error_threshold:
term = 1 / (k * (k + x))  # 计算当前项
sum_psi += term  # 将当前项加到 Psi(x) 的和中
k += 1  # 增加项数

return sum_psi  # 返回计算的 Psi(x)

# 输入要计算的 x 值列表
```

```
x_values = [0.0, 0.5, 1.0, np.sqrt(2), 10.0, 100.0, 300.0]

# 对每个 x 值计算 Psi(x)
results = {x: compute_psi(x) for x in x_values}

# 输出结果
for x, psi in results.items():
print(f"x = {x:.6f}, Psi(x) = {psi:.6f}")
```

1.4 程序输入输出

输出：

$$x = 0.000000, \Psi(x) = 1.643935$$

$$x = 0.500000, \Psi(x) = 1.226412$$

$$x = 1.000000, \Psi(x) = 0.999001$$

$$x = 1.414214, \Psi(x) = 0.873984$$

$$x = 10.000000, \Psi(x) = 0.291898$$

$$x = 100.000000, \Psi(x) = 0.050875$$

$$x = 300.000000, \Psi(x) = 0.019947$$

2 程序 2

2.1 题目描述

根据美国 1920 年到 1970 年的各年人口数据（见表 A1），使用拉格朗日插值法计算 1910 年、1965 年和 2002 年的估计人口，并分析这些估计值的准确性。已知 1910 年实际人口为 91772000。

2.2 算法描述

拉格朗日插值法是一种通过给定数据点来构造多项式的插值方法。对于给定的六个数据点（1920 年到 1970 年的年份和对应人口），可以通过以下的拉格朗日插值公式进行估算：

$$P(x) = \sum_{i=0}^{5} y_i L_i(x)$$

其中，

$$L_i(x) = \prod_{j=0, j\neq i}^{5} \frac{x - x_j}{x_i - x_j}$$

- x_i 和 y_i 是已知数据点的年份和对应的 population 数据。
- $L_i(x)$ 是拉格朗日基函数。

对于目标年份（例如 1910 年、1965 年、2002 年），通过计算插值多项式 $P(x)$，可以得到这些年份的估计人口。

此外，还需要计算误差系数来估算插值的准确性。误差系数的计算公式为：

$$E(x) = \frac{P(x) - A(x)}{(x - x_0)(x - x_1)\cdots(x - x_5)}$$

其中，$A(x)$ 是实际人口数据。

2.3 程序代码

```
import numpy as np

# 拉格朗日插值的实现
def lagrange_interpolate(x_points, y_points, target_x):
'''进行拉格朗日插值计算'''
x0, x1, x2, x3, x4, x5 = x_points
y0, y1, y2, y3, y4, y5 = y_points

# 定义每个拉格朗日基函数
L0 = lambda x: (x - x1) * (x - x2) * (x - x3) * (x - x4) * (x - x5) / ((x0 -
x1) * (x0 - x2) * (x0 - x3) * (x0 - x4) * (x0 - x5))
L1 = lambda x: (x - x0) * (x - x2) * (x - x3) * (x - x4) * (x - x5) / ((x1 -
x0) * (x1 - x2) * (x1 - x3) * (x1 - x4) * (x1 - x5))
L2 = lambda x: (x - x0) * (x - x1) * (x - x3) * (x - x4) * (x - x5) / ((x2 -
x0) * (x2 - x1) * (x2 - x3) * (x2 - x4) * (x2 - x5))
L3 = lambda x: (x - x0) * (x - x1) * (x - x2) * (x - x4) * (x - x5) / ((x3 -
x0) * (x3 - x1) * (x3 - x2) * (x3 - x4) * (x3 - x5))
L4 = lambda x: (x - x0) * (x - x1) * (x - x2) * (x - x3) * (x - x5) / ((x4 -
x0) * (x4 - x1) * (x4 - x2) * (x4 - x3) * (x4 - x5))
L5 = lambda x: (x - x0) * (x - x1) * (x - x2) * (x - x3) * (x - x4) / ((x5 -
x0) * (x5 - x1) * (x5 - x2) * (x5 - x3) * (x5 - x4))

# 计算插值结果
return y0 * L0(target_x) + y1 * L1(target_x) + y2 * L2(target_x) + y3 *
L3(target_x) + y4 * L4(target_x) + y5 * L5(target_x)
```

```
# 计算误差系数项
def calculate_error_coefficient(x_points, target_x, predicted_y, actual_y):
'''计算误差系数'''
x0, x1, x2, x3, x4, x5 = x_points
delta_y = actual_y - predicted_y
return delta_y / ((target_x - x0) * (target_x - x1) * (target_x - x2) *
(target_x - x3) * (target_x - x4) * (target_x - x5))

# 计算误差
def compute_error(x_points, target_x, error_coefficient):
'''计算指定年份的误差'''
x0, x1, x2, x3, x4, x5 = x_points
return error_coefficient * (target_x - x0) * (target_x - x1) * (target_x - x2)
* (target_x - x3) * (target_x - x4) * (target_x - x5)

# 数据（年份和对应人口数据）
years = [1920, 1930, 1940, 1950, 1960, 1970]
populations = [105711, 123203, 131669, 150697, 179323, 203212]

# 目标年份：1910、1965、2002
target_years = [1910, 1965, 2002]

# 使用拉格朗日插值法估算这些年份的人口
estimated_populations = [lagrange_interpolate(years, populations, year) for
year in target_years]

# 输出估算结果
for target_year, estimated_population in zip(target_years,
estimated_populations):
print(f"预估{target_year}年人口：{estimated_population:.0f}（千人）")

# 实际人口数据
actual_population_1910 = 91772  # 1910年实际人口（千人）

# 计算误差系数
error_coefficient = calculate_error_coefficient(years, target_years[0],
estimated_populations[0], actual_population_1910)

# 计算1965年和2002年的误差
error_1965 = compute_error(years, target_years[1], error_coefficient)
error_2002 = compute_error(years, target_years[2], error_coefficient)

# 输出误差
print(f"1965年误差：{error_1965:.0f}（千人）")
print(f"2002年误差：{error_2002:.0f}（千人）")
```

2.4 程序输入输出

输出：

预估 1910 年人口: 31872 (千人)

预估 1965 年人口: 193082 (千人)

预估 2002 年人口: 26139 (千人)

1965 年误差: −1228 (千人)

2002 年误差: 2128310 (千人)

3 程序 3

3.1 题目描述

根据表 A1 的数据点（年份，人口/千人）：

$$((x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n))$$

通过牛顿插值法估计 1965 年与 2012 年的人口数。

3.2 算法描述

牛顿插值法通过差商表逐步计算插值多项式，利用已知数据点计算目标点的估算值。其算法过程如下：

1. 计算差商表：

$$f[x_0] = y_0$$
$$f[x_1, x_0] = \frac{f[x_1] - f[x_0]}{x_1 - x_0}$$
$$f[x_2, x_1, x_0] = \frac{f[x_2] - f[x_1]}{x_2 - x_1} - \frac{f[x_1] - f[x_0]}{x_1 - x_0}$$

   以此类推，计算差商表。

2. 使用牛顿插值公式计算目标点的估算值：

$$P(x) = f[x_0] + (x - x_0)f[x_1, x_0] + (x - x_0)(x - x_1)f[x_2, x_1, x_0] + \ldots$$

3.3 程序代码

```python

import numpy as np
def newton_interpolation(X, Y, x):
'''牛顿插值法实现'''
n = len(X)
# 初始化差商表
f = np.zeros((n, n))  # 使用NumPy来初始化差商表
for i in range(n):
f[i][0] = Y[i]  # 第一列是Y值，即原始人口数据

# 计算差商
for j in range(1, n):
for i in range(n - j):
f[i][j] = (f[i + 1][j - 1] - f[i][j - 1]) / (X[i + j] - X[i])

# 使用牛顿插值公式计算目标年份的估算人口
result = f[0][n - 1]
for i in range(n - 2, -1, -1):
result = f[0][i] + (x - X[i]) * result

return result

# 给定的年份和对应的人口数据
X = np.array([1920, 1930, 1940, 1950, 1960, 1970])
Y = np.array([105711, 123203, 131669, 150697, 179323, 203212])

# 估算1965年和2002年的人口
x0, x1 = 1965, 2002
y0, y1 = newton_interpolation(X, Y, x0), newton_interpolation(X, Y, x1)

# 输出结果
print(f"1965年估算人口：{y0:.0f}（千人)")
print(f"2002年估算人口：{y1:.0f}（千人)")
```

3.4 程序输入输出

输出：

1965 年估算人口: 193082 (千人)

2002 年估算人口: 26139 (千人)

4 程序 7

4.1 题目描述

给定 $n+1$ 个插值点和一阶导数的端点值 (m_0, m_n)，用三次样条插值法构造函数 $S(x)$，并求在给定点 x 处 $S(x)$ 的值。

4.2 算法描述

本题使用**三次样条插值**方法来计算函数值。具体地，三次样条插值法构造一个连续的三次多项式来拟合数据，并且要求在端点处满足一阶导数条件。对于给定的插值点 $(x_i, f(x_i))$，$i = 0, 1, 2, \ldots, n$，以及端点的导数值 m_0 和 m_n，我们使用三次样条插值的数学表达式：

$$S(x) = \left\{ f_0 + m_0(x - x_0) + \frac{(x-x_0)^2}{2}\left(f_0'' + \frac{f_0''-f_1''}{x_1-x_0}\right), \quad x_0 \le x \le x_1 \right.$$

其中，所有中间区间 $S(x)$ 由类似的三次多项式构成，并且满足平滑性和导数连续性。

4.3 程序代码

```python
import numpy as np
from scipy.interpolate import CubicSpline

def cubic_spline_interpolation(x_values, y_values, m_0, m_n, x_eval):
"""
 使用三次样条插值并考虑边界条件（端点导数）。

 参数：
x_values (array-like): 插值点的 x 值
y_values (array-like): 插值点的 y 值
m_0 (float): 在 x_0 处的导数值
m_n (float): 在 x_n 处的导数值
x_eval (float): 要计算 S(x) 的 x 值

 返回：
float: 在 x_eval 处的 S(x) 的值
"""
# 构造三次样条插值，考虑边界导数条件
cs = CubicSpline(x_values, y_values, bc_type=((1, m_0), (1, m_n)))

# 计算 x_eval 处的 S(x) 的值
return cs(x_eval)

if __name__ == "__main__":
```

```python
# 输入插值点数 n
n = int(input("插值点数: "))

# 输入插值点
x_values = []
y_values = []
for _ in range(n + 1):
x, y = map(float, input("x y:").split())
x_values.append(x)
y_values.append(y)

# 输入一阶导数的端点值
m_0 = float(input("在 x_0 处的导数值 m_0: "))
m_n = float(input("在 x_n 处的导数值 m_n: "))

# 输入要计算的函数点 x_eval
x_eval = float(input("要计算的函数点: "))

# 调用函数进行计算
s_x = cubic_spline_interpolation(x_values, y_values, m_0, m_n, x_eval)

# 输出结果
print(f"x = {x_eval} 处的 S(x) 的值为: {s_x:.6f}")
```

4.4 程序输入输出

示例输入：

插值点数：3

x y:

$$\begin{array}{cc} 0 & 0 \\ 1 & 1 \\ 2 & 0 \\ 3 & -1 \end{array}$$

$$\text{在}x_0\text{ 处的导数值}m_0 : 1$$

$$\text{在}x_n\text{ 处的导数值}m_n : -1$$

要计算的函数点: 1.5

示例输出：

$$\mathrm{x} = 1.5 \text{ 处的}S(x)\text{ 的值为: } 0.666667$$

5 程序 10

5.1 题目描述

使用 Newton 迭代法求解非线性方程组：

$$\begin{cases} f(x) = x^2 + y^2 - 1 = 0 \\ g(x) = x^3 - y = 0 \end{cases}$$

给定初始点 $(x_0, y_0) = (0.8, 0.6)$，使用误差控制条件：

$$\max(|x_k - x_{k-1}|, |y_k - y_{k-1}|) \leq 10^{-5}$$

5.2 算法描述

1. **初始化**：选择初始猜测点 (x_0, y_0)，并设定容许的误差阈值。
2. **迭代过程**：使用 Newton 迭代法的公式：

$$\mathbf{J}(x_k, y_k) \cdot \boldsymbol{\Delta} = -\mathbf{F}(x_k, y_k)$$

   其中，$\mathbf{J}(x_k, y_k)$ 是雅可比矩阵，$\mathbf{F}(x_k, y_k)$ 是方程组的函数值，$\boldsymbol{\Delta} = (\Delta x_k, \Delta y_k)$ 是更新量。

3. **更新**：根据迭代结果更新 x_k 和 y_k，直到满足误差控制条件或达到最大迭代次数。
4. **收敛性检查**：当 $\max(|\Delta x_k|, |\Delta y_k|)$ 小于给定的误差阈值时，停止迭代，输出结果。

5.3 程序代码

```python
import numpy as np

# 定义方程 f 和 g
def f(x, y):
return x**2 + y**2 - 1

def g(x, y):
return x**3 - y

# 定义雅可比矩阵
def jacobian(x, y):
return np.array([[2 * x, 2 * y],
[3 * x**2, -1]])
```

```
# Newton 迭代法
def newton_method(x0, y0, tol=1e-5, max_iter=100):
x, y = x0, y0
for i in range(max_iter):
# 计算函数值和雅可比矩阵
F = np.array([f(x, y), g(x, y)])
J = jacobian(x, y)

# 解线性方程 J * delta = -F, 求出 delta
delta = np.linalg.solve(J, -F)

# 更新 x 和 y
x += delta[0]
y += delta[1]

# 打印当前迭代步的结果
print(f"Iteration {i+1}: x = {x}, y = {y}")

# 检查收敛条件
if max(abs(delta[0]), abs(delta[1])) < tol:
print("Converged.")
return i + 1, x, y

print("Maximum iterations reached without convergence.")
return max_iter, x, y

if __name__ == "__main__":
# 输入初始点 (x0, y0) 和精度控制值 tol
x0, y0 = map(float, input("输入初始点 (x0, y0): ").split())
tol = float(input("输入精度控制值 e: "))

# 调用 Newton 方法求解
k, x, y = newton_method(x0, y0, tol)

# 输出结果
print(f"迭代次数 k = {k}")
print(f"第 {k} 步的迭代解: x = {x:.6f}, y = {y:.6f}")
```

5.4 程序输入输出

输入初始点 (x_0, y_0)：0.8, 0.6 输入精度控制值 e：$1e-5$

迭代 1: $x = 0.8270491803278689, y = 0.5639344262295083$

迭代 2: $x = 0.8260323731676462, y = 0.5636236767037873$

迭代 3: $x = 0.8260313576552345, y = 0.5636241621608473$

收敛。

迭代次数 $k = 3$

第 3 步的迭代解: $x = 0.826031, y = 0.563624$