

# 计算方法实验

2022 年 4 月 19 日

数据显示结果已保留 4 位小数。

## 1 实验题目 1：拉格朗日 (Lagrange) 插值

### 1.1 问题分析

准确描述并总结出实验题目 (摘要)，并准确分析原题的目的和意义。

#### 1.1.1 方法概要

给定平面上  $n+1$  个不同的数据点  $(x_k, f(x_k)), k=0, 1, \dots, n, x_i \neq x_j, i \neq j$  则满足条件

$$P_n(x_k) = f(x_k), k=0, 1, \dots, n$$

的  $n$  次拉格朗日插值多项式

$$P_n(x) = \sum_{k=0}^n f(x_k) l_k(x)$$

是存在唯一的。若  $x_k \in [a, b], k=0, 1, \dots, n$ ，且函数  $f(x)$  充分光滑，则当  $x \in [a, b]$  时，有误差估计式

$$f(x) - P_n(x) = \frac{f^{n+1}(\xi)}{(n+1)!} (x-x_0)(x-x_1) \cdots (x-x_n), \xi \in [a, b]$$

#### 1.1.2 实验目的

利用拉格朗日插值多项式  $P_n(x)$  求  $f(x)$  的近似值。

输入：  $n+1$  个数据点  $x_k, f(x_k), k=0, 1, \dots, n$ 、插值点  $x$

输出:  $f(x)$  在插值点  $x$  的近似值  $P_n(x)$

## 1.2 数学原理

数学原理表达清晰且书写准确。

### 1.2.1 证明 $P_n(x)$ 存在且唯一

证明: 使用归纳法证明。

当  $n = 0$ , 一定存在  $P_0(x) = C = f_0(x)$  满足要求。

假设当  $n = k - 1$  时, 存在满足要求的  $P_{k-1}(x)$ , 则当  $n = k$ , 有

$$P_k(x) = P_{k-1}(x) + c(x - x_0)(x - x_1) \cdots (x - x_{k-1})(x - x_k), \text{ } c \text{ 为系数}$$

则  $\because P_n(x_n) = f(x_n), \therefore$  参数  $c$  是可求的, 故  $P_n(x)$  是存在的。

由多项式基本定理,  $\because P_n(x)$  的次数  $\leq n, \therefore P_n(x)$  是唯一存在的。

### 1.2.2 计算方法

对平面上  $n + 1$  个点  $(x_k, f(x_k)), k = 0, 1, \cdots, n, x_i \neq x_j, i \neq j$  定义  $n$  次多项式:

$$L_k(x) = \frac{(x - x_0)(x - x_1) \cdots (x - x_{k-1})(x - x_{k+1}) \cdots (x - x_n)}{(x_k - x_0)(x_k - x_1) \cdots (x_k - x_{k-1})(x_k - x_{k+1}) \cdots (x_k - x_n)}$$

则  $L_k(x_k) = 1, L_k(x_m) = 0, m \neq k$ 。

定义:

$$P_n(x) = f(x_0)L_0(x) + f(x_1)L_1(x) + \cdots + f(x_n)L_n(x) = \sum_{k=0}^n f(x_k)L_k(x)$$

为  $f(x)$  的  $n$  次拉格朗日插值多项式。

## 1.3 程序设计流程

编译通过, 根据输入能得到正确输出。

```
[14]: # 引入需要的包
import numpy as np
```

```
from pandas import DataFrame
from matplotlib import pyplot as plt
```

```
[15]: #  $L_k(x)$ 
def L(k, x_list: np.ndarray):
    def l(x: float):
        x_k = x_list[k]
        slice = np.array([*x_list[:k], *x_list[k+1:]])
        repeat = np.array([x, ] * (len(x_list) - 1))
        repeat_k = np.array([x_k, ] * (len(x_list) - 1))
        return np.prod(repeat - slice) / np.prod(repeat_k - slice)
    return l

# 拉格朗日多项式  $P_n(x)$ 
def P(f, x_list: np.ndarray, x: float):
    return np.sum([f(x_list[k]) * L(k, x_list)(x) for k in range(len(x_list))])
```

### 1.3.1 问题一

```
[16]: def problem1():
    print("问题 1: 拉格朗日插值多项式的次数 n 越大越好吗? ")

    def sub(targets_x, targets_n, solve):
        return DataFrame({n: {x: solve(n, x) for x in targets_x} for n in
        ↪targets_n}).round(4)

    print("(1) 考虑  $f(x) = 1 / (1 + x^2)$  in  $[-5, 5]$ ")
    targets_x_1 = [0.75, 1.75, 2.75, 3.75, 4.75]
    targets_n = [5, 10, 20]
    print(sub(
        targets_x=targets_x_1,
        targets_n=targets_n,
        solve=lambda n_i, x: P(
            lambda x_i: 1 / (1 + x_i ** 2), np.linspace(-5, 5, n_i), x
        )
    ))
```

```

print("差值: ")
print(sub(
    targets_x=targets_x_1,
    targets_n=targets_n,
    solve=lambda n_i, x: ((1 / (1 + x ** 2)) - P(
        lambda x_i: 1 / (1 + x_i ** 2), np.linspace(-5, 5, n_i), x
    ))
))

print("(2) 考虑  $f(x) = e^x$  in  $[-1, 1]$ ")
targets_x_2 = [-0.95, -0.05, 0.05, 0.95]
print(sub(
    targets_x=targets_x_2,
    targets_n=targets_n,
    solve=lambda n_i, x: P(lambda x_i: np.e ** x_i,
        np.linspace(-1, 1, n_i), x)
))

print("差值: ")
print(sub(
    targets_x=targets_x_2,
    targets_n=targets_n,
    solve=lambda n_i, x: (np.e ** x - P(lambda x_i: np.e ** x_i,
        np.linspace(-1, 1, n_i), x))
))

print("画出两个函数以及其拉格朗日多项式的图像: ")
slice_fluent_size = 1000
x_linespace_2 = np.linspace(-1, 1, slice_fluent_size)
x_linespace_10 = np.linspace(-5, 5, slice_fluent_size)
y1 = 1 / (1 + x_linespace_10**2)
Ls1 = {n_i: np.array([P(lambda x_i: np.e ** x_i, np.linspace(-5, 5, n_i), x)
    for x in x_linespace_10]) for n_i in targets_n}

plt.figure(dpi=150)
plt.title("(1) Consider  $f(x) = 1 / (1 + x^2)$ ")
plt.legend(handles=plt.plot(x_linespace_10, y1, label=" $f(x) = 1 / (1 + x^2)$ "), loc='best')
plt.figure(dpi=150)

```



3.75 0.4232 -0.0420 -0.1239

4.75 0.2020 0.2785 -6.3726

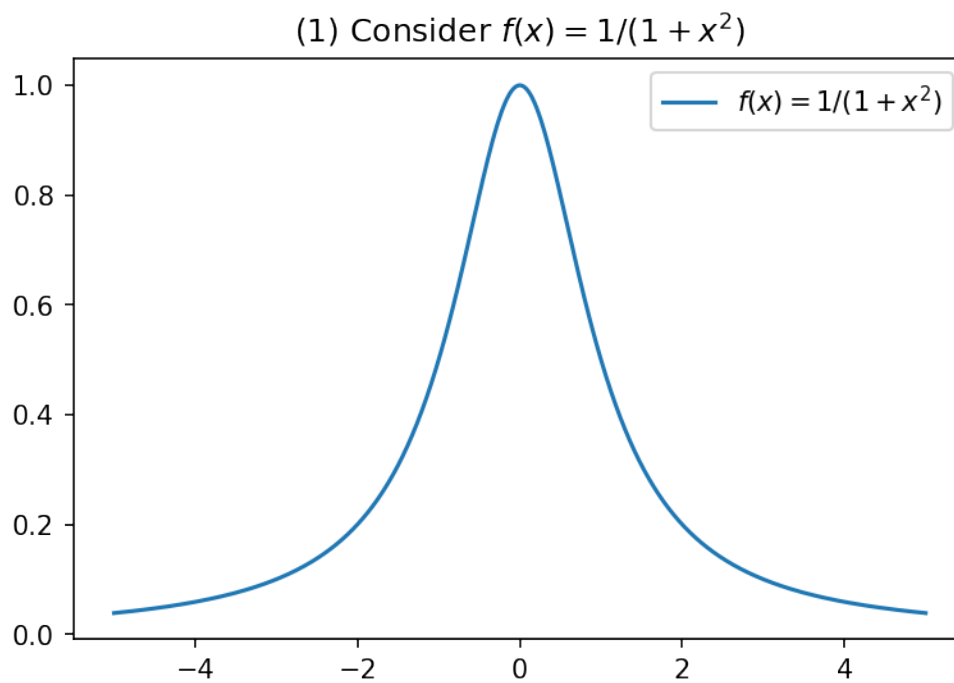
(2) 考虑  $f(x) = e^x$  in  $[-1, 1]$

	5	10	20
-0.95	0.3863	0.3867	0.3867
-0.05	0.9513	0.9512	0.9512
0.05	1.0512	1.0513	1.0513
0.95	2.5863	2.5857	2.5857

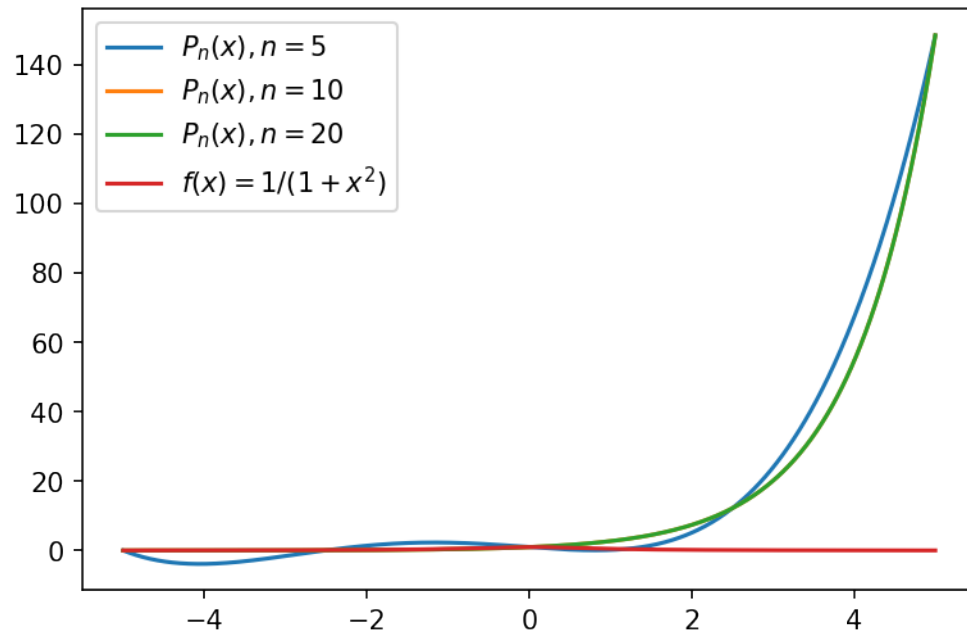
差值:

	5	10	20
-0.95	0.0004	-0.0	0.0
-0.05	-0.0001	-0.0	0.0
0.05	0.0001	-0.0	-0.0
0.95	-0.0006	-0.0	-0.0

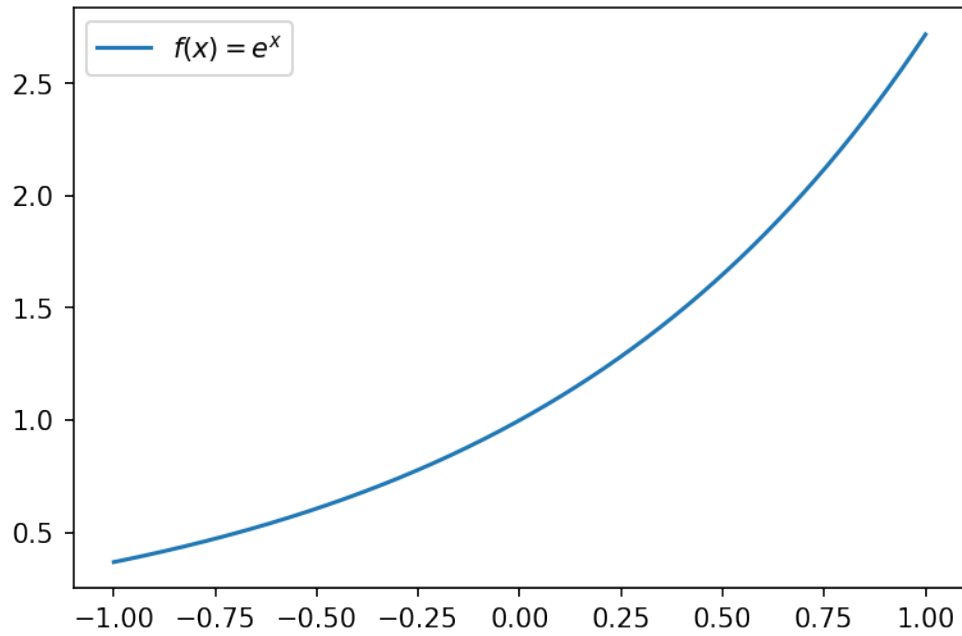
画出两个函数以及其拉格朗日多项式的图像:



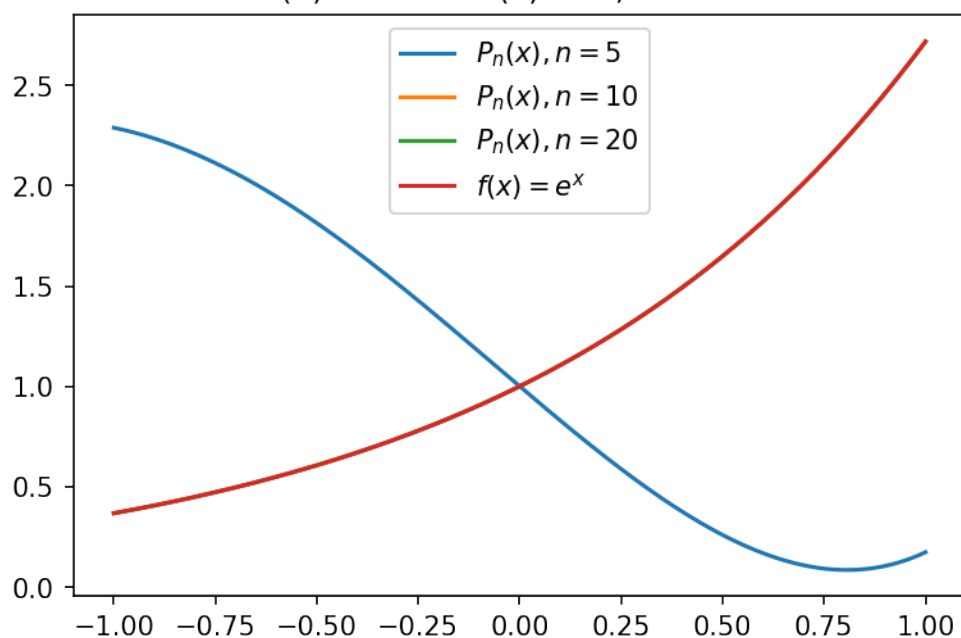
(1) Consider  $f(x) = 1/(1 + x^2)$ , select  $n$



(2) Consider  $f(x) = e^x$



(2) Consider  $f(x) = e^x$ , select n



### 1.3.2 问题二

```
[17]: def problem2():
    print("问题 2: 插值区间越小越好吗? ")

    def sub(targets_x, targets_n, solve):
        return DataFrame({n: {x: solve(n, x) for x in targets_x} for n in
        ↪targets_n}).round(4)

    print("(1) 考虑 f(x) = 1 / (1 + x^2) in [-1, 1]")
    targets_x_1 = [-0.95, -0.05, 0.05, 0.95]
    targets_n = [5, 10, 20]
    print(sub(
        targets_x=targets_x_1,
        targets_n=targets_n,
        solve=lambda n_i, x: P(
            lambda x_i: 1 / (1 + x_i ** 2), np.linspace(-5, 5, n_i), x
        )
    ))
```



```

))
print("差值: ")
print(sub(
    targets_x=targets_x_1,
    targets_n=targets_n,
    solve=lambda n_i, x: ((1 / (1 + x ** 2)) - P(
        lambda x_i: 1 / (1 + x_i ** 2), np.linspace(-5, 5, n_i), x
    ))
))

print("(2) 考虑  $f(x) = e^x$  in  $[-5, 5]$ ")
targets_x_2 = [0.75, 1.75, 2.75, 3.75, 4.75]
print(sub(
    targets_x=targets_x_2,
    targets_n=targets_n,
    solve=lambda n_i, x: P(lambda x_i: np.e ** x_i,
        np.linspace(-1, 1, n_i), x)
))

print("差值: ")
print(sub(
    targets_x=targets_x_2,
    targets_n=targets_n,
    solve=lambda n_i, x: (np.e ** x - P(lambda x_i: np.e ** x_i,
        np.linspace(-1, 1, n_i), x))
))

print("画出两个函数以及其拉格朗日多项式的图像: ")
slice_fluent_size = 1000
x_linespace_2 = np.linspace(-1, 1, slice_fluent_size)
x_linespace_10 = np.linspace(-5, 5, slice_fluent_size)
y1 = 1 / (1 + x_linespace_2**2)
Ls1 = {n_i: np.array([P(lambda x_i: np.e ** x_i, np.linspace(-5, 5, n_i), x)
    for x in x_linespace_2]) for n_i in targets_n}

plt.figure(dpi=150)
plt.title("(1) Consider  $f(x) = 1 / (1 + x^2)$ ")
plt.legend(handles=plt.plot(x_linespace_2, y1, label=" $f(x) = 1 / (1 + x^2)$ "), loc='best')

```

```

plt.figure(dpi=150)
plt.title("(1) Consider  $f(x) = 1 / (1 + x^2)$ , select n")
plt.legend(handles=[*plt.plot(x_linespace_2, Ls1[n_i],
↪label=f"$P_n(x), n={n_i}$") [0] for n_i in Ls1],
            plt.plot(x_linespace_2, y1, label="$f(x) = 1 / (1 + ↪
↪x^2)$") [0]), loc='best')
y2 = np.e**x_linespace_10
Ls2 = {n_i: np.array([P(lambda x_i: np.e ** x_i, np.linspace(-5, 5, n_i), x)
                      for x in x_linespace_10]) for n_i in targets_n}
plt.figure(dpi=150)
plt.title("(2) Consider  $f(x) = e^x$ ")
plt.legend(handles=plt.plot(x_linespace_10, y2, label="$f(x) = e^x$"),
↪loc='best')
plt.figure(dpi=150)
plt.title("(2) Consider  $f(x) = e^x$ , select n")
plt.legend(handles=[*plt.plot(x_linespace_10, Ls2[n_i],
↪label=f"$P_n(x), n={n_i}$") [0] for n_i in Ls2],
            plt.plot(x_linespace_10, y2, label="$f(x) = e^x$") [0]),
↪loc='best')

problem2()

```

问题 2: 插值区间越小越好吗?

(1) 考虑  $f(x) = 1 / (1 + x^2)$  in  $[-1, 1]$

	5	10	20
-0.95	0.8499	0.6013	0.5213
-0.05	0.9996	0.8607	0.9905
0.05	0.9996	0.8607	0.9905
0.95	0.8499	0.6013	0.5213

差值:

	5	10	20
-0.95	-0.3243	-0.0757	0.0044
-0.05	-0.0021	0.1368	0.0070
0.05	-0.0021	0.1368	0.0070

0.95 -0.3243 -0.0757 0.0044

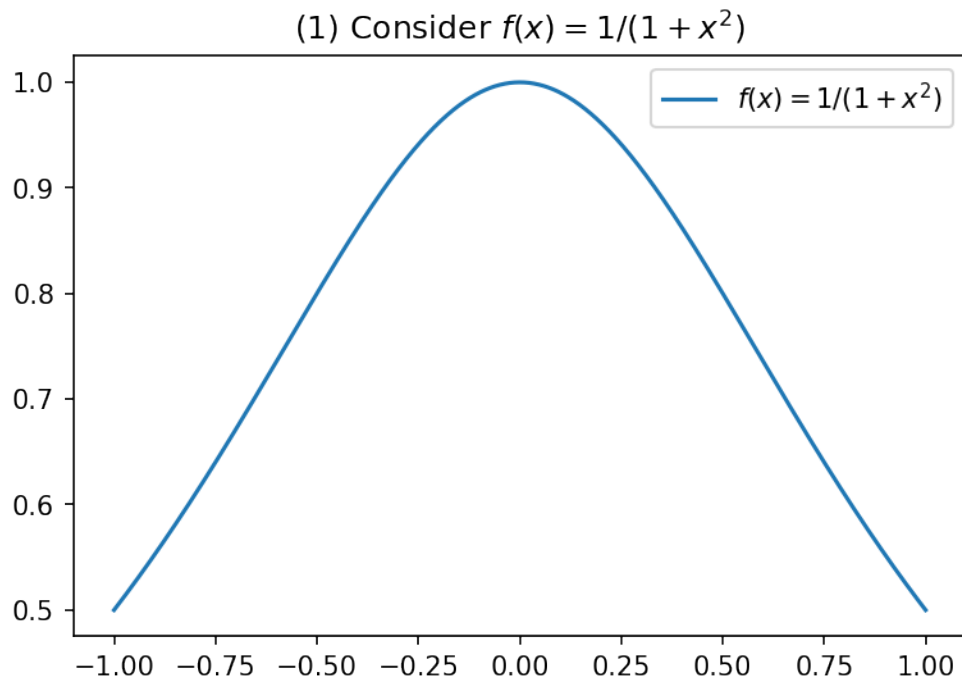
(2) 考虑  $f(x) = e^x$  in  $[-5, 5]$

	5	10	20
0.75	2.1180	2.1170	2.1170
1.75	5.6343	5.7546	5.7546
2.75	13.6951	15.6357	15.2270
3.75	29.7101	42.3238	-237.3955
4.75	58.1314	113.0561	-19138.6250

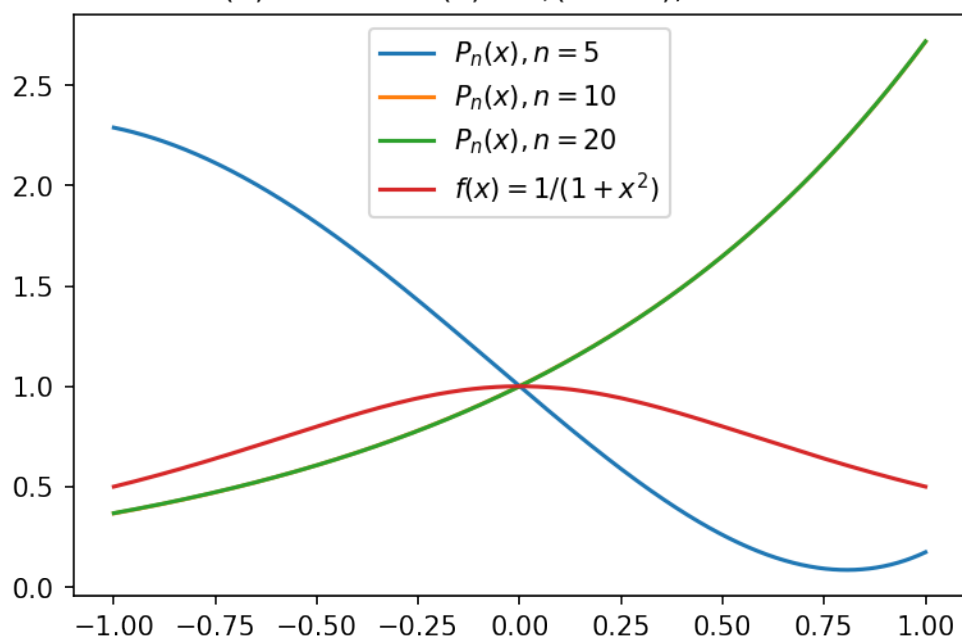
差值:

	5	10	20
0.75	-0.0010	0.0000	-0.0000
1.75	0.1203	0.0000	0.0000
2.75	1.9475	0.0069	0.4156
3.75	12.8110	0.1973	279.9166
4.75	57.4529	2.5282	19254.2093

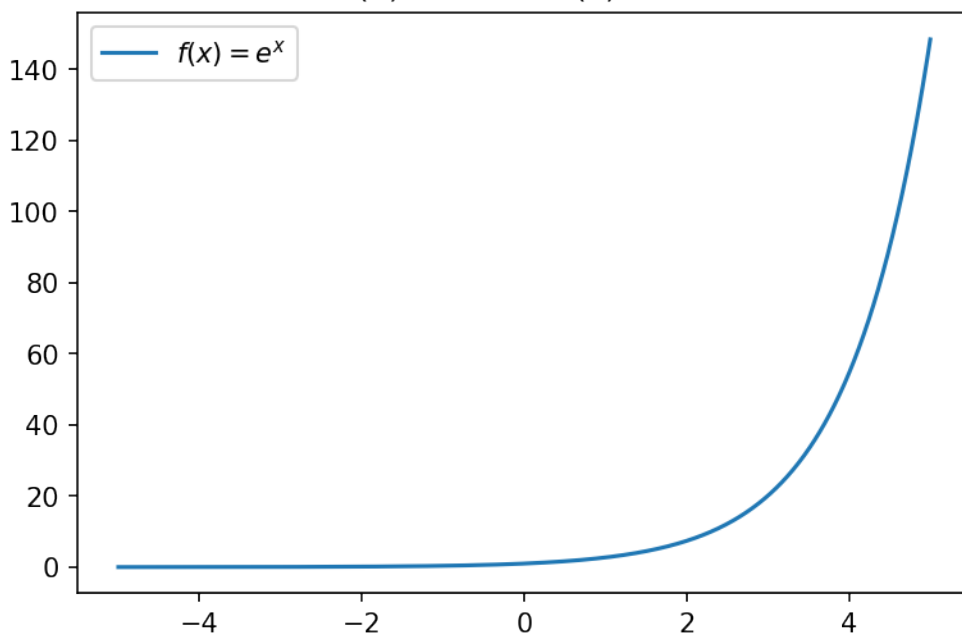
画出两个函数以及其拉格朗日多项式的图像:



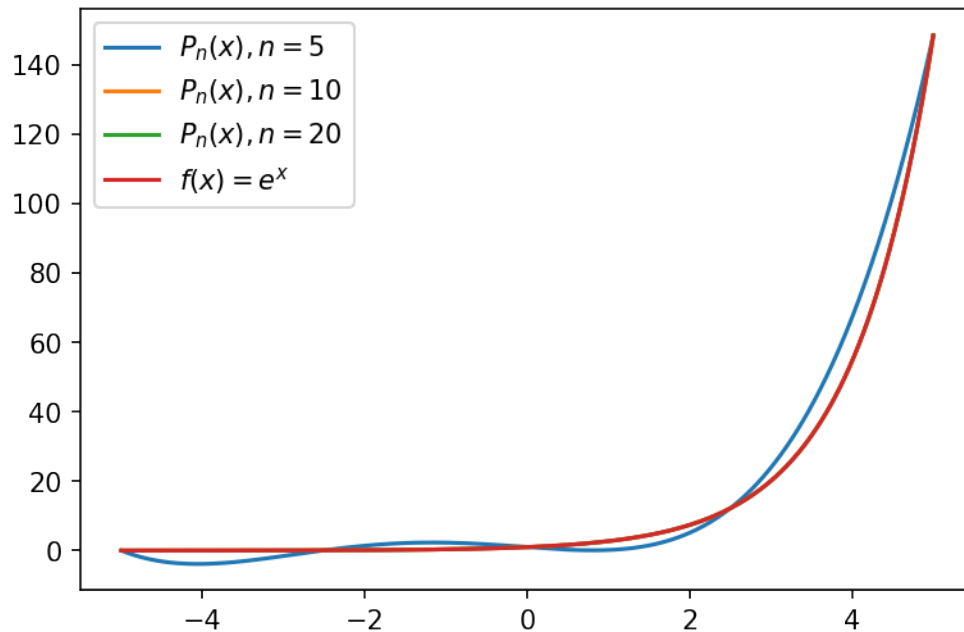
(1) Consider  $f(x) = 1/(1 + x^2)$ , select  $n$



(2) Consider  $f(x) = e^x$



(2) Consider  $f(x) = e^x$ , select  $n$



### 1.3.3 问题四

```
[18]: def problem4():
    print("问题 4: 考虑拉格朗日插值问题, 内插比外推更可靠吗? ")

    def f(x: float):
        return np.sqrt(x)

    target_x = [5, 50, 115, 185]

    print("考虑函数  $f(x) = \sqrt{x}$ ")

    def do_by_node(index: int, nodes: list):
        print(f"({index}) 考虑以", ", ".join(
            [f"x_{i} = {nodes[i]}" for i in range(3)]), "为节点的拉格朗日插值多项式  $P_2(x)$ ")
        print(DataFrame({
            "函数值": {x: P(f, nodes, x) for x in target_x},
```

```

        "f(x)": {x: f(x) for x in target_x},
        "差值": {x: f(x) - P(f, nodes, x) for x in target_x}
    }).round(4))
mean = np.mean([f(x) - P(f, nodes, x) for x in target_x])
print(f"平均差值: {mean:.4f}")
return mean

nodes_data = [
    [(i + 1) ** 2 for i in range(3)],
    [(i + 5) ** 2 for i in range(3)],
    [(i + 9) ** 2 for i in range(3)],
    [(i + 12) ** 2 for i in range(3)]
]
means = [do_by_node(i + 1, nodes=nodes_data[i])
          for i in range(len(nodes_data))]
return DataFrame({
    "x_0,x_1,x_2": [",".join([str(n) for n in nodes_data[i]]) for i in
↪range(len(nodes_data))],
    "平均差值": [means[i] for i in range(len(nodes_data))]
}).round(4)

problem4()

```

问题 4: 考虑拉格朗日插值问题, 内插比外推更可靠吗?

考虑函数  $f(x) = \sqrt{x}$

(1) 考虑以  $x_0 = 1$ ,  $x_1 = 4$ ,  $x_2 = 9$  为节点的拉格朗日插值多项式  $P_2(x)$

	函数值	$f(x)$	差值
5	2.2667	2.2361	-0.0306
50	-20.2333	7.0711	27.3044
115	-171.9000	10.7238	182.6238
185	-492.7333	13.6015	506.3348

平均差值: 179.0581

(2) 考虑以  $x_0 = 25$ ,  $x_1 = 36$ ,  $x_2 = 49$  为节点的拉格朗日插值多项式  $P_2(x)$

	函数值	$f(x)$	差值
5	2.8205	2.2361	-0.5844

50	7.0688	7.0711	0.0023
115	9.0385	10.7238	1.6853
185	5.6527	13.6015	7.9488

平均差值: 2.2630

(3) 考虑以  $x_0 = 81$ ,  $x_1 = 100$ ,  $x_2 = 121$  为节点的拉格朗日插值多项式  $P_2(x)$

	函数值	$f(x)$	差值
5	4.0952	2.2361	-1.8592
50	7.1742	7.0711	-0.1031
115	10.7256	10.7238	-0.0018
185	13.3659	13.6015	0.2356

平均差值: -0.4321

(4) 考虑以  $x_0 = 144$ ,  $x_1 = 169$ ,  $x_2 = 196$  为节点的拉格朗日插值多项式  $P_2(x)$

	函数值	$f(x)$	差值
5	5.1411	2.2361	-2.9050
50	7.6026	7.0711	-0.5316
115	10.7508	10.7238	-0.0270
185	13.6026	13.6015	-0.0012

平均差值: -0.8662

[18]:

	$x_0, x_1, x_2$	平均差值
0	1, 4, 9	179.0581
1	25, 36, 49	2.2630
2	81, 100, 121	-0.4321
3	144, 169, 196	-0.8662

## 1.4 实验结果

准确规范地给出各个实验题目的结果, 并对相应的思考题给出正确合理的回答与说明。

由题目(1)代码、数据和图像可知:

1. 对  $f(x) = 1/(1+x^2)$  函数而言, 在  $[-5, 5]$  范围内, 并不是  $n$  越大越好,  $n$  越大反而误差增大。
2. 对  $f(x) = e^x$  函数而言, 在  $[-1, 1]$  范围内,  $n$  越大拟合效果越好。

所以不是  $n$  越大越好, 需要结合具体函数考虑。

由题目(2)代码、数据和图像, 并且结合题目(1)的数据可知:

1. 对  $f(x) = 1/(1+x^2)$  函数而言,  $[-1, 1]$  差值区间效果要比  $[-5, 5]$  好。
2. 对  $f(x) = e^x$  函数而言,  $[-5, 5]$  差值区间效果要比  $[-1, 1]$  好。

所以不是差值区间越小越好，需要结合具体函数考虑。

由题目（4）代码、数据和图像，对函数  $f(x) = \sqrt{x}$ ，内插确实比外推可靠。

### 思考题

对问题一存在的问题，应该如何解决？

问题一中， $f(x) = \frac{1}{1+x^2}$  在  $[-5, 5]$  的差值区间、 $n \in \{10, 20\}$  的情况下拟合效果并不好， $n = 10, n = 20$  的时候多项式在  $x$  较大的时候明显偏大。

由实验数据可知不应选择过大的插值多项式次数， $n$  应该  $< 10$ 。

对问题二中存在的问题的回答，试加以说明。

插值区间不是越小越好，如这两个函数： $f(x) = \frac{1}{1+x^2}$  和  $f(x) = e^x$ ，前者在  $[-1, 1]$  上插值效果较好而在  $[-5, 5]$  上效果不好；后者在  $[-1, 1]$  上效果不好而在  $[-5, 5]$  上效果较好。

如何理解插值问题中的内插和外推？

内插即只对已知数据集内部范围的点的插值运算，外推即对已知数据集外部范围的点进行插值运算。

内插运算比外推更可靠，偏差更小的原因是内插能够更加有效地利用已知数据集的限制条件，尽量利用已知的信息进行计算推测，故更加可靠。

## 2 实验题目 2：龙贝格积分法

### 2.1 问题分析

准确描述并总结出实验题目（摘要），并准确分析原题的目的和意义。

#### 2.1.1 方法概要

利用复化梯形求积公式、复化辛普生求积公式、复化柯特斯求积公式的误差估计式计算积分  $\int_a^b f(x)dx$ ，记  $h = \frac{b-a}{n}$ ,  $x_k = a + k \times h, k = 0, 1, \dots, n$ ，其计算公式：



$$\begin{aligned}
T_n &= \frac{1}{2}h \sum_{k=1}^n [f(x_{k-1}) + f(x_k)] \\
T_{2n} &= \frac{1}{2}T_n + \frac{1}{2}h \sum_{k=1}^n f(x_k - \frac{1}{2}h) \\
S_n &= \frac{1}{3}(4T_{2n} - T_n) \\
C_n &= \frac{1}{15}(16S_{2n} - S_n) \\
R_n &= \frac{1}{63}(64C_{2n} - C_n)
\end{aligned}$$

或:

$$\begin{aligned}
T_0(h) &= T(h) \\
T_m(h) &= \frac{T_{m-1}(\frac{h}{2}) - (\frac{1}{2})^{2m}T_{m-1}(h)}{1 - (\frac{1}{2})^{2m}} \\
&= \frac{4^m T_{m-1}(\frac{h}{2}) - T_{m-1}(h)}{4^m - 1}
\end{aligned}$$

### 2.1.2 实验目的

用龙贝格积分法求函数  $f(x)$  从  $a$  到  $b$  的积分, 即  $\int_a^b f(x)$ 。

输入:  $a, b, \varepsilon, f$

输出: 龙贝格 T 数表

## 2.2 数学原理

数学原理表达清晰且书写准确。

利用复化梯形求积公式、复化辛普生求积公式、复化柯特斯求积公式的误差估计式可求得龙贝格积分法的公式:

$$\begin{aligned}
T_0(h) &= T(h) \\
T_m(h) &= \frac{T_{m-1}(\frac{h}{2}) - (\frac{1}{2})^{2m}T_{m-1}(h)}{1 - (\frac{1}{2})^{2m}} \\
&= \frac{4^m T_{m-1}(\frac{h}{2}) - T_{m-1}(h)}{4^m - 1}
\end{aligned}$$

## 2.3 程序设计流程

编译通过，根据输入能得到正确输出。

[226]: # 添加需要的库

```
import numpy as np
from pandas import DataFrame
from typing import *
```

```
[227]: def romberg(
    f: Callable[[float], float],
    a: float, b: float, epsilon: float,
    *args,
    get_steps: bool = False, max_len: int = 32, **kwargs):
    max_len: int = 32
    h = b - a
    i = 1
    T = np.array([[0.0 for _ in range(max_len)] for _ in range(max_len)])
    T[0][0] = (f(a) + f(b)) * h / 2
    # print(T[0][0])

    def get_slice():
        return np.array(T[0:(i+1), 0:(i+1)])
    while True:
        ii = 2**(i-1)
        # print(f"i = {i}, ii = {ii}")
        T[0][i] = T[0][i-1] / 2 + h * \
            sum([f(a + (0.0 + k - 1 / 2) * h) for k in range(1, ii + 1)]) / 2
        # print(f"T[0][i] = {T[0][i]}")
        for m in range(1, i + 1):
            k = i - m
            T[m][k] = (4**m * T[m-1][k+1] - T[m-1][k]) / (4**m - 1)
            # print(f"T[i][0] - T[i-1][0] = {T[i][0]} - {T[i-1][0]} = {T[i][0] -
            ↪ T[i-1][0]}")
            # print(f"T[i][0] - T[i-1][0] = {T[i][0] - T[i-1][0]}")
            if abs(T[i][0] - T[i-1][0]) < epsilon:
                if get_steps:
```

```

        return True, i
    else:
        return True, get_slice()
    h = h / 2
    i = i + 1
if get_steps:
    return False, i
else:
    return False, get_slice()

```

[228]: # 使用 Romberg 计算积分

```

global_args = [
    [lambda x: x**2 * np.exp(x), 0, 1, 1e-6],
    [lambda x: np.sin(x) * np.exp(x), 1, 3, 1e-6],
    [lambda x: 4 / (1 + x**2), 0, 1, 1e-6],
    [lambda x: 1 / (1 + x), 0, 1, 1e-6]
]

def run_once(*args, show_result: bool = True, show_T: bool = True, **kwargs):
    res, T = romberg(*args, **kwargs)
    # print(T)
    if res:
        if not isinstance(T, int):
            if show_T:
                print(DataFrame(T))
            if show_result:
                print(f"result = {T[-1][0]}")
            return T[-1][0]
        else:
            return T
    else:
        print("Error")
        return None

```

```
def run(index: int, data_source=global_args, **kwargs):
    return run_once(*data_source[index], **kwargs)
```

[229]: # 第 (1) 问

```
run(0)
```

	0	1	2	3	4
0	1.359141	0.885661	0.760596	0.728890	0.720936
1	0.727834	0.718908	0.718321	0.718284	0.000000
2	0.718313	0.718282	0.718282	0.000000	0.000000
3	0.718282	0.718282	0.000000	0.000000	0.000000
4	0.718282	0.000000	0.000000	0.000000	0.000000

result = 0.7182818284623739

[229]: 0.7182818284623739

[230]: # 第 (2) 问

```
run(1)
```

	0	1	2	3	4	5
0	5.121826	9.279763	10.520554	10.842043	10.923094	10.943398
1	10.665742	10.934151	10.949207	10.950111	10.950167	0.000000
2	10.952045	10.950210	10.950171	10.950170	0.000000	0.000000
3	10.950181	10.950170	10.950170	0.000000	0.000000	0.000000
4	10.950170	10.950170	0.000000	0.000000	0.000000	0.000000
5	10.950170	0.000000	0.000000	0.000000	0.000000	0.000000

result = 10.950170314683838

[230]: 10.950170314683838

[231]: # 第 (3) 问

```
run(2)
```

	0	1	2	3	4	5
0	3.000000	3.100000	3.131176	3.138988	3.140942	3.14143
1	3.133333	3.141569	3.141593	3.141593	3.141593	0.000000
2	3.142118	3.141594	3.141593	3.141593	0.000000	0.000000

```

3  3.141586  3.141593  3.141593  0.000000  0.000000  0.000000
4  3.141593  3.141593  0.000000  0.000000  0.000000  0.000000
5  3.141593  0.000000  0.000000  0.000000  0.000000  0.000000
result = 3.141592653638244

```

[231]: 3.141592653638244

## 2.4 实验结果

准确规范地给出各个实验题目的结果，并对相应的思考题给出正确合理的回答与说明。

```

[232]: DataFrame([run(i, show_T=False, show_result=False)
                  for i in range(3)], ["(1)", "(2)", "(3)"])

```

```

[232]:          0
(1)    0.718282
(2)   10.950170
(3)    3.141593

```

实验题目 1 中各个小问的结果如上表格所示。

思考题：在实验 1 中二分次数和精度的关系如何？

我们使用更高的精度要求进行进一步测试：

```

[233]: def test_epsilon():

        def get_data(e: float): # -> List[List[float]]:
            return [[*item[:-1], e] for item in global_args]

        def get_once(epsilon: float):
            return [run(i, data_source=get_data(epsilon), get_steps=True) for i in
                    range(3)]

        epsilon_list = [1e-5, 1e-6, 1e-9, 1e-12, 1e-14, 1e-16]
        print(DataFrame([get_once(e) for e in epsilon_list], epsilon_list))

test_epsilon()

```

	0	1	2
1.000000e-05	4	5	4
1.000000e-06	4	5	5
1.000000e-09	5	6	6
1.000000e-12	6	7	7
1.000000e-14	6	7	8
1.000000e-16	6	11	13

由数据可知，随着要求精度的提高，二分次数也在随之升高。

### 3 实验题目 3：四阶龙格——库塔方法

#### 3.1 问题分析

准确描述并总结出实验题目（摘要），并准确分析原题的目的和意义。

给定常微分方程初值问题：

$$\begin{cases} \frac{dy}{dx} = f(x, y), & a \leq x \leq b \\ y(a) = \alpha, & h = \frac{b-a}{N} \end{cases}$$

求其数值解  $y_n, n = 1, 2, \dots, N$ 。

##### 3.1.1 实验目的

输入：  $a, b, \alpha, N$

输出：初值问题的数值解  $x_n, y_n, n = 0, 1, 2, \dots, N$

#### 3.2 数学原理

数学原理表达清晰且书写准确。

记  $x_n = a + n \times h, n = 0, 1, \dots, N$ ，利用四阶龙格——库塔方法：

$$\begin{aligned}
K_1 &= hf(x_n, y_n) \\
K_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{K_1}{2}) \\
K_3 &= hf(x_n + \frac{h}{2}, y_n + \frac{K_2}{2}) \\
K_4 &= hf(x_n + h, y_n + K_3) \\
y_{n+1} &= y_n + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4) \\
n &= 0, 1, \dots, N-1
\end{aligned}$$

即可逐次求出微分方程初值问题的数值解  $x_n, y_n, n = 0, 1, 2, \dots, N$ 。

### 3.3 程序设计流程

编译通过，根据输入能得到正确输出。

[126]: # 引入需要的包

```
from typing import *
import numpy as np
from pandas import DataFrame
```

[127]: # 四阶龙格——库塔方法

```
def runge_kutta(
    f: Callable[[float, float], float],
    a: float, b: float, alpha: float, N: int):
    x_list, y_list = [], []
    h = (b - a) / N
    x, y = a, alpha
    x_list.append(x)
    y_list.append(y)
    for _ in range(N):
        k_1 = h*f(x, y)
        k_2 = h*f(x+h/2, y+k_1/2)
        k_3 = h*f(x+h/2, y+k_2/2)
        k_4 = h*f(x+h, y+k_3)
        x = x + h
        y = y + (k_1 + 2*k_2 + 2*k_3 + k_4) / 6
```

```

        x_list.append(x)
        y_list.append(y)
    return x_list, y_list

```

[128]: # 运行测试参数

```

global_args = [
    [lambda x, y: x + y, 0, 1, -1, [5, 10, 20], lambda x: -x-1, "问题 1 (1)"],
    [lambda x, y: -y**2, 0, 1, 1, [5, 10, 20],
     lambda x: 1 / (x + 1), "问题 1 (2)"],
    [lambda x, y: 2 * y / x + x**2 +
     np.exp(x), 1, 3, 0, [5, 10, 20], lambda x: x**2 * (np.exp(x) - np.e),
     ↪ "问题 2 (1)"],
    [lambda x, y: (y + y**2) / x, 1, 3, -2, [5, 10, 20],
     lambda x: 2 * x / (1 - 2 * x), "问题 2 (2)"],
    [lambda x, y: -20 * (y-x**2) + 2 * x, 0, 1, 1.0 / 3,
     [5, 10, 20], lambda x: x**2 + np.exp(-20*x)/3, "问题 3 (1)"],
    [lambda x, y: -20 * y + 20 *
     np.sin(x) + np.cos(x), 0, 1, -1, [5, 10, 20], lambda x: np.exp(-20*x) +
     ↪ np.sin(x), "问题 3 (2)"],
    [lambda x, y: -20*(y-np.exp(x)*np.sin(x)) + np.exp(x)*(np.sin(x) + np.
     ↪ cos(x)),
     0, 1, 0, [5, 10, 20], lambda x: np.exp(x)*np.sin(x), "问题 3 (3)"]
]

```

[129]: # 求数据的均方误差

```

def get_error(f: Callable[[float], float], data):
    x, y = data
    standard = np.array([f(x_i) for x_i in x])
    return sum((y - standard) ** 2) / len(x)

```

[130]: # 运行一次

```

def run(index: int):
    res = []
    for n in global_args[index][-3]:
        data = runge_kutta(*[
            *global_args[index][: -3], n

```



```

    ])
    error = get_error(global_args[index][-2], data)
    res.append({
        "N": n,
        "标号": global_args[index][-1],
        "均方误差": error,
        "x": data[0],
        "y": data[1],
    })
    return res

```

[131]: # 运行所有并且返回结果表格

```

def run_all():
    all_data = [run(i) for i in range(len(global_args))]
    all = []
    for d in all_data:
        all.extend(d)
    return DataFrame(all)

run_all()

```

```

[131]:      N  ...                                     y
0      5  ...  [-1, -1.2, -1.4, -1.5999999999999999, -1.79999...
1     10  ...  [-1, -1.1, -1.2000000000000002, -1.300000000000...
2     20  ...  [-1, -1.05, -1.1, -1.1500000000000001, -1.2000...
3      5  ...  [1, 0.8333390356230387, 0.7142921304635431, 0...
4     10  ...  [1, 0.9090911863322196, 0.8333337288430721, 0...
5     20  ...  [1, 0.9523809630269818, 0.9090909268125394, 0...
6      5  ...  [0, 2.6076891538492872, 8.124196625549118, 17...
7     10  ...  [0, 1.0055508321940254, 2.613659182614748, 4.9...
8     20  ...  [0, 0.434963271851454, 1.0058077554774965, 1.7...
9      5  ...  [-2, -1.5539889980952382, -1.3836172899114931,...
10    10  ...  [-2, -1.7142451804511538, -1.5555228848496192,...
11    20  ...  [-2, -1.8333328294259301, -1.7142851698413297,...
12     5  ...  [0.3333333333333333, 2.506666666666667, 11.69...
13    10  ...  [0.3333333333333333, 0.2511111111111111, 0.363...

```

```

14 20 ... [0.3333333333333333, 0.16437499999999997, 0.16...
15 5 ... [-1, -4.802661893779973, -24.623829295619263, ...
16 10 ... [-1, -0.2335276701694724, 0.0874382457825747, ...
17 20 ... [-1, -0.32502148139805487, -0.0407937778694009...
18 5 ... [0, 0.2986462127501341, 0.927219870027348, 2.8...
19 10 ... [0, 0.11205510913037421, 0.2451165144244346, 0...
20 20 ... [0, 0.05259503995574239, 0.11040898628183947, ...

```

[21 rows x 5 columns]

为防止输出 PDF 时表格格式被破坏，在此放入上方表格的图片。

	N	标号	均方误差	x	y
0	5	问题 1 (1)	2.465190e-32	[0, 0.2, 0.4, 0.6000000000000001, 0.8, 1.0]	[-1, -1.2, -1.4, -1.5999999999999999, -1.79999...
1	10	问题 1 (1)	3.182337e-31	[0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0...	[-1, -1.1, -1.2000000000000002, -1.30000000000...
2	20	问题 1 (1)	2.206932e-31	[0, 0.05, 0.1, 0.15000000000000002, 0.2, 0.25,...	[-1, -1.05, -1.1, -1.1500000000000001, -1.2000...
3	5	问题 1 (2)	2.569560e-11	[0, 0.2, 0.4, 0.6000000000000001, 0.8, 1.0]	[1, 0.8333390356230387, 0.7142921304635431, 0....
4	10	问题 1 (2)	1.282857e-13	[0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0...	[1, 0.9090911863322196, 0.8333337288430721, 0....
5	20	问题 1 (2)	5.366889e-16	[0, 0.05, 0.1, 0.15000000000000002, 0.2, 0.25,...	[1, 0.9523809630269818, 0.9090909268125394, 0....
6	5	问题 2 (1)	2.023870e+03	[1, 1.4, 1.7999999999999998, 2.199999999999999...	[0, 2.6076891538492872, 8.124196625549118, 17....
7	10	问题 2 (1)	1.541237e+03	[1, 1.2, 1.4, 1.5999999999999999, 1.7999999999...	[0, 1.0055508321940254, 2.613659182614748, 4.9...
8	20	问题 2 (1)	1.315425e+03	[1, 1.1, 1.2000000000000002, 1.300000000000000...	[0, 0.434963271851454, 1.0058077554774965, 1.7...
9	5	问题 2 (2)	7.459298e-07	[1, 1.4, 1.7999999999999998, 2.199999999999999...	[-2, -1.5539889980952382, -1.3836172899114931,...
10	10	问题 2 (2)	4.401675e-10	[1, 1.2, 1.4, 1.5999999999999999, 1.7999999999...	[-2, -1.7142451804511538, -1.5555228848496192,...
11	20	问题 2 (2)	8.946298e-14	[1, 1.1, 1.2000000000000002, 1.300000000000000...	[-2, -1.8333328294259301, -1.7142851698413297,...
12	5	问题 3 (1)	3.263086e+05	[0, 0.2, 0.4, 0.6000000000000001, 0.8, 1.0]	[0.3333333333333333, 2.5066666666666677, 11.69...
13	10	问题 3 (1)	4.889478e-01	[0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0...	[0.3333333333333333, 0.2511111111111111, 0.363...
14	20	问题 3 (1)	4.815643e-01	[0, 0.05, 0.1, 0.15000000000000002, 0.2, 0.25,...	[0.3333333333333333, 0.16437499999999997, 0.16...
15	5	问题 3 (2)	1.697642e+06	[0, 0.2, 0.4, 0.6000000000000001, 0.8, 1.0]	[-1, -4.802661893779973, -24.623829295619263, ...
16	10	问题 3 (2)	3.852943e-01	[0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0...	[-1, -0.2335276701694724, 0.0874382457825747, ...
17	20	问题 3 (2)	2.209629e-01	[0, 0.05, 0.1, 0.15000000000000002, 0.2, 0.25,...	[-1, -0.32502148139805487, -0.0407937778694009...
18	5	问题 3 (3)	3.617925e+02	[0, 0.2, 0.4, 0.6000000000000001, 0.8, 1.0]	[0, 0.2986462127501341, 0.927219870027348, 2.8...
19	10	问题 3 (3)	9.913683e-06	[0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0...	[0, 0.11205510913037421, 0.2451165144244346, 0...
20	20	问题 3 (3)	1.212561e-08	[0, 0.05, 0.1, 0.15000000000000002, 0.2, 0.25,...	[0, 0.05259503995574239, 0.11040898628183947, ...

### 3.4 实验结果

准确规范地给出各个实验题目的结果，并对相应的思考题给出正确合理的回答与说明。

实验数据结果如上表所示。

思考题：

1. 对实验 1，数值解和解析解相同吗？为什么？试加以说明。

在误差范围内基本可以认为相同。由上表可知，对问题 1，当  $N = 20$  时，其结果和标准值的

均方误差均小于  $10^{-15}$ ，都是非常小的，所以在误差范围内可以认为数值解和解析解相同。

2. 对实验 2， $N$  越大越精确吗？试加以说明。

在实验 2 的数据中，随着  $N$  的增大，其均方误差越来越小，所以对实验二， $N$  越大越精确。

3. 对实验 3， $N$  较小会出现什么现象？试加以说明。

在实验 3 的数据中，当  $N$  较小时，其均方误差非常大，达到  $10^2$  甚至  $10^6$ 。

## 4 实验题目 4：牛顿迭代法

### 4.1 问题分析

准确描述并总结出实验题目（摘要），并准确分析原题的目的和意义。

#### 4.1.1 方法概要

求非线性方程  $f(x) = 0$  的根  $x^*$ ，牛顿分析法计算公式：

$$x_0 = \alpha, x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, n = 0, 1, \dots$$

一般地，牛顿迭代法具有局部收敛性，为了保证迭代收敛，要求，对充分小的  $\delta, \alpha \in O(x^*, \delta)$ 。如果  $f(x) \in C^2[a, b], f(x^*) = 0, f'(x^*) \neq 0$ ，那么，对充分小的  $\delta > 0$ ，当  $\alpha \in O(x^*, \delta)$  时，由牛顿迭代法计算出的  $\{x_n\}$  收敛于  $x^*$ ，且收敛速度是 2 阶的；如果  $f(x) \in C^m[a, b], f(x^*) = f'(x^*) = \dots = f^{(m-1)}(x^*) = 0, f^{(m)}(x^*) \neq 0 (m > 1)$ ，那么，对充分小的  $\delta > 0$ ，当  $\alpha \in O(x^*, \delta)$  时，由牛顿迭代法计算出的  $\{x_n\}$  收敛于  $x^*$ ，且收敛速度是 1 阶的。

#### 4.1.2 实验目的

利用牛顿迭代法求  $f(x) = 0$  的根。

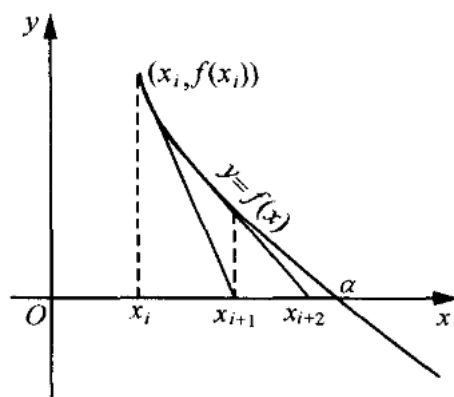
输入：初值  $\alpha$ ，精度  $\varepsilon_1, \varepsilon_2$ ，最大迭代次数  $N$

输出：方程  $f(x) = 0$  根  $x^*$  的近似值或计算失败标志

### 4.2 数学原理

数学原理表达清晰且书写准确。

### 4.2.1 牛顿迭代法的几何意义



由上图所示，方程  $f(x) = 0$  的根  $\alpha$  是曲线  $y = f(x)$  与直线  $y = 0$  的交点的横坐标。牛顿迭代法是取过  $(x_i, f(x_i))$  点的切线方程

$$y = f(x_i) + f'(x_i)(x - x_i)$$

与  $y = 0$  的交点的横坐标

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

作为根的新的近似值。由此往复，只要初值取得接近根  $\alpha$ ， $\{x_0, x_1, \dots, x_n\}$  会很快收敛于  $\alpha$ 。

## 4.3 程序设计流程

编译通过，根据输入能得到正确输出。

```
[52]: # 引入需要的包
import numpy as np
from pandas import DataFrame
from matplotlib import pyplot as plt
from typing import *
```

```
[53]: def newton(
    f: Callable[[float], float],
    f_: Callable[[float], float],
    alpha: float,
```

```

    N: int,
    epsilon_1: float,
    epsilon_2: float,
    *args,
    **kwargs):# -> Optional[float]:
history: List[float] = []
n = 1
x = alpha
while n <= N:
    history.append(x)
    v, v_ = f(x), f_(x)
    if abs(v) < epsilon_1:
        return x, history
    if abs(v_) < epsilon_2:
        return None, history
    x_ = x - v / v_
    if abs(x_ - x) < epsilon_1:
        history.append(x_)
        return x_, history
    n = n + 1
    x = x_
return None, history

```

```

[54]: def show_history(title: str, history: List[float]):
    plt.figure(dpi=150)
    plt.title(title)
    plt.plot(range(len(history)), history)

```

```

[55]: def run_question(*args, **kwargs):
    res, history = newton(*args, **kwargs)
    if res is None:
        print("拟合失败!")
    else:
        print(f"x^* = {res:.4f}")
    show_history(kwargs.get('title', ''), history)

```

```
[56]: # 问题一
def question_1():
    print("问题 1 (1)")
    run_question(
        f=lambda x: np.cos(x) - x,
        f_=lambda x: -np.sin(x) - 1,
        alpha=np.pi / 4,
        N=10,
        epsilon_1=1e-6,
        epsilon_2=1e-4,
        title="$Q_1 (1): \cos{x} - x = 0, \alpha = " + f"{np.pi / 4:.9f}" +
        "\$"
    )
    print("问题 1 (2)")
    run_question(
        f=lambda x: np.exp(-x) - np.sin(x),
        f_=lambda x: -np.exp(x) - np.cos(x),
        alpha=0.6,
        N=10,
        epsilon_1=1e-6,
        epsilon_2=1e-4,
        title="$Q_1 (2): e^{-1}-\sin{x} = 0, \alpha = 0.6$"
    )

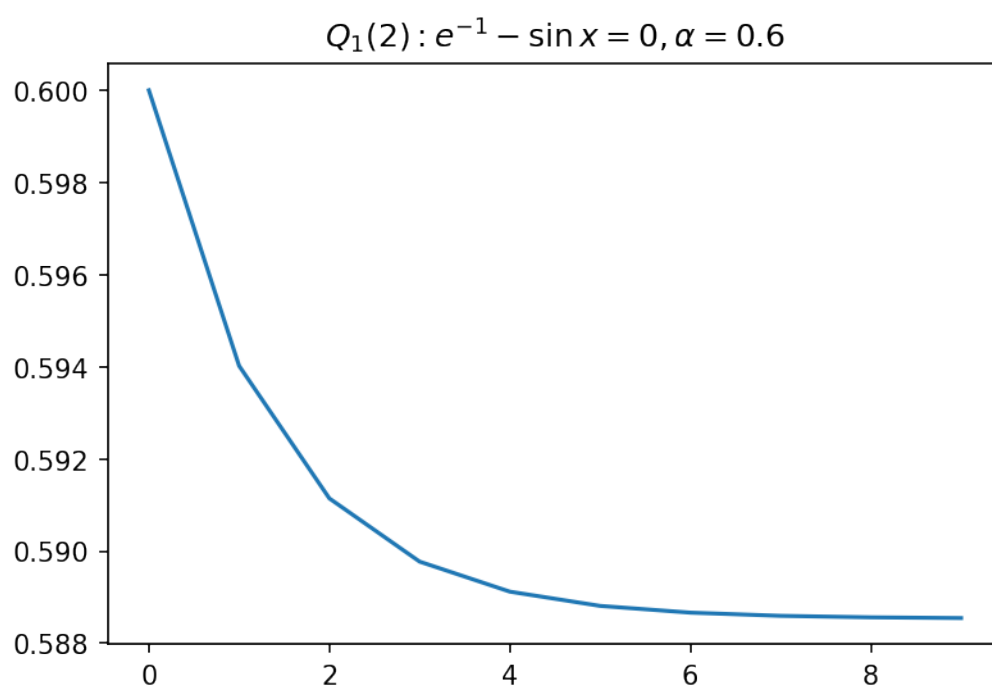
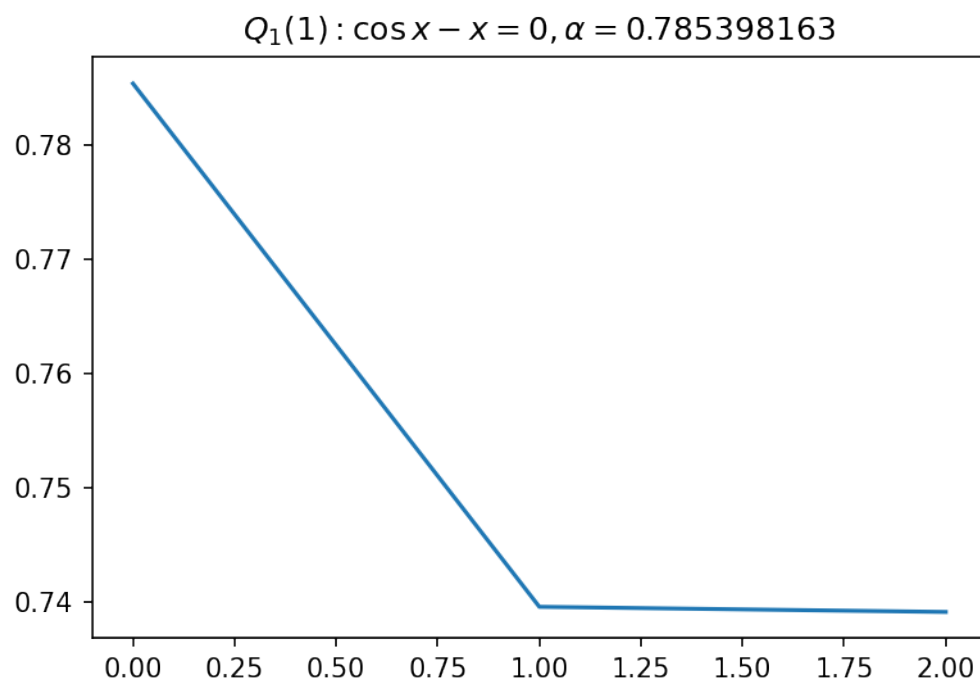
question_1()
```

问题 1 (1)

$x^* = 0.7391$

问题 1 (2)

拟合失败!



```
[57]: # 问题二
def question_2():
    print("问题 2 (1)")
    run_question(
        f=lambda x: x - np.exp(-x),
        f_=lambda x: 1 + np.exp(-x),
        alpha=0.5,
        N=10,
        epsilon_1=1e-6,
        epsilon_2=1e-4,
        title="$Q_2 (1): x-e^{-x}=0, \\alpha = 0.5$")
    print("问题 2 (2)")
    run_question(
        f=lambda x: x**2 - 2 * x * np.exp(-x) + np.exp(-2 * x),
        f_=lambda x: -2*np.exp(-2*x) - 2*np.exp(-x) + 2*x + 2*np.exp(-x)*x,
        alpha=0.5,
        N=10,
        epsilon_1=1e-6,
        epsilon_2=1e-4,
        title="$Q_2 (2): x^2-2xe^{-x}+e^{-2x} = 0, \\alpha = 0.5$")

question_2()
```

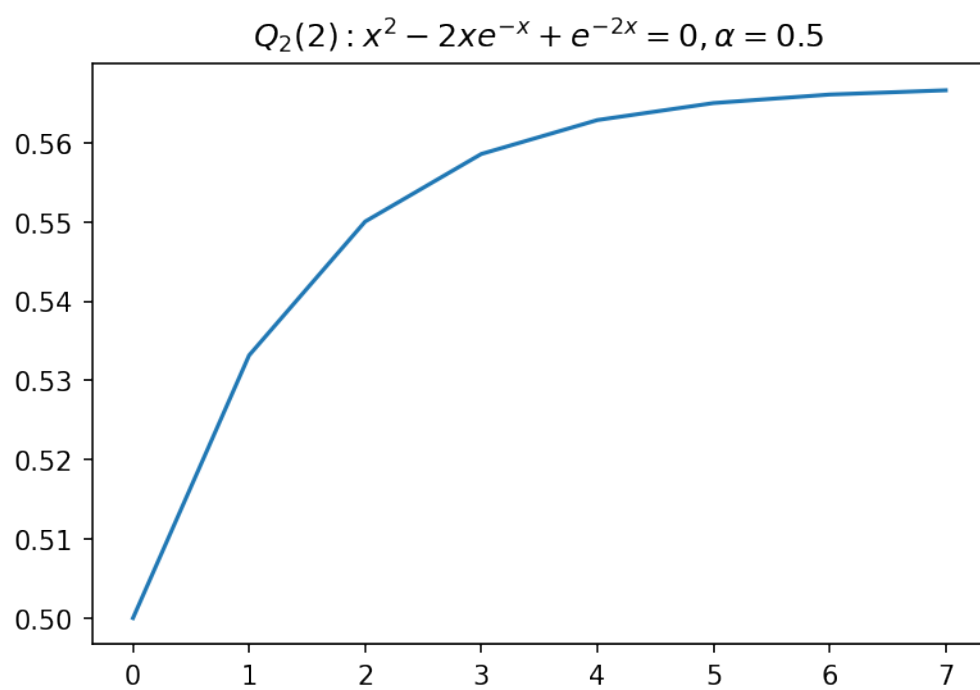
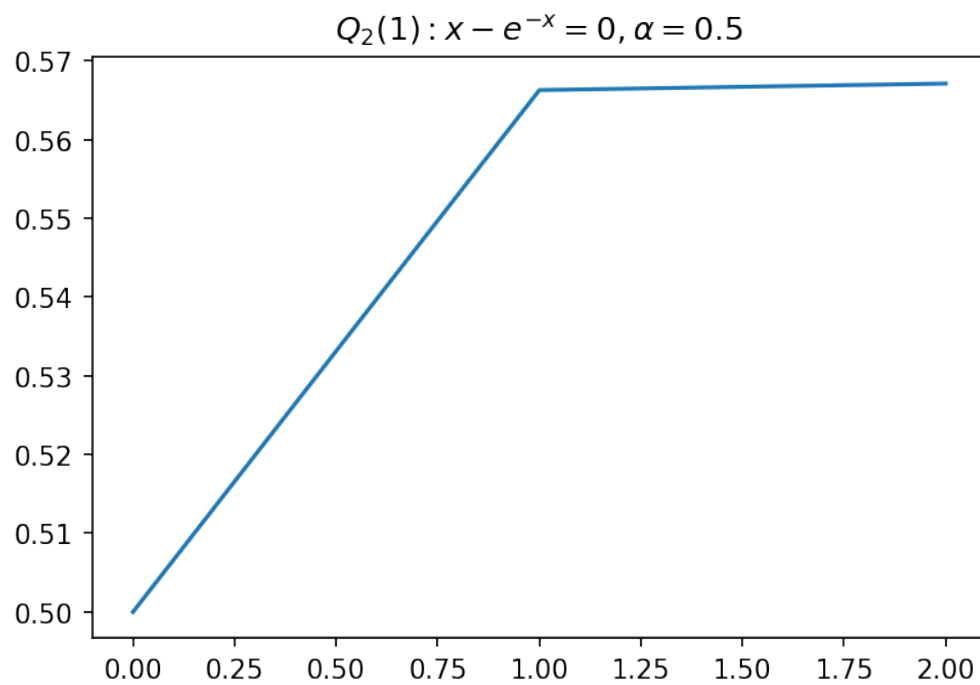
问题 2 (1)

$x^* = 0.5671$

问题 2 (2)

$x^* = 0.5666$





## 4.4 实验结果

准确规范地给出各个实验题目的结果，并对相应的思考题给出正确合理的回答与说明。

由问题 1 输出、图像可知：

1. 第一问在第二次迭代即收敛到目标精度，得结果  $x^* = 0.7391$ （保留四位小数）
2. 第二问在  $N$  次数内收敛失败

由问题 2 输出、图像可知：

1. 第一问在第二次迭代即收敛到目标精度，得结果  $x^* = 0.5671$ （保留四位小数）
2. 第一问在第七次迭代才收敛到目标精度，得结果  $x^* = 0.5666$ （保留四位小数）

思考题：

1. 对实验 1，确定初值的原则是什么？实际计算中应如何解决？初值如果选择得偏离根太远，很可能会出现迭代次数过多或者发散的情况。因此，初值最好选择在靠近根的位置。在实际计算中，如果仅仅使用牛顿迭代法收敛定理来选择初始值，往往比较复杂，一般使用简化方法：

对方程  $f(x) = 0$ ，如果

$$f''(x_0) \neq 0, |f'(x_0)|^2 > \left| \frac{f(x_0)f''(x_0)}{2} \right|$$

则可以保证大多数情况下的牛顿迭代法的收敛性。

2. 对实验 2，如何解释在计算中出现的现象？试加以说明由于牛顿迭代法的收敛阶都是 2，而第二问所求函数是第一问的平方，即  $f_2(x) = f_1^2(x)$ ，平方后的函数的斜率相对原来小许多，所以第二问中收敛就比第一问慢。