计算物理第三次上机作业

白博臣 2022141220036

Problem 1

(a)

在单精度（float）和双精度（double）下，可以使用 Python 中的 NumPy 库来进行计算。使用四倍精度（quadruple precision）进行计算可能需要使用第三方库，因为 Python 的内置浮点数类型（float、double）不支持四倍精度。一个常用的第三方库是 mpmath，它提供了多精度计算功能。

具体代码如下：

```python
import numpy as np
from mpmath import mp


def taylor_exp_single(x, n_terms=10):
    result_single = np.float32(1.0)
    term_single = np.float32(1.0)
    for i in range(1, n_terms):
        term_single = term_single * x / i
        result_single = np.float32(result_single + term_single)
    return result_single


def taylor_exp_double(x, n_terms=10):
    result_double = np.float64(1.0)
    term_double = np.float64(1.0)
    for i in range(1, n_terms):
        term_double = term_double * x / i
        result_double = np.float64(result_double + term_double)
    return result_double


def taylor_exp_quad(x, n_terms=10):
    mp.dps = 36
    result_quad = mp.mpf('1.0')
    term_quad = mp.mpf('1.0')
    for i in range(1, n_terms):
        term_quad = term_quad * x / i
```

```
        result_quad = result_quad + term_quad
    return result_quad


x = 2.0
n_terms = 20

result_single = taylor_exp_single(x, n_terms)
result_double = taylor_exp_double(x, n_terms)
result_quad = taylor_exp_quad(mp.mpf(str(x)), n_terms)

print(f"x=2,Single precision result (float): {result_single}")
print(f"x=2,Double precision result (double): {result_double}")
print(f"x=2,Quadruple precision result: {result_quad}")

x = 10.0
n_terms = 20

result_single = taylor_exp_single(x, n_terms)
result_double = taylor_exp_double(x, n_terms)
result_quad = taylor_exp_quad(mp.mpf(str(x)), n_terms)

print(f"x=10,Single precision result (float): {result_single}")
print(f"x=10,Double precision result (double): {result_double}")
print(f"x=10,Quadruple precision result: {result_quad}")
x = -2.0
n_terms = 20

result_single = taylor_exp_single(x, n_terms)
result_double = taylor_exp_double(x, n_terms)
result_quad = taylor_exp_quad(mp.mpf(str(x)), n_terms)
print(f"x=-2,Single precision result (float): {result_single}")
print(f"x=-2,Double precision result (double): {result_double}")
print(f"x=-2,Quadruple precision result: {result_quad}")

x = -10.0
n_terms = 20

result_single = taylor_exp_single(x, n_terms)
result_double = taylor_exp_double(x, n_terms)
result_quad = taylor_exp_quad(mp.mpf(str(x)), n_terms)
print(f"x=-10,Single precision result (float): {result_single}")
print(f"x=-10,Double precision result (double): {result_double}")
print(f"x=-10,Quadruple precision result: {result_quad}")
```

运算结果如下：

```
x=2,Single precision result (float): 7.38905668258667
x=2,Double precision result (double): 7.3890560989301735
x=2,Quadruple precision result: 7.38905609893017409630292600244713644
x=10,Single precision result (float): 21950.37890625
x=10,Double precision result (double): 21950.37884943194
x=10,Quadruple precision result: 21950.378849431941493977484462047358
x=-2,Single precision result (float): 0.1353352665901184
x=-2,Double precision result (double): 0.1353352832362194
x=-2,Quadruple precision result: 0.135335283236219309154006099710957353
x=-10,Single precision result (float): -27.706260681152344
x=-10,Double precision result (double): -27.706310237425754
x=-10,Quadruple precision result: -27.7063102374256075502656006202454018
```

图表 1 Problem1 的运算结果

当$x < 0$时，由于级数项数太少（$N = 20$），造成了较大的误差，我们可以适当提高N的取值，我们发现得到的结果与理论值更加接近。

```
x=-10,Single precision result (float): 9.518076694803312e-05
x=-10,Double precision result (double): 4.539992962303128e-05
x=-10,Quadruple precision result: 0.0000453999297624848515355915155608305539
4.5399929762484854e-05
```

图表 2 增加N的取值后的结果

（b）

我们使用 if 条件语句进行条件判断，针对不同的正负数应用不同的计算方法，代码如下：

```python
import numpy as np
from mpmath import mp


def taylor_exp_single(x, n_terms=10):
    if x >= 0:
        result = np.float32(1.0)
        term = np.float32(1.0)
        for i in range(1, n_terms):
            term = term * x / i
            result = np.float32(result + term)
    else:
        result = np.float32(1.0) / taylor_exp_single(-x, n_terms)
    return result


def taylor_exp_double(x, n_terms=10):
```

```python
    if x >= 0:
        result = np.float64(1.0)
        term = np.float64(1.0)
        for i in range(1, n_terms):
            term = term * x / i
            result = np.float64(result + term)
    else:
        result = np.float64(1.0) / taylor_exp_double(-x, n_terms)
    return result


def taylor_exp_quad(x, n_terms=10):
    if x >= 0:
        mp.dps = 30
        result = mp.mpf('1.0')
        term = mp.mpf('1.0')
        for i in range(1, n_terms):
            term = term * x / i
            result = result + term
    else:
        result = 1 / taylor_exp_quad(mp.mpf(str(-x)), n_terms)
    return result


test_values = [10, 2, -2, -10]
n_terms = 20

for x in test_values:
    result_single = taylor_exp_single(x, n_terms)
    result_double = taylor_exp_double(x, n_terms)
    result_quad = taylor_exp_quad(mp.mpf(str(x)), n_terms)

    print(f"x = {x}: ")
    print(f"单精度结果 (float): {result_single}")
    print(f"双精度结果 (double): {result_double}")
    print(f"四倍精度结果: {result_quad}")
    print("-----------------------------")
```

得到如下的运算结果：

```
x = 10:
单精度结果 (float): 21950.37890625
双精度结果 (double): 21950.37884943194
四倍精度结果: 21950.378849431941493977484462
-----------------------------
x = 2:
单精度结果 (float): 7.38905668258667
双精度结果 (double): 7.3890560989301735
四倍精度结果: 7.38905609893017409630292600245
-----------------------------
x = -2:
单精度结果 (float): 0.1353352665901184
双精度结果 (double): 0.13533528323662142
四倍精度结果: 0.135335283236621412536131370316
-----------------------------
x = -10:
单精度结果 (float): 4.555729901767336e-05
双精度结果 (double): 4.555730025706956e-05
四倍精度结果: 0.0000455573002570695590487473566669
-----------------------------


进程已结束，退出代码为 0
```

图表 3 改进后的输出结果

针对单精度、双精度、四倍精度的运算，我们发现越高精度的计算最后计算出的结果越接近理论值，原因是在进行级数累加的时候，越高精度意味这更多的正确位数，也就意味着更精确的数值。

Problem 2

首先使用 python 编写这个循环语句：

```python
import numpy as np


def cal(n):
    a_0 = 1 / np.exp(1) * (np.exp(1) - 1)
    for i in range(1, n+1):
        a_n = 1 - (i) * a_0
        a_0 = a_n
    return a_n


# n = eval(input('n='))
# value = cal(n)
# print(value)
for i in range(2, 30):
    print('n=', i, cal(i))
```

```
n= 14 0.06273108042387321
n= 15 0.059033793641901866
n= 16 0.05545930172957014
n= 17 0.05719187059730757
n= 18 -0.02945367075153626
n= 19 1.559619744279189
n= 20 -30.19239488558378
n= 21 635.0402925972594
n= 22 -13969.886437139707
n= 23 321308.38805421325
n= 24 -7711400.313301118
n= 25 192785008.83252797
n= 26 -5012410228.645727
n= 27 135335076174.43463
n= 28 -3789382132883.17
n= 29 109892081853612.92
```

图表 4 部分输出结果

我们发现在$n \geq 17$的时候，其算数误差越来越大，甚至当$n \geq 20$时，其结果逐渐发散，而不是趋近于 0.

我们利用符号计算库 sympy 进行计算：

```
n= 14 -55107190151 + 87178291200*(-1 + E)*exp(-1)
n= 15 -1307674368000*(-1 + E)*exp(-1) + 826607852266
n= 16 -13225725636255 + 20922789888000*(-1 + E)*exp(-1)
n= 17 -355687428096000*(-1 + E)*exp(-1) + 224837335816336
n= 18 -4047072044694047 + 6402373705728000*(-1 + E)*exp(-1)
n= 19 -121645100408832000*(-1 + E)*exp(-1) + 76894368849186894
n= 20 -1537887376983737879 + 2432902008176640000*(-1 + E)*exp(-1)
n= 21 -51090942171709440000*(-1 + E)*exp(-1) + 32295634916658495460
n= 22 -710503968166486900119 + 1124000727777607680000*(-1 + E)*exp(-1)
n= 23 -25852016738884976640000*(-1 + E)*exp(-1) + 16341591267829198702738
n= 24 -392198190427900768865711 + 620448401733239439360000*(-1 + E)*exp(-1)
n= 25 -15511210043330985984000000*(-1 + E)*exp(-1) + 9804954760697519221642776
n= 26 -254928823778135499762712175 + 403291461126605635584000000*(-1 + E)*exp(-1)
n= 27 -10888869450418352160768000000*(-1 + E)*exp(-1) + 6883078242009658493593228726
n= 28 -192726190776270437820610404327 + 304888344611713860501504000000*(-1 + E)*exp(-1)
n= 29 -8841761993739701954543616000000*(-1 + E)*exp(-1) + 5589059532511842696797701725484
```

图表 5 利用 sympy 计算出的结果

我们选取在之前计算中差距较大的项，再次计算：

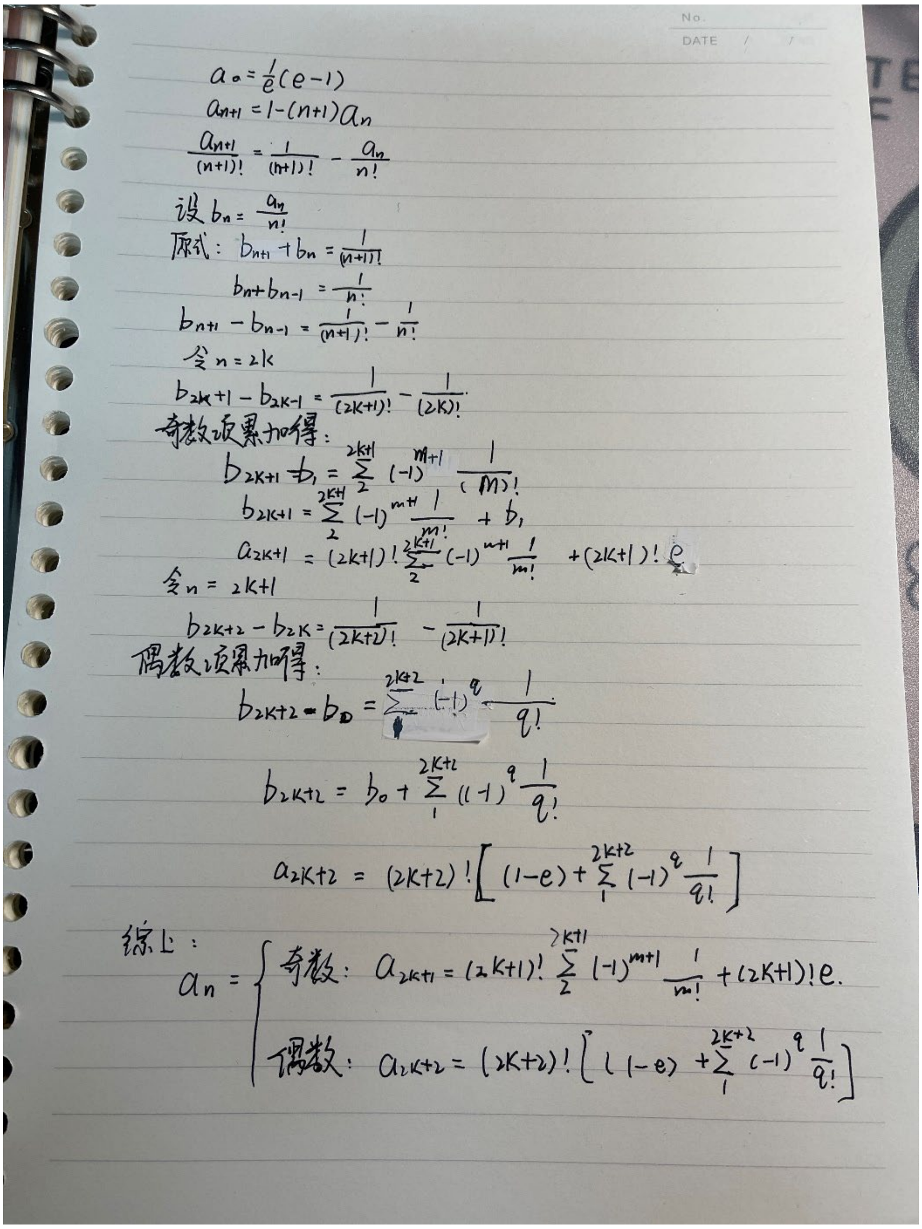
```python
import numpy as np
a=-8841761993739701954543616000000*(-1 + np.exp(1))*np.exp(-1) + 5589059532511842696797701725484
b=-455 + 720*(-1 + np.exp(1))*np.exp(-1)
c=-4047072044694047 + 6402373705728000*(-1 + np.exp(1))*np.exp(-1)
print(a,b,c)
```

```
0.0 0.12680235656154082 0.0
```

图表 6 再次计算某些项的数值

我们发现与我们通过循环语句写出来的结果不符合,所以可以推断出应该是在计算循环的过程中,随着某些项的位数逐渐增多,原本的精度已经不支持计算,从而导致了数值计算不准确。

同样我们也可以推导出该数列的通项公式:



$$a_0=\frac{1}{e}(e-1)$$

$$a_{n+1}=1-(n+1)a_n$$

$$\frac{a_{n+1}}{(n+1)!}=\frac{1}{(n+1)!}-\frac{a_n}{n!}$$

设 $b_n=\frac{a_n}{n!}$

原式: $b_{n+1}+b_n=\frac{1}{(n+1)!}$

$$b_n+b_{n-1}=\frac{1}{n!}$$

$$b_{n+1}-b_{n-1}=\frac{1}{(n+1)!}-\frac{1}{n!}$$

令 $n=2k$

$$b_{2k+1}-b_{2k-1}=\frac{1}{(2k+1)!}-\frac{1}{(2k)!}$$

奇数项累加得:

$$b_{2k+1}-b_1=\sum_{2}^{2k+1}(-1)^{m+1}\frac{1}{(m)!}$$

$$b_{2k+1}=\sum_{2}^{2k+1}(-1)^{m+1}\frac{1}{m!}+b_1$$

$$a_{2k+1}=(2k+1)!\sum_{2}^{2k+1}(-1)^{m+1}\frac{1}{m!}+(2k+1)!e$$

令 $n=2k+1$

$$b_{2k+2}-b_{2k}=\frac{1}{(2k+2)!}-\frac{1}{(2k+1)!}$$

偶数项累加得:

$$b_{2k+2}-b_0=\sum^{2k+2}(-1)^{q}\frac{1}{q!}$$

$$b_{2k+2}=b_0+\sum_{1}^{2k+2}(-1)^{q}\frac{1}{q!}$$

$$a_{2k+2}=(2k+2)!\left[(1-e)+\sum_{1}^{2k+2}(-1)^{q}\frac{1}{q!}\right]$$

综上:

$$a_n=\begin{cases}\text{奇数: } a_{2k+1}=(2k+1)!\sum_{2}^{2k+1}(-1)^{m+1}\frac{1}{m!}+(2k+1)!e \\ \text{偶数: } a_{2k+2}=(2k+2)!\left[(1-e)+\sum_{1}^{2k+2}(-1)^{q}\frac{1}{q!}\right]\end{cases}$$

得出来的通项我们发现后面恰好是泰勒展开式，所以可以得出，该数列在趋近于无穷的时候应该趋近于 0.