

数值分析课程作业 第一章

康豪

2023 年 9 月 7 日

题目 1. 分析单精度计算 $fl(9.4) - fl(9) - fl(0.4)$ 的结果，并进行计算机实践.

解答.

1. 推导过程：单精度具有 1 位符号，8 位指数，23 位小数，故超过位数的数字将被截断，并按照舍入规则进行舍入. 下面根据计算机存储单精度数据的方式及舍入规则推导 $fl(9.4) - fl(9) - fl(0.4)$ 的计算结果.

$$fl(9.4) = (1001.\overline{01110})_2 = (1.001\overline{01110} \times 2^3) = 1.00101100110011001100110 \times 2^3$$

由于 $fl(9.4)$ 末尾为 0，故直接舍去后续位数. 这将导致误差

$$-(0.\overline{01110} \times 2^{-23} \times 2^3) = -0.4 \times 2^{-20}$$

$$fl(9) = 1001, \text{ 不需截断.}$$

$$fl(0.4) = (0.\overline{01110})_2 = (1.\overline{1001} \times 2^{-2}) = 1.10011001100110011001100 \times 2^{-2}$$

$fl(0.4)$ 末尾为 1，且后续位数上不全为 0，故要舍去后续位数后进 1 位，即变为：

$$fl(0.4) = (0.\overline{01110})_2 = (1.\overline{1001} \times 2^{-2}) = 1.10011001100110011001101 \times 2^{-2}$$

这将导致误差

$$-\overline{0.1100} \times 2^{-23} \times 2^{-2} + 2^{-23} \times 2^{-2} = -\overline{0.0110} \times 2^{-24} + 2^{-25} = -0.8 \times 2^{-25} + \times 2^{-25} = 0.2 \times 2^{-25}$$

故最终导致的误差为:

$$-0.4 \times 2^{-20} - 0.2 \times 2^{-25} = -13 \times 2^{-25}$$

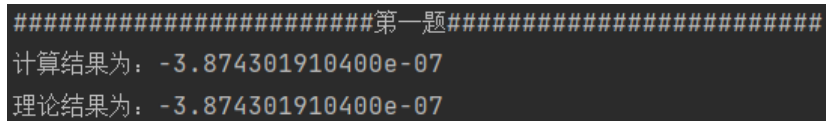
2. 编程实现:

```

1 import numpy as np
2 ans1 = np.float32(9.4) - np.float32(9) - np.float32(0.4)
3 # Python 中保存数据默认双精度, np.float32 方法将数据以单精度形式保存
4 ans2 = -13 * math.pow(2, -25)
5 print(" 计算结果为: %.12e" % ans1)
6 print(" 理论结果为: %.12e" % ans2)

```

3. 结果展示:



```

#####第一题#####
计算结果为: -3.874301910400e-07
理论结果为: -3.874301910400e-07

```

图 1: 问题一 Python 编程结果截图

题目 1 结果分析. 理论分析结果与编程计算结果完全一致. 由于存在截断误差, 计算机的计算结果无法实现完全准确, 单精度计算 $fl(9.4) - fl(9) - fl(0.4)$ 的结果与实际上的 0 小了 10^7 数量级.

题目 2. 设计高效的多项式算法 $P(x) = 1 + 2x^3 + 3x^7 + 4x^{11} + 5x^{15}$. 计算机编程比较直接算法与优化算法的计算时间 (采用双精度计算, $x = 2, x = 2.222222$ 分别循环 10^7 次)

解答.

1. 推导过程直接计算多项式，采用形式

$$1 + 2 * x^3 + 3 * x^7 + 4 * x^{11} + 5 * x^{15}$$

为优化计算方法，将多项式等价变换为下列形式

$$((((5 \times x^4 + 4) \times x^4) + 3) \times x^4 + 2) \times x^3 + 1$$

2. 编程实现：

```

1  import time
2  turns = 10 ** 7  # 循环次数
3  a1 = 2
4  a2 = 2.222222
5
6  x = a1
7  t1 = time.time()
8  for i in range(turns):
9      # 直接计算
10     ans = 1 + 2 * x ** 3 + 3 * x ** 7 + 4 * x ** 11 + 5 * x ** 15
11 t2 = time.time()
12
13 t3 = time.time()
14 for i in range(turns):
15     # 优化算法后
16     ans = (((5 * x ** 4 + 4) * x ** 4) + 3) * x ** 4 + 2) * x ** 3 + 1
17 t4 = time.time()
18 print("x=2, 直接计算多项式所需总时间为：", t2 - t1)
19 print("x=2, 优化算法后计算多项式所需总时间为：", t4 - t3)
20
21 x = a2
22 t1 = time.time()
23 for i in range(turns):
24     # 直接计算
25     ans = 1 + 2 * x ** 3 + 3 * x ** 7 + 4 * x ** 11 + 5 * x ** 15
26 t2 = time.time()
27
28 t3 = time.time()
29 for i in range(turns):
30     # 优化算法后
31     ans = (((5 * x ** 4 + 4) * x ** 4) + 3) * x ** 4 + 2) * x ** 3 + 1
32 t4 = time.time()

```

```

33 print("x=2.222222, 直接计算多项式所需总时间为: ", t2 - t1)
34 print("x=2.222222, 优化算法后计算多项式所需总时间为: ", t4 - t3)

```

3. 结果展示:

```

#####第二题#####
x=2,直接计算多项式所需总时间为:  9.640822887420654
x=2,优化算法后计算多项式所需总时间为:  9.418365955352783
x=2.222222,直接计算多项式所需总时间为:  7.12360405921936
x=2.222222,优化算法后计算多项式所需总时间为:  6.976006746292114

```

图 2: 问题二 Python 编程结果截图

题目 2 结果分析. 直接计算多项式需要 $(1 + 2) + (1 + 6) + (1 + 10) + (1 + 14) = 36$ 次乘法. 将多项式等价变换形式后, 每次计算多项式仅需 $(1 + 3) + (1 + 3) + (1 + 3) + (1 + 2) = 15$ 次乘法. 观察程序运行结果, 发现优化算法后计算多项式的时间相较于优化时明显减少.

题目 3. 推导计算 $I_n = \int_0^1 x^n e^x dx$, 用正向和逆向推导计算 $I_0 \sim I_{20}$, 比较分析两种迭代的误差和稳定性.

解答.

1. 推导过程:

$$I_n = \int_0^1 x^n e^x dx = \int_0^1 x^n d e^x = x^n d e^x \Big|_0^1 - n \int_0^1 e^x x^{n-1} dx = e - n I_{n-1}$$

故正向迭代公式为:

$$I_n = e - n I_{n-1},$$

反向迭代公式为:

$$I_{n-1} = \frac{1}{n}(e - I_n).$$

2. 编程实现:

```

1 import numpy as np
2 print('反向迭代: ')
3 I = 2.7 * 0.5 ** 20 # 初值估计, 约为 0
4 I = np.zeros((22, 1))
5 I[21] = 2.7 * 0.5 ** 20
6 for i in range(21, 0, -1):
7     I[i - 1] = (1 / i) * (np.e - I[i])
8     print("%.12e" % I[i - 1], i - 1)
9
10 print('正向迭代: ')
11 I = np.e - 1
12 for i in range(1, 22):
13     print("%.12e" % I, i - 1)
14     I = np.e - i * I

```

3. 结果展示:

反向迭代:	正向迭代:
1.294418692161e-01 20	1.718281828459e+00 0
1.294419979621e-01 19	1.000000000000e+00 1
1.362547279209e-01 18	7.182818284590e-01 2
1.434459500299e-01 17	5.634363430819e-01 3
1.514609340252e-01 16	4.645364561314e-01 4
1.604263059021e-01 15	3.955995478020e-01 5
1.705237015038e-01 14	3.446845416469e-01 6
1.819827233539e-01 13	3.054900369304e-01 7
1.950999311619e-01 12	2.743615330158e-01 8
2.102651581081e-01 11	2.490280313166e-01 9
2.280015154865e-01 10	2.280015152935e-01 10
2.490280312973e-01 9	2.102651602311e-01 11
2.743615330180e-01 8	1.950999056864e-01 12
3.054900369301e-01 7	1.819830545361e-01 13
3.446845416470e-01 6	1.705190649530e-01 14
3.955995478020e-01 5	1.604958541639e-01 15
4.645364561314e-01 4	1.503481618374e-01 16
5.634363430819e-01 3	1.623630772232e-01 17
7.182818284590e-01 2	-2.042535615583e-01 18
1.000000000000e+00 1	6.599099498066e+00 19
1.718281828459e+00 0	-1.292637081329e+02 20

(a) 反向迭代

(b) 正向迭代

图 3: 问题三 Python 编程结果截图

题目 3 结果分析.

在该问题中，反向迭代显然更加稳定. 考察正向迭代公式

$$I_n = e - nI_{n-1},$$

假设迭代初值存在误差 ϵ ，那么在每次迭代中，误差将被放大 $-n$ 倍. 在计算结果中，正向迭代从 I_{18} 开始出现震荡情况，且每两次相邻迭代异号， I_{19} 迭代结果约为 6.599， I_{20} 迭代结果约为-129，大概是第 19 次迭代的 -20 倍，符合分析结果.