

计算物理第三次作业

白博臣 2022141220036

一、Problem 1

利用 Gregory-Leibniz 公式: $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \dots + (-1)^{n-1} \frac{1}{2n-1}$ 利用 python 程序编写一个无穷级数, 通过设置 n 的取值来改变级数的项数:

$$\pi = 4 * \sum_{n=1}^n (-1)^{n-1} \frac{1}{2n-1} \quad (1)$$

通过 NumPy 程序包进行矢量运算来加快对级数的求解, 详细代码如下:

```
1 import numpy as np
2
3 # 定义级数项数和精度
4 num_terms = 500000
5 precision = 10
6
7 # 计算正序级数和
8 series = 4 * np.sum((-
9 1) ** (np.arange(num_terms+1)) / (2 * np.arange(num_terms+1) + 1))
10 sum_value_01 = np.around(series, precision)
11
12 print('正序计算 Pi 值: \n' + '{:.{}f}'.format(sum_value_01, precision))
13
14 # 计算级数和 (倒序)
15 series_reverse = 4 * np.sum((-1) ** (np.arange(num_terms, -1, -
16 1)) / (2 * np.arange(num_terms, -1, -1) + 1))
17 sum_value_reverse = np.around(series_reverse, precision)
18
19 print('逆序计算 Pi 值:
20 \n' + '{:.{}f}'.format(sum_value_reverse, precision))
```

计算出的结果:

```
正序计算Pi值:
3.1415946536
逆序计算Pi值:
3.1415946536
```

图表 1 Problem 1 的输出结果

通过 python 语言编写的程序我们发现正序计算和逆序计算并没有差别，这是因为无论是正序，还是逆序，计算过程中的精度并没有发现变化；然而对于一些其他语言编写的程序，先对高精度数字的计算也就是逆序计算的精确度会更高，而先对低精度数字也就是正序计算可能会导致后续计算时的精度不够，导致更大的误差。

二、Problem 2

利用 Machin 公式：

$$\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239} \quad (2)$$

通过对 $\arctan \frac{1}{5}$ 和 $\arctan \frac{1}{239}$ 的精度调控，即可得到一定精度下的 π 值。

由于程序中浮点数的有效位数有限（小数点后很长的话会被省略，很大的数如果是小数也会用科学技术法），会损失很大精度，所以用整数来计算 π 。用取模操作代替除法，但是取模对于除法来说本身就是有精度损失的，为了弥补这部分精度，可以先把想精确的位数增加 k ，最后把结果减少 k 位，具体代码如下：

```
1 import sys
2 import time
3
4 # 增加整数字符串转换限制
5 sys.set_int_max_str_digits(10000000)
6
7 n = int(input('请输入圆周率小数点后的位数:'))
8 start_time = time.time() # 计算 pi 计时起点
9 w = n + 10 # 考虑精度，位数增加 k，结果减少 k 位
10 b = 10 ** w # 用整数计算 pi 如 3.14 * 10^2 = 314
11 x1 = b * 4 // 5 # 第一项前半部分
12 x2 = b // -239 # 第一项后半部分
13 sum = x1 + x2 # 第一项的值
14 n *= 2
15 for i in range(3, n, 2):
16     x1 //= -25
17     x2 //= -239 * 239
18     x = (x1 + x2) // i
19     sum += x
20 mpi = sum * 4
21 mpi //= 10 ** 10
22 end_time = time.time()
```

```

23 run_time = str(end_time - start_time)
24 mpi_str = str(mpi)
25 mpi_str = mpi_str[:1] + '.' + mpi_str[1:]
26
27 print('运行时间: ' + run_time)
28 print('计算结果: ', mpi_str)
29 print('')

```

该算法效率较高，实测对 100000 位的计算仅需要 5s 左右

```

请输入圆周率小数点后的位数:100000
运行时间: 5.184260845184326

```

图表 2 Problem2 的输出结果

三、Problem 3

使用 C 语言来判断大端序还是小端序：

```

1.  #include <stdio.h>
2.
3.  int main()
4.  {
5.      int a = 0x11223344;
6.      char *p = (char *) &a;
7.      if(*p == 0x44)
8.          printf("小端序! \n");
9.      if(*p == 0x11)
10.         printf("大端序! \n");
11.     return 0;
12. }

```

```

小端序!
-----
Process exited after 0.1748 seconds with return value 0, 3172 KB mem used.
Press ANY key to exit...|

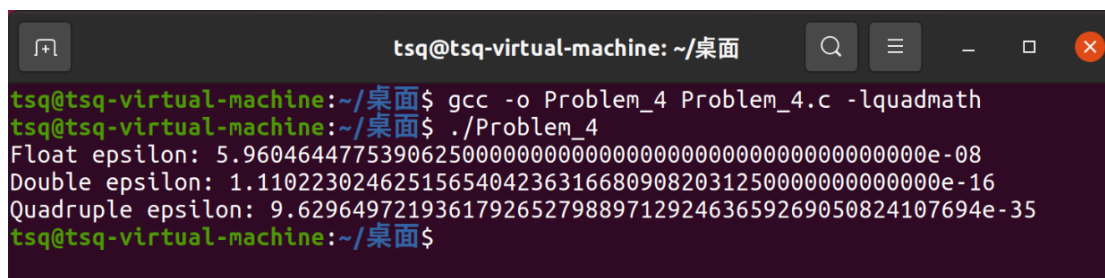
```

图表 3 Problem3 的运行结果

四、Problem 4

使用 gcc 并调用 quadmath.h 程序包，使用如下代码进行确定：

```
1 #include <stdio.h>
2 #include <float.h>
3 #include <quadmath.h>
4
5 int main() {
6     // Calculate float epsilon
7     float f = 1.0f;
8     while (1.0f + f != 1.0f) {
9         f /= 2.0f;
10    }
11    printf("Float epsilon: %.50e\n", f);
12
13    // Calculate double epsilon
14    double d = 1.0;
15    while (1.0 + d != 1.0) {
16        d /= 2.0;
17    }
18    printf("Double epsilon: %.50e\n", d);
19
20    // Calculate quadruple precision epsilon
21    __float128 epsilon = 1.0Q;
22
23    while (1.0Q + epsilon != 1.0Q) {
24        epsilon /= 2.0Q;
25    }
26
27    char buf[1024];
28    quadmath_snprintf(buf, sizeof(buf), "%50.50Qe", epsilon);
29
30    printf("Quadruple epsilon: %s\n", buf);
31
32    return 0;
33 }
```



```
tsq@tsq-virtual-machine: ~/桌面
tsq@tsq-virtual-machine:~/桌面$ gcc -o Problem_4 Problem_4.c -lquadmath
tsq@tsq-virtual-machine:~/桌面$ ./Problem_4
Float epsilon: 5.960464477539062500000000000000000000000000000000000000000000000000e-08
Double epsilon: 1.110223024625156540423631668090820312500000000000000000000000e-16
Quadruple epsilon: 9.62964972193617926527988971292463659269050824107694e-35
tsq@tsq-virtual-machine:~/桌面$
```

图表 4 Problem4 的运算结果