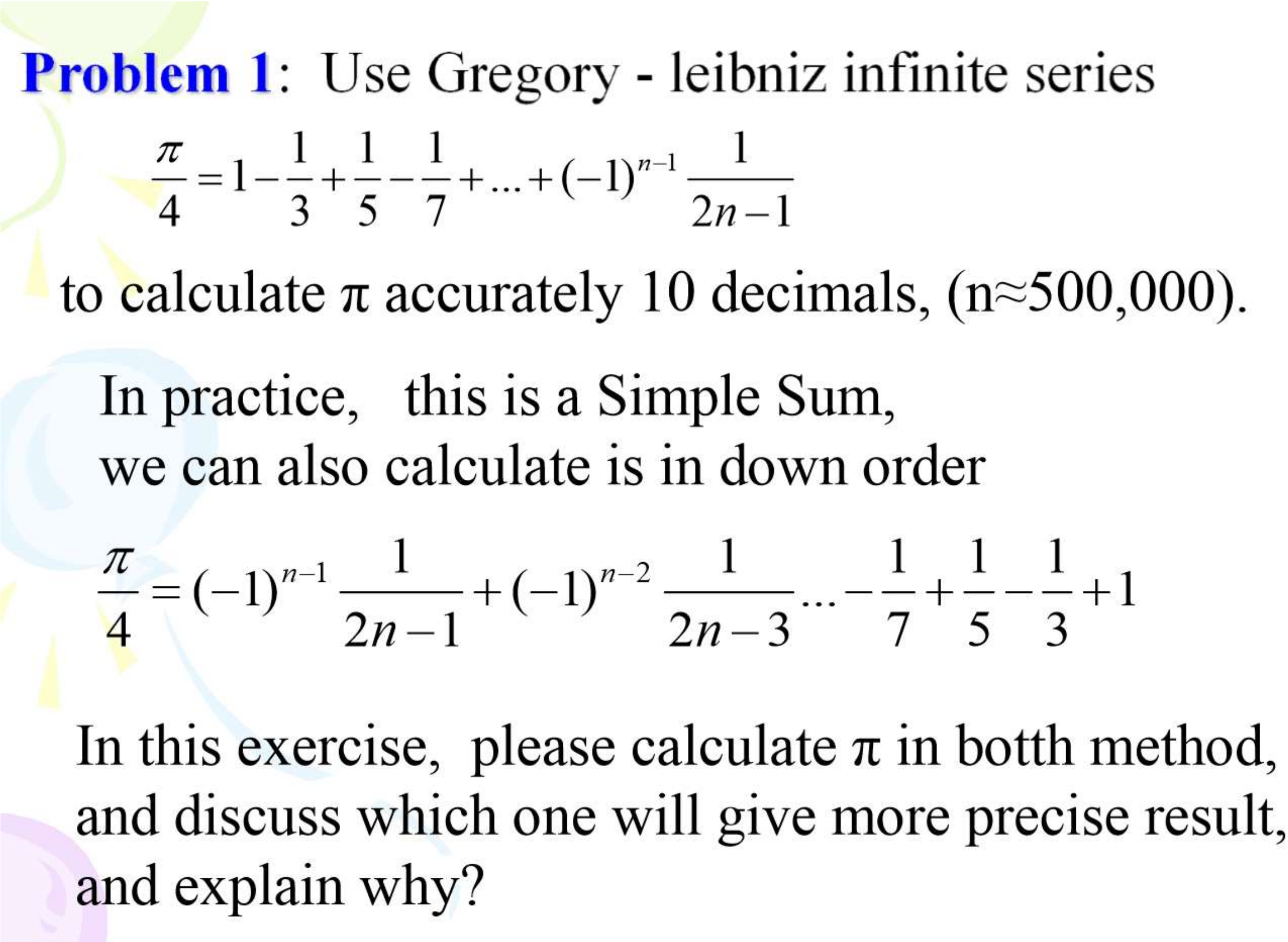
Homework_2

1.Problem_1



1.1. code

代码为**正向求和**，保存在GL_series1.c中

```c
#include <stdio.h>
void main() //problem1_1
{
  double i,k,s,pi;
  k=1;
  s=0;
  for(i=1;i<=500000;i++)
    {
      s+=k*(1/(2*i-1));
      k=-k;
    }
  pi=4*s;
  printf("pi/4=%.20f\n",s);
  printf("pi=%.20f\n",pi);
}
```

代码为**反向求和**，保存在GL_series2.c中

```c
#include <stdio.h>
int main() //problem1_2
{
  double i,k,s,pi;
  k=-1;
  s=0;
  for(i=500000;i>=1;i=i-1)
    {
      s+=k*(1/(2*i-1));
      k=-k;
    }
  pi=4*s;
  printf("pi/4=%.20f\n",s);
  printf("pi=%.20f\n",pi);
}
```

1.2. 结果

正向求和结果为：
pi/4=0.78539766339742300705
pi=3.14159**0**65358**6**9202819
反向求和结果为：
pi/4=0.78539766339744832013
pi=3.14159**0**65358**7**9328053

- 两结果均在第6位(第一位加粗数字)出现误差，与π目前认为的准确值（3.141592653...）有所偏差，该误差应该是n的取值不够大造成的

1.3. 修正

因为上述代码中出现的偏差较大，所以，将n改为500000000，再次运行程序，得到结果如下：

正向求和结果为：
pi/4=0.78539816289731445575
pi=3.1415926515589**2**5782302
反向求和结果为：
pi/4=0.78539816289744834865
pi=3.1415926515589**7**93**3**9461

1.4. 结果分析

- 正向求和与反向求和在第13位（第二个加粗数字）出现不同
- 与π=3.141592653589793**2**38462...相比，反向求和所得结果更准确，在第16位（小数点后第二个加粗数字）出现误差

2.Problem_2

Problem 2: Use Machin's formula

$$\frac{\pi}{4} = 4\arctan\frac{1}{5} - \arctan\frac{1}{239}$$

to compute π to 100 decimal place.

[Hints:

1: $\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots + (-1)^{n-1}\frac{x^{2n-1}}{2n-1}$

2: Quadmath.h]

Machin enjoyed a high mathematical reputation. His ingenious quadrature of the circle was investigated by Hutton, and in 1706 he computed the value of π by Halley's method to one hundred decimals places.

2.1.code

代码保存在M_formula.c中

```
#include <stdio.h>
#include <math.h>
#include <quadmath.h>

__float128 arctan(__float128 x) //计算arctan的函数
{
  __float128 s,i,k;
  s=0;
  k=1;
  for(i=1;powq(x,i)/i>powq(10,-100);i+=2) //用精度判断循环结束
    {
      s+=k*(powq(x,i)/i);
      k=-k;
    }
    return s;
}

void main() //主函数
{
  __float128 t,pi,i1,i2,i3;
  int l;
  i1=1.0Q;
  i2=5.0Q;
  i3=239.0Q;
  t=4*arctan(i1/i2)-arctan(i1/i3);
  pi=4*t;
  printf("pi/4=%.40Qe\n",t);
  printf("pi=%.40Qe\n",pi);
}
```

2.2.结果

pi/4=7.8539816339744830961566084581987569936977e-01
pi=3.14159265358979323846264338327950**2**7974791e+00

2.3.结果分析

π=3.14159265358979323846264338327950**2**88419716939...
通过上述代码计算出的π在第34位（加粗数字）出现误差

3.分析

3.1.计算机中浮点数的存储

根据IEEE754规定，任意一个浮点数可以表示成如下形式（以64bits为例）

双精度浮点数存储模型 https://blog.csdn.net/qq_36391130

> 因为计算机中浮点数用二进制数存储，所以β(base)为 2

- S(sign) 符号位：当S=0时，该浮点数为正数；S=1时，该浮点数为负数
- E(exponent) 指数
- M(mantissa) 有效数字：对于二进制来说1<M<2

3.2.上述π计算结果误差分析

- 对于64bits的数据，M=52bits，其精度为2^{-52}(=2.220446049250313e-16)，所以对于double型数据，小数点后16位都是准确的

> 但上述结果在13位就出现误差，可能是由于在计算过程中的近似处理造成的误差

- 对于四精度(128bits)的数据，大概会在小数点后第34位出现误差

3.3.补充

> 精度的定义为： 1 与大于 1 的最小浮点数之差

根据IEEE的规定，浮点数的表示方法为S+E+M(sign+exponent+mantissa),其中指数可以为负数，所以对于float型数据能够表示的最小数据可以小于精度
（下代码保存在test.c中）

```c
#include <stdio.h>
#include <math.h>
void main()
{
    float a,b,c,d,min,minTest;
    a=pow(2,-23); //float型数据的精度
    b=pow(2,-126);  //计算通过指数能表示的最小数据
    min=pow(2,-149);  //计算计算机能表示的最小数据
    minTest=pow(2,-150);
    c=pow(10,-20);  //检验由于计算机二进制存储数据的方法带来的误差
    d=0.100001;
    printf("a  =%.50f\n",a);
    printf("b  =%.50f\n",b);
    printf("min=%.50f\n",min);
    printf("minTest=%.50f\n",minTest);
    printf("c  =%.50f\n",c);
    printf("d  =%.50f\n",d);
}
```

运行结果为:

a =0.00000011920928955078125000000000000000000000000000

b =0.00000000000000000000000000000000000001175494350822

min=0.00000000000000000000000000000000000000000000140130

minTest=0.00000000000000000000000000000000000000000000000000

c =0.0000000000000000000**0999999968265522538896788746349**

d =0.10000**09998679161071777343750000000000000000000000**

分析

- float型数据能表示的最小数据(min)远小于其精度
- 由c, d结果分析可得，两者所得到的数据都不是其准确值，是因为在计算机中存储浮点数时需将其（我们通常使用十进制）转换为二进制数据，二进制浮点数和十进制浮点数不是一一对应的，所以会出现上述误差（加粗数字部分）