Pi 值的估算

基于蒙特卡洛算法

实验报告

一、计算方法：

计算π的值有很多种方^，具有代表性的如下：

① 采用直接测量的方法，寻找生活中的正圆，测量周长和直径，并且通过公式：

$$\pi = \frac{C}{d}$$

计算可得。

② 使用无穷级数来计算π的值：

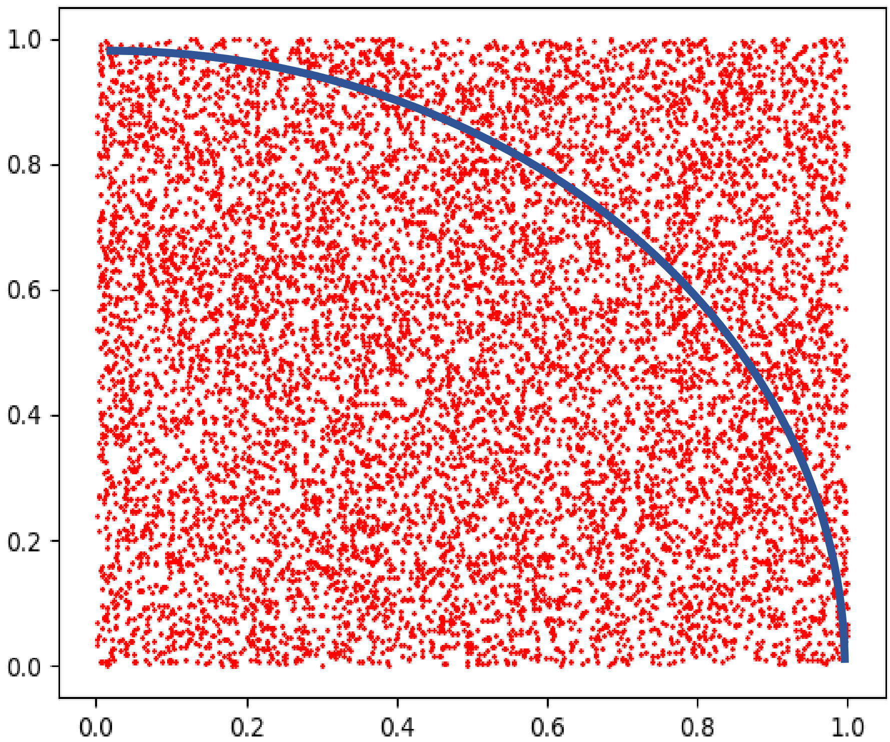
格雷戈里-莱布尼兹无穷级数：$\pi = 4 \times \left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} \ldots\ldots\right)$

Nilakantha 级数：$\pi = 3 + \frac{4}{2\times3\times4} - \frac{4}{4\times5\times6} + \frac{4}{6\times7\times8} - \frac{4}{8\times9\times10} \cdots\cdots$

③ 蒙特卡洛算法估算：

类似于投针问题，向 1×1 的正方形中随机落点，以左下角为圆心，绘制 1/4 圆，半径为 1，记落在圆内的点数量为 m，总共投掷 n 次，则：

$$\pi = \frac{m}{n} \times 4$$



④ 使用极限来计算π的值：

$$x \times \sin\frac{180}{x} \to \pi$$

当 x 越大时，所计算的值越趋近于π的真值。

⑤ ……

二、代码解读：

在这里，我们根据题意选择了方法③，通过随机落点的方法来估算π的值。

具体的代码实现解析如下：

```
#define N (unsigned long long)10000000
#define THREAD_NUM 2
#define PRE_THREAD_NUM (N/THREAD_NUM)
```

如上是开头的宏定义，其中 N 代表了需要运算的总次数，THREAD_NUM 代表了线程数，PRE_THREAD_NUM 代表了每个线程需要计算的次数。

```
unsigned long long data[THREAD_NUM]; // 用来记录所有的数据，在开始之初，用于记录种子
```

而这是本次实验中使用的唯一一个全局变量，因为对同一个全局变量的访问可能导致需要加锁、多线程变为串行执行等多方面的问题，所以本次实验中尽量减少了全局变量的使用，这个全局数组也不存在多个线程对同一个数据的共享，因为每个线程都独自访问其中的某一个元素，而不存在冲突。

```
 for (int i = 0; i < THREAD_NUM; ++i)
    {
        gettimeofday(&time, NULL);
        data[i] = time.tv_usec;
#ifdef GRPDEBUG
        printf("seed %d : %llu\n", i, data[i]);
#endif

        pthread_create(&my_thread[i], &attr, compute_pi, (void*)&data[i]);
    }
```

上述部分为创建线程的部分，通过 gettimeofday 函数得到当前时间的微秒级

数据，使用数据中的微秒部分作为随机种子，存入全局变量中，向线程函数传参，并且循环创建线程。

```c
void *compute_pi(void *ptr)
{
    unsigned long long seed_long = 0; // 种子
    unsigned int seed = 0;
    unsigned long long *count = (unsigned long long *)ptr; // 将参数进行类型转换
    seed_long = *count; // 此时这个位置上放置的是种子
    seed = (unsigned int)seed_long;
    double x = 0, y = 0;
    *count = 0;
    for (unsigned long long i = 0; i < PRE_THREAD_NUM; ++i)
    {
        x = (double)(rand_r(&seed)) / (double)(RAND_MAX);
        y = (double)(rand_r(&seed)) / (double)(RAND_MAX);
        if (x * x + y * y <= 1)
            ++(*count);

    }
    pthread_exit(0);
}
```

这是每个线程调用的π值计算函数，传参的指针是一个解释为 unsigned long long 类型，之后进行截断，取出 unsigned int 类型的数据，作为随机种子，在 rand_r 函数中调用该种子，通过除法随机生成了落点的 x/y 坐标，位于 0~1 之间，如果落于 1/4 圆内，则计数加一。

```c
for (int i = 0; i < THREAD_NUM; ++i)
    {
        pthread_join(my_thread[i], NULL);
        total_data += data[i];
        printf("Thread %d : %llu\n", i, data[i]);
    }

    double pi = ((double)total_data / (PRE_THREAD_NUM * THREAD_NUM)) * 4.0;

    time_end = GetTime();
```

如上的部分为等待函数，等待所有的线程结束，并逐步将线程中的数据合并到

总数据中，记录落点在 1/4 圆内的总次数，进行π值的计算，即可。

最后进行输出操作，完成了所有需求。

三、结果分析：

首先对计算出的π值结果进行误差的分析：

这里的π值计算分析均采用双线程的方式，运行程序

得到数据如下：

计算次数	1000	100000	10000000	1000000000	10000000000	100000000000
1	3.1280000000	3.1354000000	3.1407744000	3.1416004080	3.1415883340	3.1415880482
2	3.2520000000	3.1339200000	3.1410192000	3.1415871880	3.1415863656	3.1415876920
3	3.0920000000	3.1463600000	3.1418416000	3.1415861560	3.1415865584	3.1415872368
4	3.0680000000	3.1366400000	3.1416608000	3.1415863440	3.1415864784	3.1415876562
5	3.2240000000	3.1387600000	3.1410916000	3.1415918200	3.1415878632	3.1415880986

对于精确值，有：

$$\pi=3.1415926536$$

所以，可以分析得到，对于不同的实验次数，平均的误差值为：

其中ε_n代表随机次数为 n 的时候，所产生的较标准值的误差

$$\varepsilon_{1000} = 0.0659185307$$

$$\varepsilon_{100000} = 0.0052835922$$

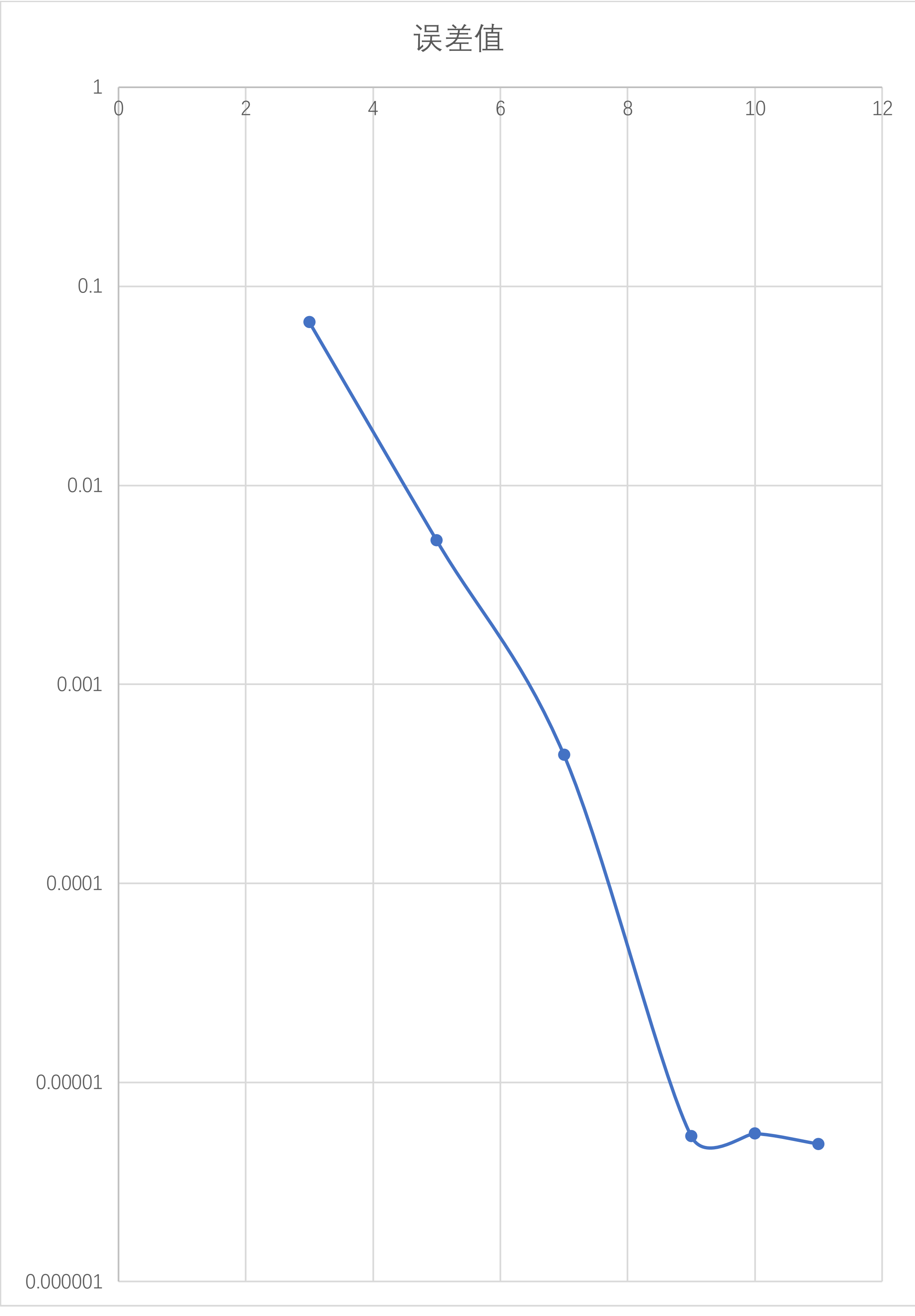
$$\varepsilon_{10000000} = 0.0004419707$$

$$\varepsilon_{1000000000} = 0.0000053722$$

$$\varepsilon_{10000000000} = 0.0000055337$$

$$\varepsilon_{100000000000} = 0.0000049072$$

对误差作图分析，可以得到：



经过分析，可以得到，当采样的数据量越来越大的时候，与标准π值比较的误差就会越小，当随机量为 1000 时，误差有处于十分之一（0.1）的量级，而当随机量达到 100000000000（一千亿）的时候，误差就只有十万分之一（0.00001）的量级了。

但是这种精度的提升，是逐渐趋于困难的（收益低），耗费了10^8量级的计算量，只换来了10^4倍的精度提升。而事实上，在数据量达到 1000000000（十亿）的时候，误差的量级就已经是 0.00001 了。而之后花费的10^2的计算量，都只在精度的系数上进行了一定的优化，而对于量级没有影响。

综上，计算量增长所带来的收益随着量级的增大，将会越来越低。

下面进行多线程运行花费时间的分析：

得到的数据如下：

计算总数为 1000000000（十亿），核心数为 4

线程数	1	2	4	8	16	32
第一次	21.8689620495	13.6654210091	11.5336351395	11.8227717876	8.0299141407	6.7820470333
第二次	21.1879270077	13.7250790596	11.5412859917	11.8550839424	7.6880569458	6.9224150181

线程数	64	128	256	512
第一次	6.7208731174	6.0240600109	5.8938031197	5.6693708897
第二次	6.3678600788	5.9190571308	5.9650270939	5.8405280113

线程数	1024	2048	4096	8192
第一次	6.4191820621	6.4108948708	6.8471639156	7.4112370014
第二次	6.3599889278	6.3983380795	6.6739878654	8.9964640141

将数据分为三段进行研究，1~4 作为一段，4~512 作为一段，512~8192 作为一段。

① 对于 1~4，因为核心数为 4，所以按照理论，当线程数少于或等于核心数的时候，随着线程数的增加，由于线程可以独立运行在各个核心上，所以运行速度应该会不断增快，但是并未达到预期的，四线程时间为单线程的 1/4，因为按照理论来说，四个线程应该可以独立运行在四个核心上，所以时间应该变为 1/4。

具体的误差原因在后续的分析中将会提到。

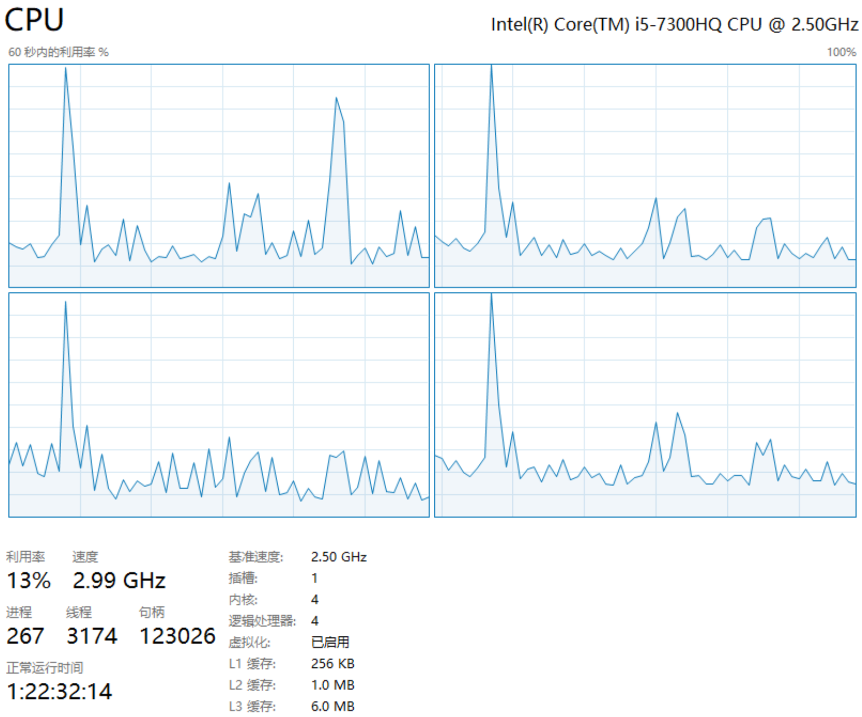
② 这是一个非常奇怪的曲线，从 4 线程开始到 512 线程，按照理论，当线程数大于内核数，应该不再具备并行所带来的时间收益，同时由于线程的切换，应该带来更多的额外时间消耗，而导致时间上长于 4 线程。

但是事实是不符合的，事实上，直到 512 线程，时间消耗都一直在降低，为此，我有一个猜想:

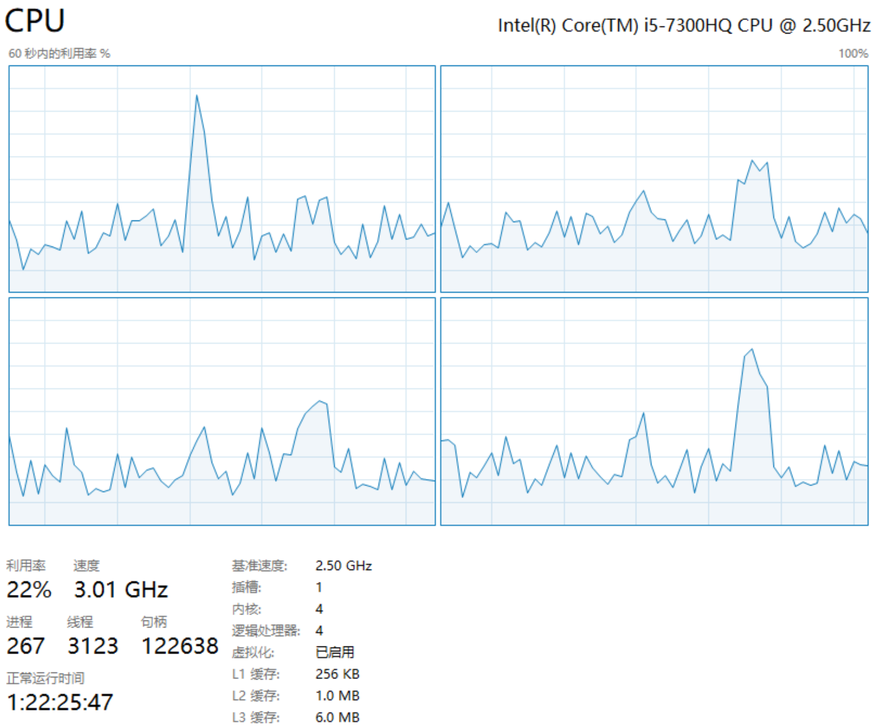
程序并未在执行时完全占用 CPU 核心的资源，而是“适度使用”。

这种适度使用就表现在，既然我使用 10%的 CPU 就可以“较好”的完成一个任务，那我就没有必要使用 100%的 CPU 资源来进行这个计算了。

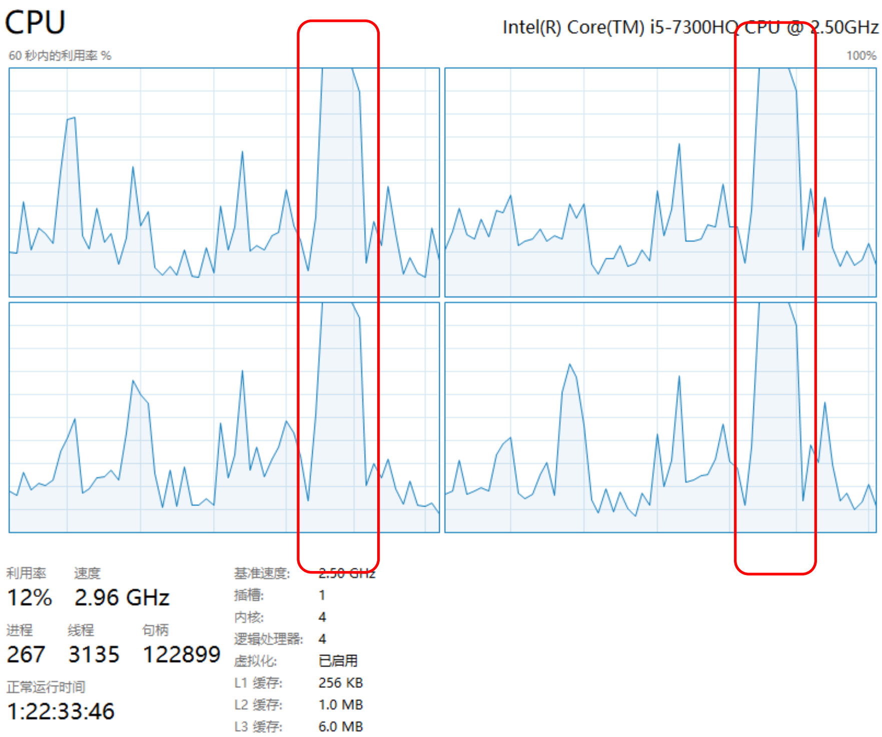
首先，CPU 在不启动该程序的情况下，有占用率:



首先，我们观察 CPU 在 4 线程的情况下的占用率：



然后观察在 512 线程下的 CPU 利用率：



可以看到，我的推测应该是正确的，CPU 在 4 线程的时候，并没有“使出全力”，而是“不用认真就过去了”了的机制。直到 CPU 达到 512 线程的时候，程序运行的压力让 CPU 意识到“该努力”了，于是使出了 100%的力气，所以就达到了理论上 4 线程分别运行在 4 个 CPU 核心上的时候的速度，即 21s/4 ≈ 5+s。

③ 当线程数高于 512 时，由于线程切换的代价，以及 CPU 占用率已经到达了 100%，所以没有在核心上的提升空间，同时还加入了线程切换的附加时间，所以从 1024 线程开始，程序所花的总时间就开始缓慢增长，增长应该来源于线程切换所花费的时间，即系统级代码的增添执行。

四、其他分析：

回归到π值的计算结果中：

可以看到，当计算量为 100000000（十亿）的时候，计算的五次结果中，相较于π的标准值，结果有的比π大，有的比π小。

但是到了 10000000000（一百亿）和 100000000000（一千亿）的数据量时，所有的 5 × 2 = 10 个结果都在π值以下，也就是小于π值。个人认为，十次计算都偏小的概率是相对而言较低的，估算一下可以知道这样的随机事件发发生的概率仅为 0.0009765625，个人认为不可能是单纯的随机导致的。

对此，笔者的猜测是:

一百亿和一千亿的数据，相较于十亿及以下的数据，越过了一个“临界值”，这个数据量的“临界值”，就是计算机的伪随机数出现规律的界线。超过了这个界线，计算机的伪随机数就可能有序列与前面的部分序列出现了循环重复。所以也就是说，在一百亿和一千亿这样的数据量中，所随机出来的序列可能已经包括了所有可能的随机数序列了。

那么，有没有可能是:

① 计算机在取伪随机数的时候，其实在概率上有一定的偏差? 并不是真的完全随机分布的（即每个数字的概率完全相等)，而是在概率上有一定的出入，这种出入在映射到对应的 double 变量的时候，就可能导致落点在园内外的概率出现了不均等，而且圆外的概率要略高于理论值，这就导致π的值计算结果永远偏小。

② 还有可能是在计算的时候导致的误差，rand_r 产生的是所有论域上的随机数，之后除以 RAND_MAX 的最大值，就可以得到 0~1 之间的小数。有没有可能是，随机数其实是完全按照随机规则分布的（或者十分近似于)，但

是当映射到 0~1 的小数的时候，由于 x/RAND_MAX 的映射方法可能导致的 double 的误差，在平方之后误差增大了，从而导致，两个 double 的平方和有趋向圆外的趋势，所以最终的结果永远偏小。

综上，这是个人对于大量级情况下的计算所得到的结果偏小的猜测。

五、CPU 噪声随机数实现：

伪随机数终究是伪随机数，这里我们考虑，使用 CPU 的噪声作为随机数的来源，有如下的代码，就不做具体的分析了：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <pthread.h> // 需要这个库的支持，因为 Linux 本来是不支持多线程的
#include <fcntl.h>
// #define GRPDEBUG
#define N (unsigned long long)100000000000
#define THREAD_NUM 1
#define PRE_THREAD_NUM (N/THREAD_NUM)
double GetTime()
{
    struct timeval time;
    gettimeofday(&time, NULL);
    return (double)time.tv_sec + (double)time.tv_usec / 1000000.0;
}
int main()
{
    struct timeval time;
    double time_begin, time_end;
    gettimeofday(&time, NULL);
    unsigned int seed = time.tv_usec;

```

```c
    unsigned long long count = 0;
    int randNum = 0;
    time_begin = GetTime();
    int fd = open("/dev/urandom", O_RDONLY);
    if (fd == -1)
        exit(-1);
    double x = 0, y = 0;
    unsigned int seed1, seed2, seed3, seed4;
    for (unsigned long long i = 0; i < N; ++i)
    {
        (unsigned int)read(fd, (char *)&seed1, sizeof(int));
        (unsigned int)read(fd, (char *)&seed2, sizeof(int));
          x = (double)seed1 / __UINT32_MAX__;
        y = (double)seed2 / __UINT32_MAX__;
        if (x * x + y * y <= 1)
            ++count;
    }
    close(fd);
    time_end = GetTime();
    printf("%llu\n", count);
    double pi = ((double)count / N) * 4.0;

    printf("Compute %llu Times\n", N);
    printf("Time : %.10f sec\n", time_end - time_begin);
    printf("Pi : %.10f\n", pi);
    return 0;
}
```

运行下来的数据如下：

计算次数	100000000（一亿）	1000000000（十亿）	10000000000（百亿）
1	3.1418435600	3.1415742680	3.1416175172
2	3.1414039200	3.1416423800	3.1415770272
3	3.1413649600	3.1416470040	
4	3.1417091600		
5	3.1414744800		

一亿数量级的误差为：0.0001804027

十亿数量级的误差为：0.0000408208

百亿数量级的误差为：0.0000202450

经过比较可以发现，在十亿以及百亿级数据以上，并没有在精度上优于前述的

伪随机数所得到的结果。所以不难总结得到，这虽然是真随机数，但是结果上并没有更加精确，可能的原因是 CPU 的噪声所得到的数据有一定的规律，从而导致了误差比预期的要更大。

所以，CPU 噪声产生的随机算法，并不是一个更优的方法，同时，花费的时间远远大于伪随机数。