武汉大学·课程结课论文

计算物理

用随机过程计算圆周率π

- 姓名：李东旭
- 学号：2015301510021
- 学院：物理科学与技术学院
- 联系方式：andu@whu. edu. cn
- 电话：15071339677
- 授课教师：蔡浩

2017 年 12 月 28 日

用随机过程计算圆周率π

内容摘要

介绍 Buffon 用随机的投针实验估算圆周率的方法，推导针与线相交的概率与圆周率π之间的函数关系。利用蒙特卡罗方法模拟 Buffon 的投针实验，模拟投针1×10^9次后得到的π值为3.142120…，其精度达到了 4 位有效数字。

分析该蒙特卡罗过程产生误差的原因，包括空间方格离散化所带来的误差、纸张不是无穷大所带来的误差、投针次数有限所带来的误差，并且提出了在一定精度内减小误差的方法。应用这些方法后，精度得到提升。

利用基于 Halton 序列的准蒙特卡罗方法再次模拟这个实验，对比发现基于伪随机数的蒙特卡罗方法的精度要优于准蒙特卡罗方法。这是因为 Halton 序列并不是随机的，而是一种低差异序列。

关键词： 随机过程 圆周率 Buffon 投针 蒙特卡罗模拟

引言

随机过程在自然界和生活中普遍存在。比如抛一枚硬币，可能正面落地，也可能反面落地。随机过程看似不可预测，实则在偶然性中蕴含着必然的规律。比如抛硬币时我们不知道某一次将会是正面落地还是反面落地，但我们知道很多次的抛掷后正面落地的情况一定占约50%。利用随机过程设计一些实验或者模拟，可以计算非常复杂的相互作用下系统的行为。本文利用这种思想来计算圆周率π。关于π的计算方法已经非常成熟，并不需要用蒙特卡罗方法来计算。不过，用本文的方法来计算π具有鲜明的物理意义，对于理解蒙特卡罗方法也是一个很好的例子。

1. Buffon 投针问题

1.1 针线相交概率与π的函数关系[1][2]

设有一张很大的纸，上面画着一些间距为l的平行线。将一些长度也为l的针随意地投到纸上，统计针与线的相交概率P。

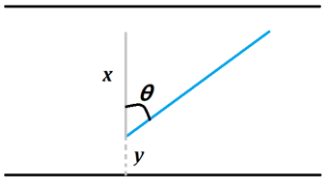

图 1

由于针的长度不大于平行线的间距，所以对一根针来说，与它相关的只有它附近的平行线。如图 1①，设“针眼”端与下方的线距离为y，设针在垂直于平行线方向上的投影（投影区分正负）为x。

①该示意图，及图 6 和图 9，使用 Windows 画图工具配合 Photoshop 绘制。

则

- 针与上方的线相交 $\Leftrightarrow y+x>l \ \Rightarrow x>l-y$
- 针与下方的线相交 $\Leftrightarrow y+x<0 \ \Rightarrow x<-y$

对于一个给定的y，设x处于x到$x+dx$之间的概率为$\rho(x)dx$,其中概率密度$\rho(x)$可以用以下方法求出：

设针与竖直方向的夹角为θ（图 1），则

$$x=l\cos\theta \tag{1}$$

微分得：

$$dx=-l\sin\theta d\theta \tag{2}$$

$$d\theta=\frac{dx}{|-l\sin\theta|} \tag{3}$$

（取绝对值是因为我们这里只关心dx的大小）

设针的夹角θ处于θ到$\theta+d\theta$之间的概率为$\rho_0(\theta)d\theta$

显然

$$\rho_0(\theta)d\theta=\frac{1}{\pi}d\theta \tag{4}$$

于是由二者的对应关系得

$$\rho(x)dx=\rho_0(\theta)d\theta=\frac{1}{\pi}d\theta=\frac{1}{\pi}\frac{dx}{|-l\sin\theta|}=\frac{1}{\pi}\frac{dx}{\sqrt{l^2-x^2}} \tag{5}$$

于是对于一个给定的y， x处于x到$x+dx$之间的概率为

$$\rho(x)dx=\begin{cases}\frac{1}{\pi}\frac{dx}{\sqrt{l^2-x^2}} & ,\text{if}-l<x<l\\ 0 & ,\text{otherwise}\end{cases} \tag{6}$$

于是针与平行线相交的概率为

$$P(y)=\int_{l-y}^{l}\rho(x)dx+\int_{-l}^{-y}\rho(x)dx \tag{7}$$

这个定积分的结果为

$$P(y)=\frac{1}{\pi}\left[\arccos\left(\frac{l-y}{l}\right)+\arccos\left(\frac{y}{l}\right)\right] \tag{8}$$

当y变动时，由于y在$(0,1)$之间等概率取值，概率密度为$\frac{1}{l}$，所以相交概率为

$$P=\int_0^l P(y)\times\frac{1}{l}dy=\frac{1}{\pi l}\left[\int_0^l\arccos\left(\frac{l-y}{l}\right)dy+\int_0^l\arccos\left(\frac{y}{l}\right)dy\right]=\frac{2}{\pi} \tag{9}$$

于是得到

$$\pi=\frac{2}{P} \tag{10}$$

我们只要通过实验或模拟得到针与线的相交概率P，就可以计算圆周率π。

1.2 用 Buffon 投针方法估算π

1777 年，Buffon 提出了用投针来估算π值的方法。[3]已是耄耋之年的他常常邀请自己的朋友来家中做客，期间便邀请客人一起玩投针游戏。客人们疑惑不解，但最后 Buffon 通过简单的演算得到了圆周率的值，客人们都惊讶不已。

历史上有很多学者做了 Buffon 投针实验，其中一些著名的实验列于下表。[4]

表 1 历史上的 Buffon 投针实验

实验者	年份	投针次数	π的实验值
Wolf	1850	5000	3.1596
Smith	1855	3204	3.1553
Fox	1894	1120	3.1419
Lazzarini	1901	3408	3.1415929

Lazzarini 的结果是值得怀疑的，3000 多次的投针次数并不能达到如此高的精度，他的结果很可能是某次实验的偶然结果。这也说明了一个问题，Buffon 投针法并不是一种高效的计算π值的方法。更好的方法是用下面这个级数展开：

$$\arctan x = x - \frac{x^2}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} + \cdots \tag{11}$$

令$x = 1$得

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} + \cdots \tag{12}$$

用无穷级数来计算π不仅精度更高，而且速度更快。但是 Buffon 投针法的意义十分重大——它被认为是用随机过程估算定值的第一种方法，直到 20 世纪由 John von Neumann 和 Stanislaw Ulam 合作将这种方法命名为 Monte Carlo 方法。[3]一般认为 Monte Carlo 方法的起源就在这里。

2. 模拟蒲丰投针问题

2.1 算法简介

2.1.1 变量

设计的变量共有 3 个：

- l ： 针的长度
- $size$ ： 纸张的大小
- N ： 投针的次数

长度的单位均为1，代表空间方格离散化的最小可分辨距离。纸张不能选得过小，否则当我们将空间离散成一个个方格时，精度太低不能满足计算要求；针的长度必须比纸张的边长小。（当然，你也可以赋予单位以意义，比如，1mm。）

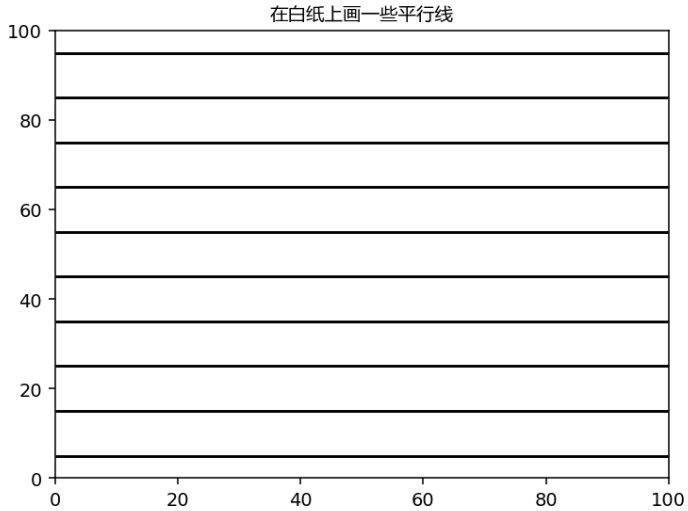

图 2

2.1.1 投针的准备工作

首先在纸张上画一些平行线。（上页图 2[①]）虽然纸张是二维的，但我们并不需要引入一个矩阵来表示纸上的图形，只需要一个一维的数组就可以表示这些平行线，数组的各个元素代表空间位置，元素为 1 代表这一行有平行线，元素为 0 代表这一行是空白。

矩阵也可以表示二维图形，但是一个矩阵可以容纳的信息远远多于我们此时需要的信息。在可以用一维数组的情况下，我们应该避免使用矩阵，后者会占用大量的内存，限制我们进行更大规模的计算。

2.1.2 确定针在纸张上的位置

生成两个随机数，组成一个数对(x_0, y_0)，代表“针眼”端的坐标。$x_0, y_0 \in (l, size - l)$，可以保证“针眼”和“针尖”都落在纸的范围内。下文我们可以看到，x_0其实是多余的，省略它可以提高计算速度。

接下来，我们要确定“针尖”端在纸上的位置。生成一个$(0,360)$范围内的浮点随机数，用θ标记，单位为度，表示针与平行线所夹的角（角的量取与极坐标量取极角的方向相同）。则针尖的位置(x, y)由下式给出：

$$x = x_0 + l\cos\theta \tag{13}$$

$$y = y_0 + l\sin\theta \tag{14}$$

同样，下文我们将看到y是不需要计算的，省略它可以提高计算速度。

2.1.3 判断是否相交

首先判断y_0与y是否相等，如果不相等，则它们之间有一些整数。遍历这些整数，2.1.1 节给出了一个数组，以这些整数为序号索引数组中的元素，如果所有元素都为 0，则这根针没有与平行线相交；如果其中有任何一个元素为 1，则这根针与平行线相交了。如果y_0与y相等，那么查找数组中编号为y_0的元素，如果它为 1，则整根针躺在平行线上；如果它为 0，则这根针没有与平行线相交。

于是我们发现，判断针是否与平行线相交，只要y_0与y就够了，“针眼”和“针尖”的横坐标x和x_0都是多于的，不需要计算。

但是，需要注意的一点是，遍历这些整数的时候，只能包含一个端点，否则我们算出的相交的概率会大于实际的相交概率。但是我们并不能直接去掉最后一个整数，因为这些整数对应的是针在垂直于平行线方向上的投影的大小。由下式可以算出，长度为单位 1 的“针尖”在垂直于平行线方向上的投影的大小为[②]

$$\frac{2}{\pi}\int_0^{\frac{\pi}{2}}\cos\theta d\theta = \frac{2}{\pi} \approx 0.6366197723675814 \tag{15}$$

于是我们可以生成一个 0 到 1 之间的随机数，当这个随机数小于上面的值的时候，去掉最后一个整数，否则保留最后一个整数。如此，便实现了投影值的修正。

2.1.4. 重复实验

一般来说，一张纸上要投上千根针，这称为一次实验，可以通过循环结构实现。在一次实验中，需要统计相交的总次数n，而n/N即为相交概率。

图 3（下页）是$size = 100; l = 10; N = 50$的示意图。实际计算中并不需要画出这样

①该图，以及图 3，图 4，图 5，图 7，图 8，图 10，图 11-19 使用 Python 的软件包 Matplotlib 绘制。
②这里计算用到了π，事实上计算机计算三角函数也需要用到π，不过没有关系，这种方法可以视为自洽演算。

的图，一方面因为N非常大，针都重叠在一起不能分辨；另一方面，画这样的图需要计算横坐标x与x_0，加重了计算负担。

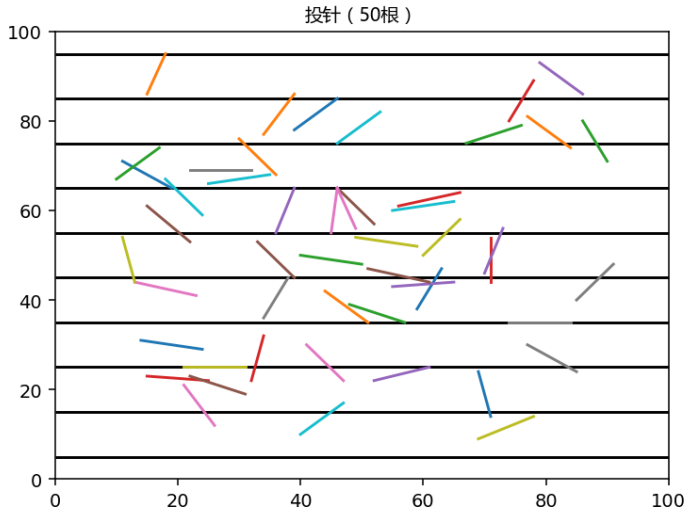

图 3

2.2 模拟结果

设定针长$l = 300$, 纸张大小$size = 1500$，模拟投针实验。（之所以这样选择针长和纸张大小，是建立在误差分析的基础上的。这样的参数可以在精度与效率之间达到平衡。误差分析见下小节内容。）其结果如下表：

表 2 模拟 Buffon 投针计算π值

投针次数	π的计算值	计算时间(s)†
$\mathbf{1 \times 10^6}$次	3.139387…	40
$\mathbf{1 \times 10^7}$次	3.142689…	412
$\mathbf{1 \times 10^8}$次	3.142134…	2614*
$\mathbf{1 \times 10^9}$次	3.142120…	25618*

†计算时间基于 Intel Core i7-5500U，可能同时运行有其它程序，因此计算时间仅供参考。

*后两个实验使用并行计算对程序进行了优化，在一定程度上缩短了计算时间。该实验的并行算法十分简单，以最后一个实验为例，只需要将10^9次投针拆分成 4 个2.5×10^8次投针，分配给 CPU 的 4 个核心进行，每个核心返回一个π值，取平均即可。如果多个核心分配的实验次数不相等，最后要取加权平均值。

在$\mathbf{1 \times 10^9}$次的投针次数下，π的计算值可以精确到 4 位有效数字。而进一步增加投针次数，已经超过了 PC 的运算极限。

事实上没有必要进一步增加投针次数，因为这个程序在于展示 Monte Carlo 的核心思想，而不在于取得极高的精度。用这种方式计算定值，有效数字的位数大约与$\sqrt{N}$成正比，后期计算效率会越来越低。需要精密计算时，应该采用其它方法，比如，使用 Machin 公式。还需要从底层设计使程序支持超长的浮点数（100 位以上）。[5]

虽然 Monte Carlo 方法在类似的问题上没有精度优势，但是很多复杂的问题（比如 RNA 分子的折叠，属于强关联问题），只有用类似的方法才有计算的可能。

3. 误差分析

3.1 针的长度引起的误差

控制投针次数$N = 10000$和和纸张大小$size = 1200$不变。依次选择l在 2 到 600 之间的 200 个值做蒙特卡罗模拟，将每相邻的 10 个l值得到的结果取平均，与标准π值相比较，做出计算的π值与随针的长度的变化图（图 4）。以标准π值为基准计算其相对误差，然后相邻的 10 个l值的误差取平均，做出相对误差随针的长度变化图（图 5）。

（注意：红色的误差曲线的计算是取了绝对值后相加平均的，所以即使某次计算的π值很接近正确值，也不意味着它的误差小。下同。）

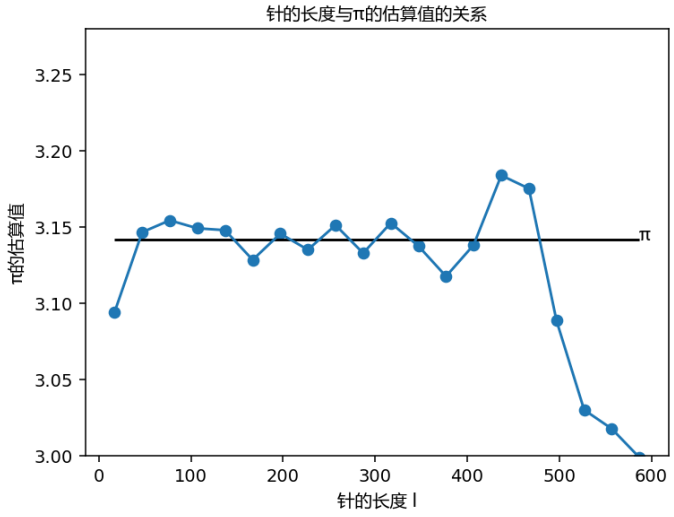

图 4

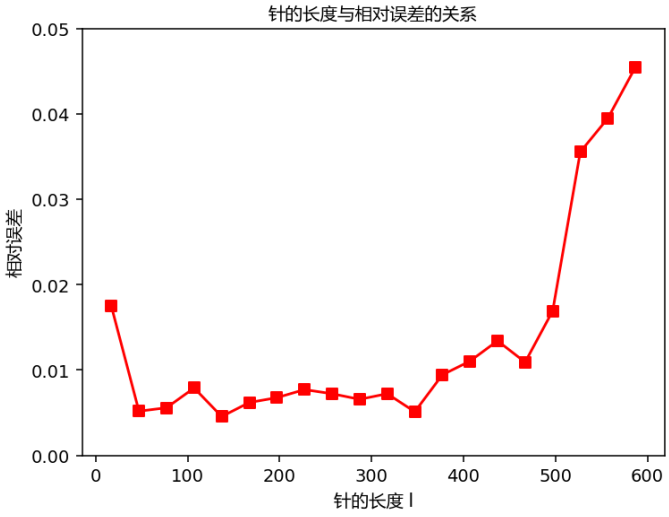

图 5

红色的误差曲线两边高，中间低，说明针的长度太长和太小都会使误差增大。针的长度太小使误差增大的原因，是由空间方格离散化导致的：

当“针眼”的位置固定之后，“针尖”所处的所有可能位置的集合构成了一个圆环。但是当我们把空间离散成方格之后，这个圆就不再是标准的圆了。（图 6，下图）

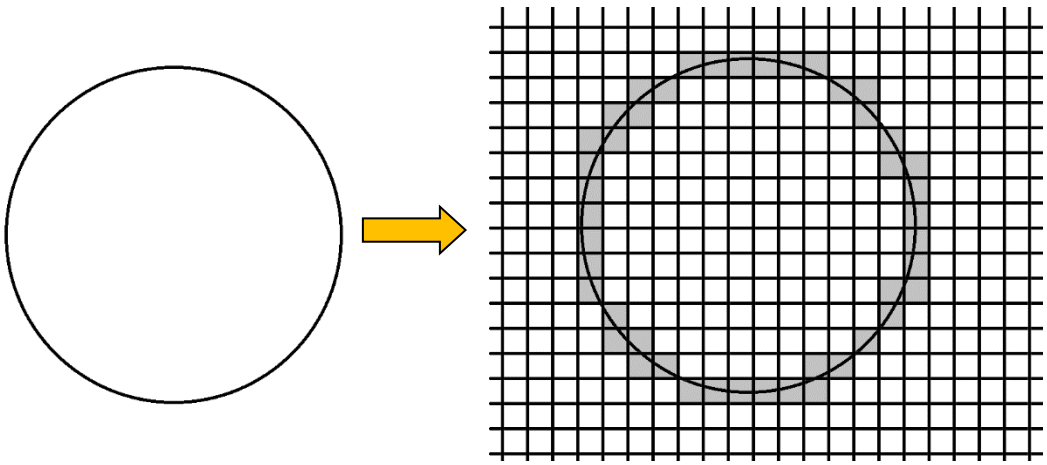

图 6

要想减小这种误差，就要提高空间离散的精度，扩大l和$size$，一方面，使得“针尖”的轨迹更接近光滑的圆；另一方面，使得平行线更加接近一维的没有宽度的线。

排除该误差，l不能取得太小，控制投针次数$N = 1000$和和纸张大小$size = 10000$不变。依次选择l在 300 到 5000 之间的 200 个值重复之前的步骤，做出以下两图（下页图 7，图 8）：

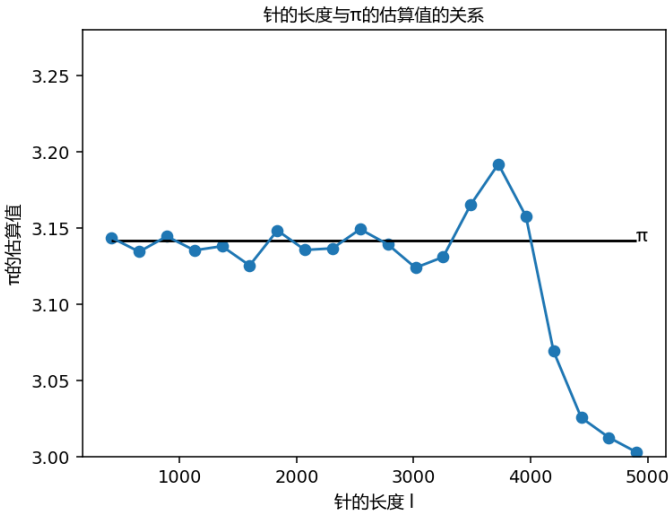

图 7

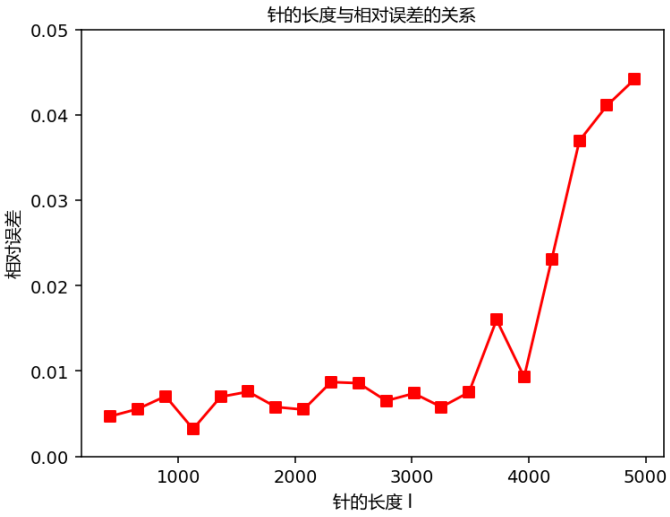

图 8

红色的误差曲线只有右边升高，说明针的长度过大会引起很大误差。这是因为当针的长度与纸的边长可比拟时，针落在纸上的各点后不再等价（下图，图 9）.

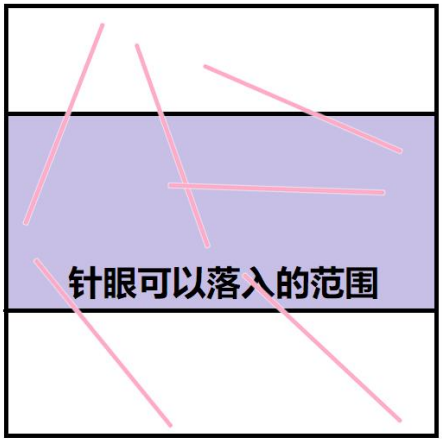

图 9

为了减小这种误差，要求$size \gg l$。于是控制投针次数$N = 1000$和和纸张大小$size = 10000$不变。依次选择l在 300 到 1000 之间的 200 个值重复之前的步骤，做出以下两图（图 10，图 11）：

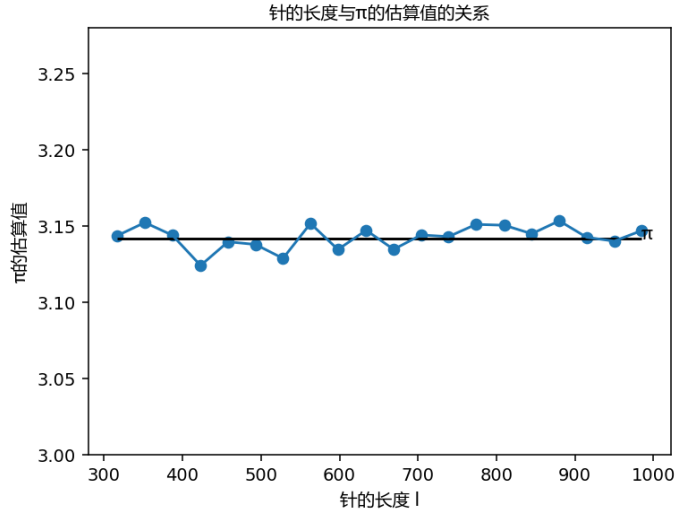

图 10

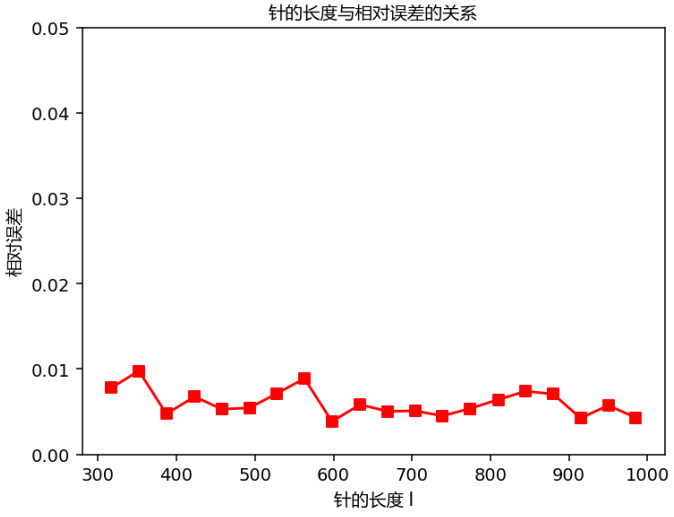

图 11

排除这个因素引起的误差后，由图 11 的红色曲线可以看到，误差已经在 1%以内。

3.2 纸张大小引起的误差

控制投针次数$N = 10000$和和针的长度$l = 300$不变。依次选择纸张大小$size$从 600 到 3000 之间的 200 个值做蒙特卡罗模拟，将每相邻的 10 个$size$值得到的结果取平均，与标准π值相比较，做出计算的π值与随纸的边长的变化图（图 12）。以标准π值为基准计算其相对误差，然后相邻的 10 个l值的误差取平均，做出相对误差随纸的边长变化图（图 13）。

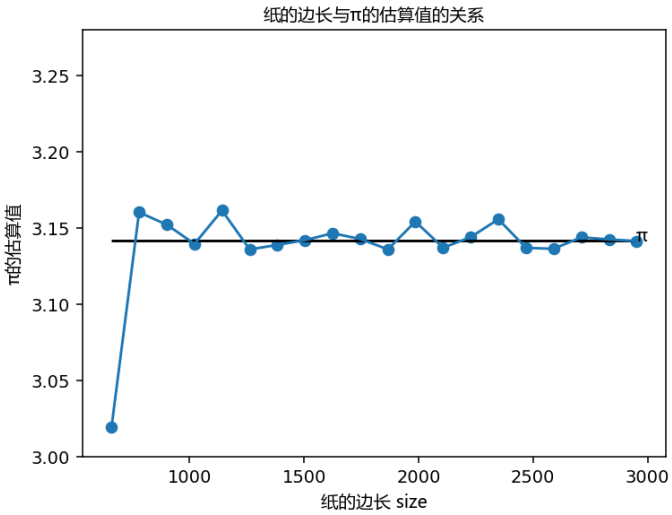

图 12

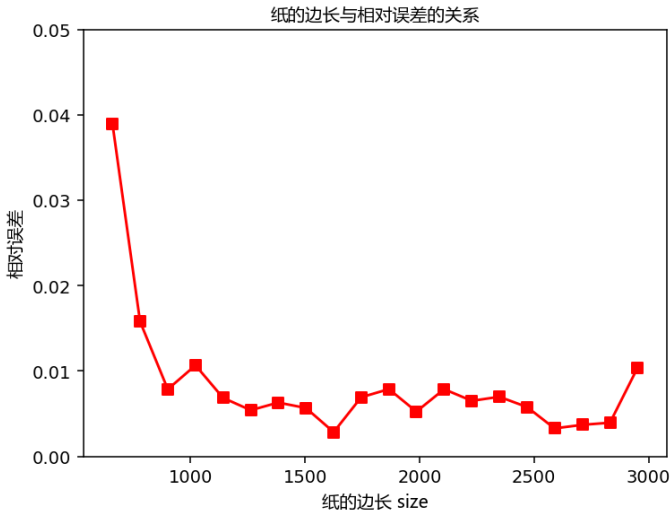

图 13

红色的误差曲线左边升高，其原因与针的长度过大所引起误差的原因相同，因为$size$过小的时候，纸张边长就与针的长度可比拟。

排除这个误差，控制投针次数$N = 10000$和和针的长度$l = 300$不变，选取$size$从 3000 到 100000 之间的 200 个值重复之前的步骤，做出以下两图（图 14，图 15）：

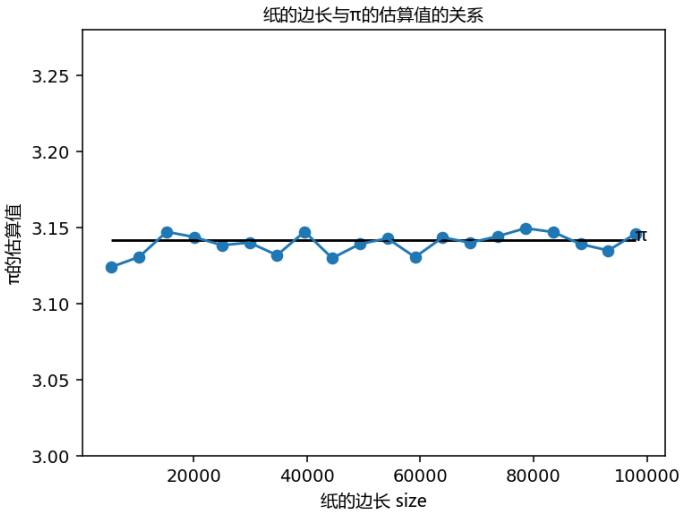

图 14

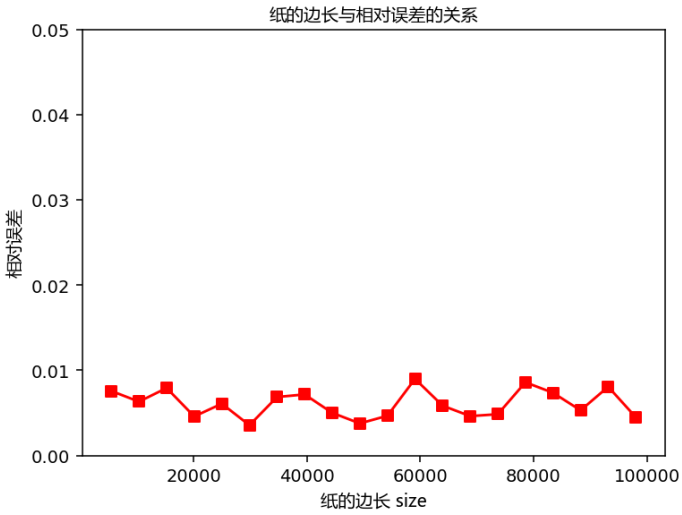

图 15

该误差排除后，由图 15 的红色曲线可以看到，误差已经在 1%以内。

值得注意的是，增加$size$会增加内存的占用，增加l会加重计算负担。在精度和效率之间找一个平衡，在 PC 上运行时推荐将$size$设为 1500，将l设为 300。本文第 2.2 节的结果正是在这种分析下选择的参数。

3.2 投针次数引起的误差

用随机过程估算一个定值，样本的大小直接影响误差，体现在本实验上就是投针的次数。直观来说，投针次数越多，误差越小。采用控制变量的思想，控制纸张大小$size = 1500$和针的长度$l = 300$不变，改变投针次数N。依次选择$N = 1000,10000,100000,1000000$做蒙特卡罗模拟，每个$N$值各重复实验 20 次。将得到的结果与标准π值相比较，画成散点图（图 16, 图 17, 图 18, 图 19）。

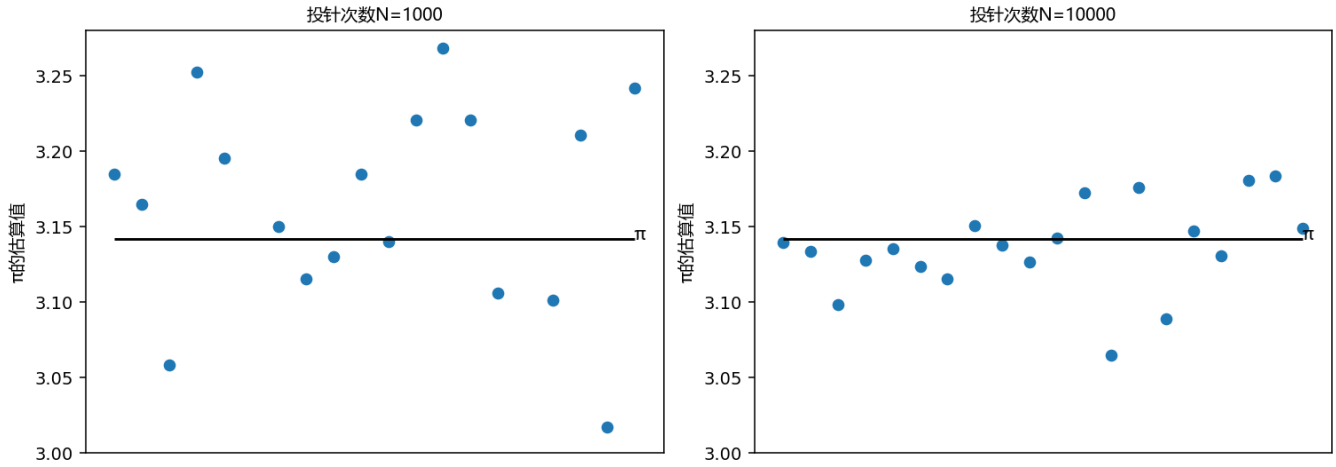

图 16

图 17

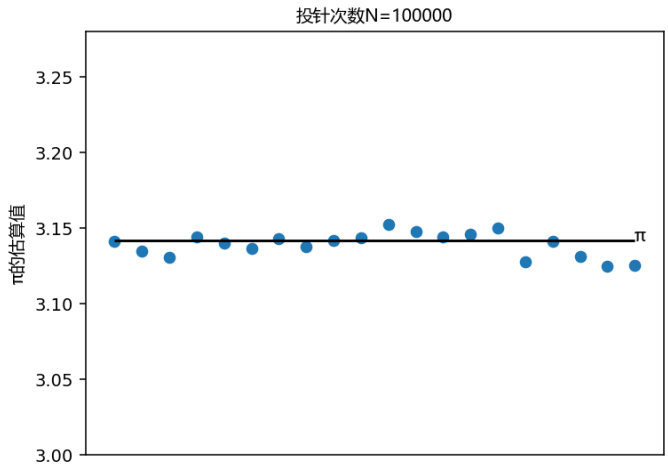

图 18

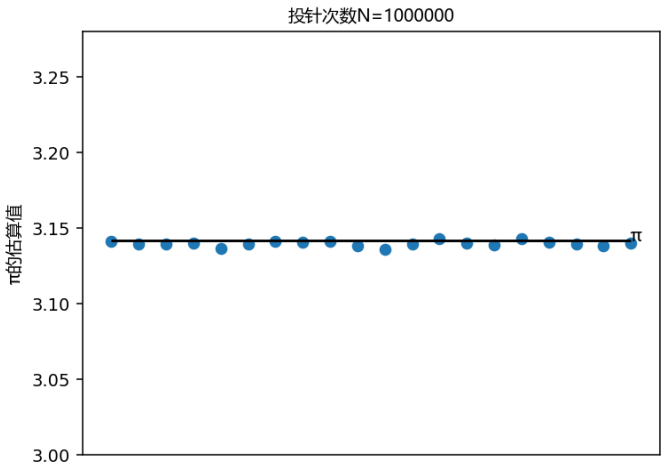

图 19

对于同一N值而言，各次实验的结果均匀地分布在精确值的两侧，说明投针的随机性引入的是随机误差。这种误差会随着N的增大而减小，表现在上方 4 图，就是当投针次数N越来越大时，同一N值重复实验的结果的方差越来越小，它们越来越靠近精确值所对应的水平线。N越大，这种误差越小。不过受 PC 计算能力的限制，N不能无限地增大。对 PC 的计算能力而言，$N = 1 \times 10^9$已是计算能力的极限。本文第 2.2 节的表 2 展示了计算时间随N的增加而增加的关系。

4. 蒙特卡罗方法模拟蒲丰投针问题

4.1 低差异序列

低差异序列(Low Discrepancy Sequence)是按某种固定的计算方法生成的数组序列[6]，对于一维情况，它就是一个数列。由于产生的方法固定，数列中的数字并不是随机数。通过特定的算法打乱数列中数字的排序，会得到一个新的数列，这是一种低差异序列。低差

异序列中的元素均匀分布在它们的定义空间。均匀而不随机，这是这些数字的特点。于是它们被称为“准随机数”(Quasi-random Number)。

所谓均匀与随机的区别，体现在以下两图的对比（图 20，图 21[①]）

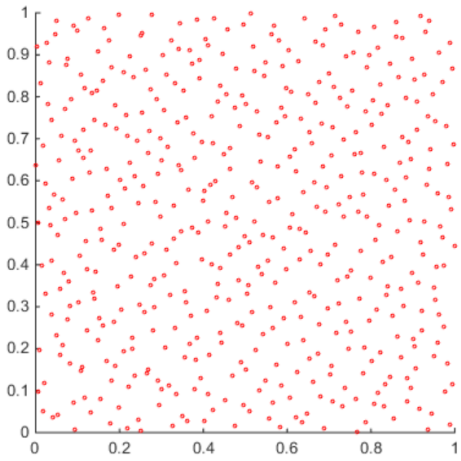

图 20 低差异序列散点

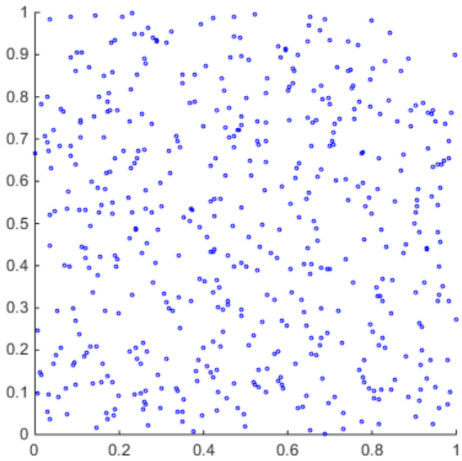

图 21 伪随机数散点

可见，低差异序列比伪随机数列更加均匀。但是，它不是随机的，无法通过随机数测试，因此只能被称为准随机数。

4.2 准蒙特卡罗方法

给定一个 Halton 序列[②]（低差异序列的一种），用 Python 的 Scipy 软件包读取其中的元素并生成一个 list。用其中的元素代替 random 软件包产生的随机数进行这个实验，就是用准蒙特卡罗方法来计算π值。

选择$l = 300, size = 10000$，投针$N = 5 \times 10^7$次，得到的π值为$3.108\cdots$，不如表 2 的蒙特卡洛方法的精度高。由于生成低差异序列需要巨大的内存，所以在 PC 上无法进行更多次的准蒙特卡罗模拟。

而选择$l = 300, size = 1 \times 10^9$，投针$N = 5 \times 10^6$次，得到的π值为$3.118\cdots$，不如表 2 的蒙特卡洛方法的精度高。增大$size$可以提高准蒙特卡罗方法的精度，因为低差异序列更均匀，对于空间大小更加敏感。更大的“纸张大小”会带来更好的结果，但是会占用极大的内存。

总得来说，蒙特卡罗方法比准蒙特卡罗方法更适合用来模拟 Buffon 投针实验，因为 Buffon 投针实验更看重随机性，而不是均匀性。但是准蒙特卡罗方法的均匀性使得它在其它领域有更好的应用，比如高维空间积分的计算。[6]

5. 结语

本文给出了两种方法，分别为蒙特卡罗方法和准蒙特卡罗方法。对于计算π值而言，前者的效果更好，所以重点分析了前者。

在完成本文的过程中，可以深刻地感觉到编写程序时最困难的部分是误差分析。但是误差的分析是非常重要的事情。对于系统误差，要分析程序本身的问题，尤其是边界与端点处的修正。对于随机过程本身的误差，可以通过提高实验次数来解决，相应的计算量会增大，

①图 20，图 21 摘录自 Matlab 帮助文档,2014 版。

②可以使用 Python 产生 Halton 序列，需要的软件包为 ghalton，但是该软件包需要 C++编译器。这里用 Matlab 代替它产生 Halton 序列，效果是一样的。ghalton 可以用 pip 安装。

这又对优化程序的计算和内存提出了挑战。

如今人们已经通过各种方法将π值计算得无比精确，可以得到上亿位有效数字。在 PC 上 Super Pi 软件可以轻易将π计算到一百万位有效数字，并且成为测试 CPU 性能和稳定性的工具。而本文的方法并不追求极致的精度，也不追求极致的效率，而是通过 Buffon 投针的模拟，揭示 Monte Carlo 方法的物理意义。

源代码在下一页的附录。附录中是打印版本，对于可运行的版本，请在 2018 年 1 月 1 日之后访问我的 github 链接，谢谢！

参考文献

［1］David Griffiths. Introduction to Quantum Mechanics[M]. 2nd ed. Upper Saddle River, NJ: Pearson Education, Inc, 2005. 21.

［2］David Griffiths. Instructor’s Solution Manual Introduction to Quantum Mechanics[M]. 2nd ed. Upper Saddle River, NJ: Pearson Education, Inc, 2005. 9-10.

［3］楚明娟. 基于蒙特卡洛序贯仿真的生产模拟算法与应用研究[D].合肥工业大学,2017.

［4］毕艳辉,程希明.用随机试验法计算欧拉常数 *e*[J].统计与管理,2016(01):9-10.

［5］汤韬,瞿洋.C++超长浮点类的设计及 π 的精确计算[J].空军工程大学学报(自然科学版),2001(05):39-41.

［6］知乎专栏. 低差异序列（一）常见序列的定义及性质[EB/OL]. https://zhuanlan.zhihu.com/p/20197323?columnSlug=graphics. 2015.

● 附录：源代码

以下为打印版本；对于可运行的版本，请于 2018 年 1 月 1 日之后访问我的 github 链接。

```
# -*- coding: utf-8 -*-
"""
Created by Li Dongxu (2015301510021) on Tue Dec 26 10:29:50 2017
Final paper : Evaluating Pi by random progress
"""

'''警告！在 PC 上运行本程序的全部内容可能需要 10 小时或者更长的时间，
因此将大部分调用函数进行计算并生成结果的命令写成了以#为开头的注释形式。
如果运行相应的部分并查看结果，请手动去掉#，就可以运行该命令。'''

'''以下两行代码用来解决中文字体在绘图时显示的问题'''
import matplotlib.font_manager as fm
zh=fm.FontProperties(fname='C:/Windows/Fonts/msyh.ttc')

'''如果你用的不是 Windows 平台，或者缺少 msyh.ttc 字体，请对上一行代码做相应的调整。'''

import random as rd
import math
import matplotlib.pyplot as plt
import numpy as np
import time
import scipy.io as sio
np.set_printoptions(threshold=np.inf)

#这个类完整地包含了进行 Buffon 投针的所有动作
class Buffon_MonteCarlo:

    #l 为针的长度，size 为白纸尺寸，N 为投针次数
    def __init__(self,l,size,N):
        self.size=size
        self.N=N
        self.l=l #针的长度
        self.counter=0  #相交次数
        if self.size<20:
            print('您选的纸的 size 太小了，不能做这个实验，建议将 size 设为 100 或更大')
        if self.size<2*self.l:
            print('针的长度必须比纸的边长的一半小')

    #画平行线
```

```
    def drawlines(self):
        lines=np.zeros((self.size,), dtype=np.int)
        i=int(self.l/2) #在针的长度的一半的位置画第一条线
        while i<=self.size-1:
            lines[i]=1
            i=i+self.l
        return lines

    #扔一次针
    def toss(self,lines):
        self.lines=lines
        #确定针眼的位置
        #x_0=rd.randint(self.l,self.size-self.l)  #这一步计算是不需要的
        y_0=rd.randint(self.l,self.size-self.l)

        #确定针尖的位置(本程序误差瓶颈)
        theta=rd.uniform(0,360)
        #x=round(x_0+self.l*math.cos(math.radians(theta))) #这一步计算是不需要的
        y=round(y_0+self.l*math.sin(math.radians(theta)))  #四舍五入取整

        #判断是否跨过平行线
        upper=max(y,y_0)
        lower=min(y,y_0)
        delta=0 #计数器的辅助变量

          #如果其中有跨过平行线的点，那么记一次相交
        _randomone=rd.uniform(0,1)
        if _randomone>0.6366197723675814:
            randomone=1
        else:
            randomone=0
        if upper!=lower:
            for k in range(lower,upper-randomone):   #减去其中一个端点的投影值
                delta=delta+lines[k]
            if delta>0.0001:
                self.counter=self.counter+1
          #如果针躺在平行线上也要记一次相交
        else:
            if lines[upper]-1<0.0001:
                self.counter=self.counter+1

    #开展一次实验，包含画线和投针
    def experiment(self):
```

```
        lines=self.drawlines()
        for i in range(self.N):
            self.toss(lines)
            #if i %1000000 ==0:
                #print(i/1000000)  #每投针 1e6 次输出一个数字来显示进度
        probability=self.counter/self.N  #相交概率
        return probability


#这个函数实例化上方定义的类，返回估算的 pi 值
def cal(l=300,size=1500,N=10000):
    Buffon=Buffon_MonteCarlo(l,size,N)
    p=Buffon.experiment()
    estimated_pi=2/p
    return estimated_pi


#这个函数用来执行投针次数非常多的模拟，输入你可以承受的运算时间（有 40,400,4000,40000
#秒四挡可以选择），函数会显示估算的 pi 值和运算实际花费的时间。
def cal_extreme(Time): #Time 是你可以承受的运算时间，单位：秒
    l=300
    size=1500

    if Time==40000:
        #N=1000000000  #1e9 是个人电脑的运算极限，将它拆分为 4x2.5e8 行并行计算(4 核心)
        N=250000000
    elif Time==4000:
        #N=100000000  #将 1e8 拆分为 4x2.5e7 并行计算(4 核心)
        N=25000000
    elif Time==400:
        N=10000000
    elif Time==40:
        N=1000000

    time_start=time.time()
    Buffon=Buffon_MonteCarlo(l,size,N)
    p=Buffon.experiment()
    estimated_pi=2/p
    time_end=time.time()

    T=time_end-time_start
    print('estimated_pi=',estimated_pi)
    print('computing time=',T)
```

```
    return estimated_pi


#这个函数用来探究针的长度对误差的影响
def err1(lengthmin,lengthmax,size):
    Lengths=np.linspace(lengthmin,lengthmax,num=200,dtype=int)
    Estimated_pi=[]
    Errors=[]
    for l in Lengths:
        l=int(l)  #将 np.float64 形式转化为 int 形式
        _pi=cal(l,size)
        Estimated_pi.append(_pi)
        _error=(abs(_pi-math.pi))/math.pi
        Errors.append(_error)

    #相邻的 10 个 l 值的结果取平均
    X=[]
    Y_pi=[]
    Y_error=[]
    for i in range(20):
        X.append(Lengths[10*i+5])
        sum_pi=0
        sum_error=0
        for k in range(10):
            sum_pi+=Estimated_pi[10*i+k]
            sum_error+=Errors[10*i+k]
        Y_pi.append(sum_pi/10)
        Y_error.append(sum_error/10)

    plt.figure(dpi=140,figsize=(6,4.5))
    plt.title('针的长度与π的估算值的关系',fontproperties=zh)
    plt.hlines(math.pi,min(X),max(X))
    plt.plot(X,Y_pi)
    plt.scatter(X,Y_pi)
    plt.ylim((3.00,3.28))
    plt.xlabel('针的长度 l',fontproperties=zh)
    plt.ylabel('π的估算值',fontproperties=zh)
    plt.text(max(X),math.pi,'π',fontproperties=zh)

    plt.figure(dpi=140,figsize=(6,4.5))
    plt.title('针的长度与相对误差的关系',fontproperties=zh)
    plt.plot(X,Y_error,c='r')
    plt.scatter(X,Y_error,c='r',marker='s')
```

```
    plt.ylim((0,0.05))
    plt.xlabel('针的长度 l',fontproperties=zh)
    plt.ylabel('相对误差',fontproperties=zh)


#这个函数用来探究纸张大小对误差的影响
def err2(sizemin,sizemax):
    Sizes=np.linspace(sizemin,sizemax,num=200,dtype=int)
    Estimated_pi=[]
    Errors=[]
    l=300
    for size in Sizes:
        size=int(size)  #将 np.float64 形式转化为 int 形式
        _pi=cal(l,size)
        Estimated_pi.append(_pi)
        _error=(abs(_pi-math.pi))/math.pi
        Errors.append(_error)

    #相邻的 10 个 l 值的结果取平均
    X=[]
    Y_pi=[]
    Y_error=[]
    for i in range(20):
        X.append(Sizes[10*i+5])
        sum_pi=0
        sum_error=0
        for k in range(10):
            sum_pi+=Estimated_pi[10*i+k]
            sum_error+=Errors[10*i+k]
        Y_pi.append(sum_pi/10)
        Y_error.append(sum_error/10)

    plt.figure(dpi=140,figsize=(6,4.5))
    plt.title('纸的边长与 π 的估算值的关系',fontproperties=zh)
    plt.hlines(math.pi,min(X),max(X))
    plt.plot(X,Y_pi)
    plt.scatter(X,Y_pi)
    plt.ylim((3.00,3.28))
    plt.xlabel('纸的边长 size',fontproperties=zh)
    plt.ylabel('π 的估算值',fontproperties=zh)
    plt.text(max(X),math.pi,'π',fontproperties=zh)

    plt.figure(dpi=140,figsize=(6,4.5))
```

```
    plt.title('纸的边长与相对误差的关系',fontproperties=zh)
    plt.plot(X,Y_error,c='r')
    plt.scatter(X,Y_error,c='r',marker='s')
    plt.ylim((0,0.05))
    plt.xlabel('纸的边长 size',fontproperties=zh)
    plt.ylabel('相对误差',fontproperties=zh)


#这个函数用来探究投针次数对误差的影响
def err3(Nmin,Nmax):
    Ns=np.linspace(Nmin,Nmax,num=200,dtype=int)
    Estimated_pi=[]
    Errors=[]
    l=300
    size=1500
    for N in Ns:
        N=int(N)  #将 np.float64 形式转化为 int 形式
        _pi=cal(l,size,N)
        Estimated_pi.append(_pi)
        _error=(abs(_pi-math.pi))/math.pi #注意这里取了绝对值
        Errors.append(_error)

    #相邻的 10 个 l 值的结果取平均
    X=[]
    Y_pi=[]
    Y_error=[]
    for i in range(20):
        X.append(Ns[10*i+5])
        sum_pi=0
        sum_error=0
        for k in range(10):
            sum_pi+=Estimated_pi[10*i+k]
            sum_error+=Errors[10*i+k]
        Y_pi.append(sum_pi/10)
        Y_error.append(sum_error/10)

    plt.figure(dpi=140,figsize=(6,4.5))
    plt.title('投针次数与π的估算值的关系',fontproperties=zh)
    plt.hlines(math.pi,min(X),max(X))
    plt.plot(X,Y_pi)
    plt.scatter(X,Y_pi)
    plt.ylim((3.00,3.28))
    plt.xlabel('投针次数 N',fontproperties=zh)
```

```
    plt.ylabel('π 的估算值',fontproperties=zh)
    plt.text(max(X),math.pi,'π',fontproperties=zh)

    plt.figure(dpi=140,figsize=(6,4.5))
    plt.title('投针次数与相对误差的关系',fontproperties=zh)
    plt.plot(X,Y_error,c='r')
    plt.scatter(X,Y_error,c='r',marker='s')
    plt.ylim((0,0.05))
    plt.xlabel('投针次数 N',fontproperties=zh)
    plt.ylabel('相对误差',fontproperties=zh)


#这个函数用来绘制固定 N 并重复 20 次实验的结果散点分布图
def err4(fixed_N):
    X=[]
    Y=[]
    for k in range(20):
        _y=cal(300,1500,fixed_N)
        X.append(k)
        Y.append(_y)

    plt.figure(dpi=140,figsize=(6,4.5))
    plt.title('投针次数 N=%d'%fixed_N,fontproperties=zh)
    plt.hlines(math.pi,min(X),max(X))
    plt.scatter(X,Y)
    plt.ylim((3.00,3.28))
    plt.ylabel('π 的估算值',fontproperties=zh)
    plt.text(max(X),math.pi,'π',fontproperties=zh)
    #去掉 x 轴，因为 x 轴在本图中没有意义
    frame=plt.gca()
    frame.axes.get_xaxis().set_visible(False)


#以下代码为准蒙特卡罗方法
class Buffon_QuasiMonteCarlo(Buffon_MonteCarlo):

    #准随机数列长度只有 1e8，所以 N 不能大于 5e7 次。
    def quasi_random(self):
        path='C:\\Users\\LDX\\Documents\\真正的文档\\学习\\物\
理\\计算物理\\FINAL\\halton\\halton.mat'

        '''这个文件的大小接近 200mb，它所提供的准随机数列可以允许的最大
        投针次数为 5e6 次。如果要做另一次准蒙特卡罗模拟，需要重新用 MATLAB
```

```
298.         生成一个新的数组，否则只会得出相同的结果'''
299.
300.         matfile=sio.loadmat(path) #读取 matlab 生成的 mat 文件
301.         halton=matfile['X0']
302.         Halton=[] #在 python 中产生准随机数的 list
303.         for k in range(2*self.N): #投 N 次针共需要 2N 个准随机数
304.             _halton=float(halton[k])
305.             Halton.append(_halton)
306.         return Halton
307.
308.     def toss(self,lines,index):
309.         self.lines=lines
310.         #确定针眼的位置
311.         #x_0=rd.randint(self.l,self.size-self.l)  #这一步计算是不需要的
312.         y_0=int(round(abs(self.l-(self.size-self.l)))*self.Halton[index]+self.l)
313.
314.         #确定针尖的位置(本程序误差瓶颈)
315.         theta=int(round(360*self.Halton[-index]))
316.         #x=round(x_0+self.l*math.cos(math.radians(theta))) #这一步计算是不需要的
317.         y=int(round(y_0+self.l*math.sin(math.radians(theta))))  #四舍五入取整
318.
319.         #判断是否跨过平行线
320.         upper=max(y,y_0)
321.         lower=min(y,y_0)
322.         delta=0 #计数器的辅助变量
323.
324.           #如果其中有跨过平行线的点，那么记一次相交
325.         _randomone=rd.uniform(0,1)
326.         if _randomone>0.6366197723675814:
327.             randomone=1
328.         else:
329.             randomone=0
330.
331.         if upper!=lower:
332.             for k in range(lower,upper-randomone):
333.                 delta=delta+lines[k]
334.             if delta>0.0001:
335.                 self.counter=self.counter+1
336.           #如果针躺在平行线上也要记一次相交
337.         else:
338.             if lines[upper]-1<0.0001:
339.                 self.counter=self.counter+1
340.
```

```
341.     def experiment(self):
342.         lines=self.drawlines()
343.         for i in range(self.N):
344.             self.toss(lines,index=i)
345.             if i %1000000 ==0:
346.                 print(i/1000000)
347.         probability=self.counter/self.N  #相交概率
348.         return probability
349.
350.     def estimate(self):
351.         '''''这里把 Halton 作为类属性而不是函数里的一个局域变量，是因为
352.         (1)它是常量，不随计算改变
353.         (2)如果不将它作为类属性，每次循环都要重新读取 mat 文件，会将程序整体的速度
354.         减小到现有程序速度的千分之一一下'''
355.
356.         self.Halton=self.quasi_random()
357.         p=self.experiment()
358.         #print('probability=',p)
359.         print('estimated Pi=',2/p)
360.
361.#用准蒙特卡罗方法计算
362.def cal_quasi_montecarlo(size=1000000000,N=5000000):
363.     Buffon2=Buffon_QuasiMonteCarlo(300,size,N)
364.     Buffon2.quasi_random()
365.     Buffon2.estimate()
366.
367.#以下代码用于生成论文中的算法介绍的示意图，不返回计算结果
368.class my_demo(Buffon_MonteCarlo):
369.     def drawlines1(self):
370.         lines=[]
371.         i=int(self.l/2) #在针的长度的一半的位置画第一条线
372.         while i<=self.size-1:
373.             _lines=i
374.             lines.append(_lines)
375.             i=i+self.l
376.         return lines
377.
378.     def plotlines(self):
379.         y=self.drawlines1()
380.         print(y)
381.         plt.figure(dpi=140,figsize=(6,4.5))
382.         plt.title('在白纸上画一些平行线',fontproperties=zh)
383.         plt.hlines(y,0,self.size)
```

```
384.         plt.xlim((0,100))
385.         plt.ylim((0,100))
386.
387.     #扔一次针
388.     def toss(self):
389.         #确定针眼的位置
390.         x_0=round(rd.uniform(self.l,self.size-self.l))
391.         y_0=rd.randint(self.l,self.size-self.l)
392.
393.         #确定针尖的位置(本程序误差瓶颈)
394.         theta=rd.uniform(0,360)
395.         x=round(x_0+self.l*math.cos(math.radians(theta)))
396.         y=round(y_0+self.l*math.sin(math.radians(theta)))  #四舍五入取整
397.         return x_0,y_0,x,y
398.
399.     #画出投针示意图
400.     def plotneedles(self):
401.         X=[0,0]
402.         Y=[0,0]
403.         fig=plt.figure(dpi=140,figsize=(6,4.5))
404.         plt.title('投针（50根）',fontproperties=zh)
405.         plt.xlim((0,100))
406.         plt.ylim((0,100))
407.         for i in range(self.N):
408.             x_0,y_0,x,y=self.toss()
409.             X[0]=x_0
410.             X[1]=x
411.             Y[0]=y_0
412.             Y[1]=y
413.             plt.plot(X,Y)
414.         plt.hlines(self.drawlines1(),0,self.size)
415.
416.
417.'''以下为计算命令，请根据需要去掉前方的#来执行运算'''
418.#err1(2,600,1200)          #两端误差都大
419.#err1(300,5000,10000)     #左端误差小，右端误差大
420.#err1(300,1000,10000)     #两端误差都很小
421.
422.#err2(600,3000)            #左边误差大
423.#err2(3000,100000)         #两端误差都很小
424.
425.#err3(100,2000)
426.#err3(2000,100000)
```

```
427.
428.#err4(1000)
429.#err4(10000)
430.#err4(100000)
431.#err4(1000000)
432.
433.cal_extreme(40)
434.#cal_extreme(400)
435.#cal_extreme(4000)  #需要并行计算，用 4 个内核分别调用此函数后求平均
436.#cal_extreme(40000) #需要并行计算，用 4 个内核分别调用此函数后求平均
437.
438.#cal_quasi_montecarlo(10000,50000000)
439.#cal_quasi_montecarlo(1000000000,5000000)
440.
441.#demo=my_demo(10,100,50)
442.#demo.plotlines()
443.#demo.plotneedles()
```