# 数值分析第一章补充阅读

朱宇嘉

## 1　什么是数值分析：以圆周率 $\pi$ 为例

让我们先来考虑一个简单的问题：我们是如何计算圆周率 $\pi$ 的？

我们都知道圆周率 $\pi$ 表示的是一个圆的面积与其半径的平方的比值，即 $\pi = \frac{S}{r^2}$。此外，圆周率的值是唯一确定的，但是怎样才能够**又快又好**的将它计算出来呢？

### 1.1　方法 1：直接测量法

直接测量出圆的面积 $S$ 和圆的半径 $r$，然后按照公式的形式做一次除法就结束了。

- 优点：直接，明了。
- 缺点：圆的面积测量起来不是那么容易，往往需要制作极其精密的仪器才可以缩小测量误差。并且对于同一测量仪器，我们没有办法减小误差。这样计算出来的 $\pi$ 在日后计算圆的面积时误差会相当之大。

所以，我们希望找到一个方法，他可以**自发地**（如增加迭代次数）减小 $\pi$ 的计算误差，甚至我们希望我们可以在理论上（数学上）证明我们得出的结果可以无限接近 $\pi$ 的真实值。

### 1.2　方法 2：投点模拟法

让我们回到计算统计的课堂，我们可以用以下的方法计算出 $\pi$：

- 不断随机生成两个随机变量 $X \sim U(-1,1), Y \sim U(-1,1)$
- 不断记录并更新 $X^2 + Y^2 \leq 1$ 的比例，并将其乘以 4 得到 $\pi$

这个简单的方法就可以通过增加实验次数的方式**自发地**减小 $\pi$ 的计算误差（感谢大数定律提供的理论保障）。我们也可以写一个简单的程序实现一下，来计算一下 $\pi$。

```
set.seed(2021)
sim_time<-1000000
X<-runif(sim_time,-1,1);Y<-runif(sim_time,-1,1)
```

```r
pi_est<-sum(X^2+Y^2<=1)/sim_time*4
print(paste("Estimated value is",pi_est))
```

```
## [1] "Estimated value is 3.140308"
```

我们经过 1000000 次模拟，便得出了使用投点模拟法的 $\pi$ 的估计值 3.140308。以上所做的所有步骤便是数值分析的一个完整过程：

- 实际问题：如何计算圆周率 $\pi$。
- 数学模型：实际在这里是一个统计模型，投点模拟模型（当且这样称呼）。
- 计算方法：模拟。
- 程序设计：5 行 R 语言代码。
- 得出结果：$\pi$ 的估计值为 3.140308。

但是我们还发现一个问题：虽然我们已经模拟了 1000000 次，但是计算的结果却不是那么准确。因为我们在小学时就知道 $\pi$ 的小数点后 7 位是 3.1415926，而我们这里只算对了前两位。

如果再写一段程序我们会发现；其实用这个方法计算 $\pi$ 要得到精确的估计值是很慢的。

```r
set.seed(2021)
sim_time<-100000000
X<-runif(sim_time,-1,1);Y<-runif(sim_time,-1,1)
out<-cumsum(X^2+Y^2<=1)/c(1:sim_time)*4
plot(out[5e5:sim_time],type='l',xlim=c(0,sim_time-5e5),
     xlab="simulation time",ylab="estimated value")
abline(h=pi,lwd=4,col="red")
```

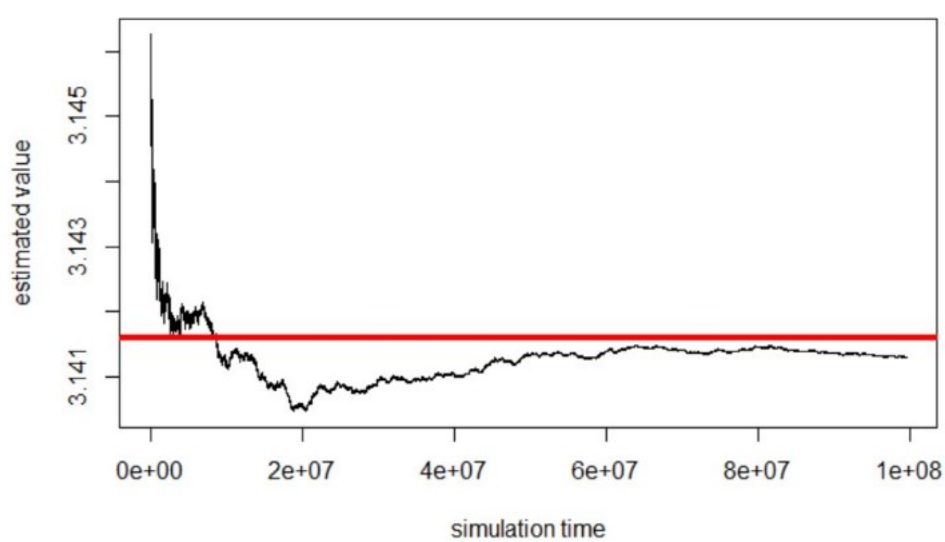


图 1: 估计值随模拟次数变化图（从第 500000 次开始）

我们做了 100000000 次模拟，发现了这个方法的缺点：**收敛慢!!!**。收敛慢的直接缺点就是：如果你要获得一个较为准确的结果，你需要做非常多的运算。大家可以自己用 R 语言跑一下上面的这段程序，看看要花多少时间。所以我们需要对这个方法做修改（请自行使用计算统计课堂中的方法，因为我懒得去翻书了）或者使用别的方法！

在进入到下一个方法之前，我们先要看三条计算机的公理：

- **公理 1**：计算机在绝大多数情况下计算的比人快。
- **公理 2**：计算机只能进行有限次运算。
- **公理 3**：计算机进行任何操作都需要一定的时间和空间。

公理 1 决定了大多数数值分析问题都是由计算机求解的。公理 2 激励我们要寻找更好、更稳定的算法，让算法用有限次（当然越少越好）的运算就可以逼近甚至是达到无限次的效果。公理 3 则表示我们在设计算法时还需要优化算法的内部结构和操作方法（可参考秦九韶算法），让算法运行地更快，占内存更少。（当然，一般情况下时间复杂度和空间复杂度不能一同减小。比如我在之前为了逃避 R 语言的缓慢的 for 循环，需要每次直接存储两个长度为 100000000 的数组，占内存共 **1.5G**）。

所以这几条公理也给了数值算法几条好坏评判标准：

- 1、收敛性：该算法是否可以收敛到真实值。
- 2、收敛速度：该算法收敛到真实值的快慢。
- 3、时间复杂度：部分算法虽然只需 5 次迭代即可收敛，但每次迭代所需的时间复杂度相当之高。
- 4、空间复杂度：同第 3 条。
- 5、稳定性：对于部分迭代类的算法，其收敛是否会受到初值的强烈影响。

## 1.3 方法 3：使用级数计算 $\pi$

对于阿基米德的圆周长近似方法和刘徽的面积近似方法（割圆术）就不讨论了，下面着重讲几个用级数计算 $\pi$ 的方法。当然，我们要从里面看出，什么算法是好的算法。

### 1.3.1 莱布尼茨公式

对于熟悉数学分析的同学，你会知道将反正切函数 $\arctan x$ 泰勒展开会变成以下这个形式：

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - ... = \sum_{i=1}^{\infty} \frac{(-1)^{i-1} x^{2i-1}}{2i-1}$$

将 $x = 1$ 代入，便可得到 $\frac{\pi}{4} = \sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{2k-1}$，即 $\pi = 4\sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{2k-1}$。

我们也可以写以下程序来看该级数的收敛速度：

```
Lebniz<-c(1:10000)
Lebniz<-(-1)^(Lebniz-1)/(2*Lebniz-1)
Lebniz<-4*cumsum(Lebniz)
for(i in 1:10)
  print(paste("After",i*1000,"iteration,estimated value is",
              round(Lebniz[i*1000],7)))
```

```
## [1] "After 1000 iteration,estimated value is 3.1405927"
## [1] "After 2000 iteration,estimated value is 3.1410927"
## [1] "After 3000 iteration,estimated value is 3.1412593"
## [1] "After 4000 iteration,estimated value is 3.1413427"
## [1] "After 5000 iteration,estimated value is 3.1413927"
## [1] "After 6000 iteration,estimated value is 3.141426"
## [1] "After 7000 iteration,estimated value is 3.1414498"
## [1] "After 8000 iteration,estimated value is 3.1414677"
## [1] "After 9000 iteration,estimated value is 3.1414815"
## [1] "After 10000 iteration,estimated value is 3.1414927"
```

经过 10000 次迭代，可以算出小数点后的 3 位有效数字，比起之前的模拟方法收敛速度有所提高，但我还是觉得太慢了。

### 1.3.2 梅钦公式

1706 年，英国数学家梅钦发现 $\frac{\pi}{4}=4\arctan\frac{1}{5}-\arctan\frac{1}{239}$，再将这两项分别泰勒展开，可以得到：

$$\pi=4[4\sum_{k=1}^{\infty}\frac{(-1)^{k-1}}{(2k-1)5^{2k-1}}-\sum_{k=1}^{\infty}\frac{(-1)^{k-1}}{(2k-1)239^{2k-1}}]$$

同样，我们也可以通过程序来看其收敛情况：

```
Machin<-c(1:10)
Machin<-4*(-1)^(Machin-1)*
  (4/((2*Machin-1)*5^(2*Machin-1))-
     1/((2*Machin-1)*239^(2*Machin-1)))
Machin<-cumsum(Machin)
for(i in 1:10)
  print(paste("After",i,"iteration,estimated value is",
              Machin[i]))
```

```
## [1] "After 1 iteration,estimated value is 3.18326359832636"
## [1] "After 2 iteration,estimated value is 3.14059702932606"
```

```
## [1] "After 3 iteration,estimated value is 3.14162102932503"
## [1] "After 4 iteration,estimated value is 3.14159177218218"
## [1] "After 5 iteration,estimated value is 3.1415926824044"
## [1] "After 6 iteration,estimated value is 3.14159265261531"
## [1] "After 7 iteration,estimated value is 3.14159265362355"
## [1] "After 8 iteration,estimated value is 3.1415926535886"
## [1] "After 9 iteration,estimated value is 3.14159265358984"
## [1] "After 10 iteration,estimated value is 3.14159265358979"
```

哇！4 次迭代算出 5 位小数点，5 次迭代算出 7 位小数点，10 次迭代算出 14 位小数点！更难得的是，在每一次的迭代中，梅钦公式的计算量只是莱布尼茨公式的 2 倍（如果认为一次乘法不影响效率的话）。

### 1.3.3 更快的算法

坐稳了，要发车了，接下来的两个算法相当恐怖。

拉马努金恐怖公式：

$$\frac{1}{\pi}=\frac{2\sqrt{2}}{99^2}\sum_{k=0}^{\infty}\frac{(4k)!}{(k!)^4}\frac{1103+26390k}{396^{4k}}$$

Chudnovsky 兄弟加强恐怖公式：

$$\frac{1}{\pi}=\frac{1}{53360\sqrt{640320}}\sum_{k=0}^{\infty}(-1)^k\frac{(6k)!}{(k!)^3(3k)!}\frac{13591409+545140134k}{640320^{3k}}$$

这两个公式算的有多快呢？我们来实验一下：

```
Ramanujan<-function(x){
  return(2*sqrt(2)/99^2*
           factorial(4*x)/factorial(x)^4*
           (1103+26390*x)/396^(4*x))
}
Chudnovsky<-function(x){
  return(1/(53360*sqrt(640320))*(-1)^x*
           factorial(6*x)/factorial(x)^3/factorial(3*x)*
           (13591409+545140134*x)/640320^(3*x))
}
PrintOutput<-function(){
  print("Ramanujan equation:")
  print(paste("After 1 iteration, estimated value is",
              1/Ramanujan(0)))
  print(paste("After 2 iteration, estimated value is",
```

```
              1/(Ramanujan(0)+Ramanujan(1))))
  print("Chudnovsky equation:")
  print(paste("After 1 iteration, estimated value is",
              1/Chudnovsky(0)))
}
PrintOutput()
```

```
## [1] "Ramanujan equation:"
## [1] "After 1 iteration, estimated value is 3.14159273001331"
## [1] "After 2 iteration, estimated value is 3.14159265358979"
## [1] "Chudnovsky equation:"
## [1] "After 1 iteration, estimated value is 3.14159265358973"
```

好吧！又快有准！yyds！事实上，拉马努金公式每算一项可以得到 8 个有效数字，Chudnovsky 兄弟公式每算一项可以得到 14 个有效数字。而之前的莱布尼茨公式和梅钦公式则分别是 0.02 个和 1.25 个。

甚至，还存在一种 Borwein 算法，在经过 7 次迭代后，就可以算出 15 万位有效数字。

但是，在实际的电脑默认程序中是用哪个方法算的呢？

答案是梅钦公式。因为后面那两个公式算每一项的时间复杂度过高，这也和我之前给出的判别准则的第 4 条相对应。而且一般的计算机的计算位数有限，用后面两个公式会导致位数消失（学名：浮点数下溢）的问题。

好的，说了那么多关于 $\pi$ 的计算方法，相信你对于什么是数值分析和什么是一个好的数值分析算法已经有了一定的认识。当然也有可能，你现在越来越糊涂了。

# 2 数值分析的两条主线

数值分析这门课这个学期的教学内容主要是有两条主线：逼近和计算。

数学中最重要的元素之二是：函数和方程。

但是，很多情况下的现实是：函数太复杂，方程太难解。

而数值分析就是要让函数不那么复杂，方程不那么难解。让函数不那么复杂的方法就是用简单的函数去逼近复杂的函数，而让方程不那么难解的方法就是开发可靠快速的求解方法。

因此，本学期的课程的思路主要是（学期末来看可能效果更佳）：

- 第二章：插值。讲述怎么样用一个简单函数*在观测点上*去**完美**逼近真实函数，让简单函数也拥有真实函数所拥有的性质（如取值，导数值）。
- 第三章：函数逼近。讲述怎么样用一个简单函数*在定义域上*去逼近复杂函数。

- 第四章：数值积分与微分。讲述怎么样用简单函数的积分或微分值去逼近复杂的积分或微分值。

——————————此处是两块内容的分割线——————————

- 第五章、第六章：解线性方程组。用两种不同的方法讲述怎么样又快又好计算线性方程组的解。
- 第七章：解非线性方程（组）。讲述怎么样又快又好计算非线性方程（组）的解。

## 3 参考

1. Approximations of pi. Wikiwand.

2. Borwein's algorithm. Wikiwand.