解析 Monte-Carlo 算法(基本原理,理论基础,应用实践)

引言

最近在和同学讨论研究 Six Sigma（六西格玛）软件开发方法及 CMMI 相关问题时，遇到了需要使用 Monte-Carlo 算法模拟分布未知的多元一次概率密度分布问题。于是花了几天时间，通过查询相关文献资料，深入研究了一下 Monte-Carlo 算法，并以实际应用为背景进行了一些实验。

在研究和实验过程中，发现 Monte-Carlo 算法是一个非常有用的算法，在许多实际问题中，都有用武之地。目前，这个算法已经在金融学、经济学、工程学、物理学、计算科学及计算机科学等多个领域广泛应用。而且这个算法本身并不复杂，只要掌握概率论及数理统计的基本知识，就可以学会并加以应用。由于这种算法与传统的确定性算法在解决问题的思路方面截然不同，作为计算机科学与技术相关人员以及程序员，掌握此算法，可以开阔思维，为解决问题增加一条新的思路。

基于以上原因，我有了写这篇文章的打算，一是回顾总结这几天的研究和实验，加深印象，二是和朋友们分享此算法以及我的一些经验。

这篇文章将首先从直观的角度，介绍 Monte-Carlo 算法，然后介绍算法基本原理及数理基础，最后将会和大家分享几个基于 Monte-Carlo 方法的有意思的实验。所以程序将使用 C#实现。

阅读本文需要有一些概率论、数理统计、微积分和计算复杂性的基本知识，不过不用太担心，我将尽量避免过多的数学描述，并在适当的地方对于用到的数学知识进行简要的说明。

Monte-Carlo 算法引导

首先，我们来看一个有意思的问题：在一个 1 平方米的正方形木板上，随意画一个圈，求这个圈的面积。

我们知道，如果圆圈是标准的，我们可以通过测量半径 r，然后用 S = pi * r^2 来求出面积。可是，我们画的圈一般是不标准的，有时还特别不规则，如下图是我画的巨难看的圆圈。

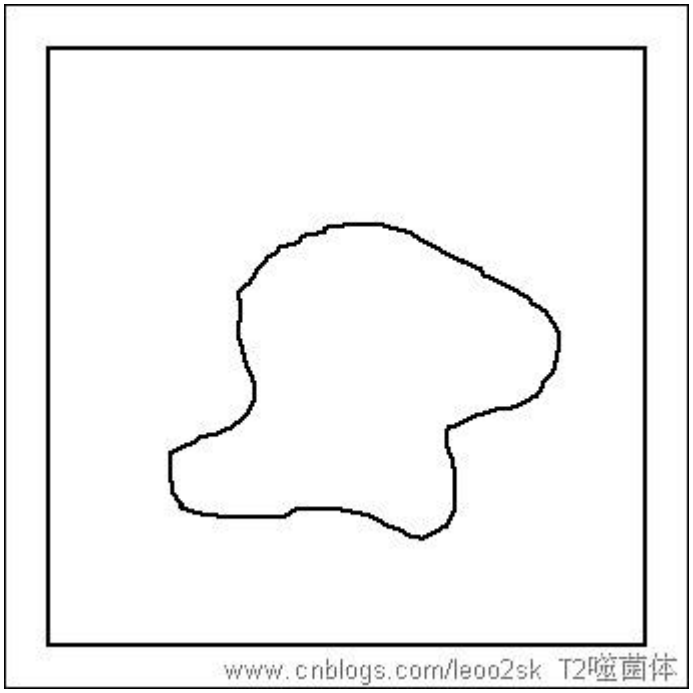

图 1、不规则圆圈

显然，这个图形不太可能有面积公式可以套用，也不太可能用解析的方法给出准确解。不过，我们可以用如下方法求这个图形的面积：

假设我手里有一支飞镖，我将飞镖掷向木板。并且，我们假定每一次都能掷在木板上，不会偏出木板，但每一次掷在木板的什么地方，是完全随机的。即，每一次掷飞镖，飞镖扎进木板的任何一点的概率的相等的。这样，我们投掷多次，例如 100 次，然后我们统计这 100 次中，扎入不规则图形内部的次数，假设为 k，那么，我们就可以用 k/100 * 1 近似估计不规则图形的面积，例如 100 次有 32 次掷入图形内，我们就可以估计图形的面积为 0.32 平方米。

以上这个过程，就是 Monte-Carlo 算法直观应用例子。

非形式化地说，Monte-Carlo 算法泛指一类算法。在这些算法中，要求解的问题是某随机事件的概率或某随机变量的期望。这时，通过“实验”方法，用频率代替概率或得到随机变量的某些数字特征，以此作为问题的解。

上述问题中，如果将“投掷一次飞镖并掷入不规则图形内部”作为事件，那么图形的面积在数学上等价于这个事件发生的概率（稍后证明），为了估计这个概率，我们用多次重复实验的方法，得到事件发生的频率 k/100 ，以此频率估计概率，从而得到问题的解。

从上述可以看出，Monte-Carlo 算法区别于确定性算法，它的解不一定是准确或正确的，其准确或正确性依赖于概率和统计，但在某些问题上，当重复实验次数足够大时，可以

从很大概率上（这个概率是可以在数学上证明的，但依赖于具体问题）确保解的准确或正确性，所以，我们可以根据具体的概率分析，设定实验的次数，从而将误差或错误率降到一个可容忍的程度。

上述问题中，设总面积为 S，不规则图形面积为 s，共投掷 n 次，其中掷在不规则图形内部的次数为 k。根据伯努利大数定理，当试验次数增多时，k/n 依概率收敛于事件的概率 s/S。下面给出严格证明：

证明：
不失一般性，设容器面积为S，不规则图形面积为S'
设事件A：投掷一次，并掷在不规则图形内。
因为投掷点服从二维均匀分布，所以$p(A) = S'/S$
设k是n次投掷中，掷在不规则图形内的次数，$\varepsilon > 0$为任意正数
根据伯努利大数定律：

$$\lim_{n\to\infty} p\left\{\left|\frac{k}{n} - p(A)\right| < \varepsilon\right\} = \lim_{n\to\infty} p\left\{\left|\frac{k}{n} - \frac{S'}{S}\right| < \varepsilon\right\} = 1$$

这就证明了，当n趋向于无穷大时，频率k/n依概率收敛于S'/S
证毕

上述证明从数学上说明用频率估计不规则图形面积的合理性，进一步可以给出误差分析，从而选择合适的实验次数 n，以将误差控制在可以容忍的范围内，此处从略。

从上面的分析可以看出，Monte-Carlo 算法虽然不能保证解一定是准确和正确，但并不是“撞大运”，其正确性和准确性依赖概率论，有严格的数学基础，并且通过数学分析手段对实验加以控制，可以将误差和错误率降至可容忍范围。

Monte-Carlo 算法的数理基础

这一节讨论 Monte-Carlo 算法的数理基础。

首先给出三个定义：优势，一致，偏真。这三个定义在后面会经常用到。

1）设 p 为一个实数，且 0.5<p<1。如果一个 Monte-Carlo 方法对问题任一实例的得到正确解的概率不小于 p，则该算法是 p 正确的，且 p-0.5 叫做此算法的优势。

2）如果对于同一实例，某 Monte-Carlo 算法不会给出不同的解，则认为该算法时一致的。

3）如果某个解判定问题的 Monte-Carlo 算法，当返回 true 时是一定正确的。则这个算法时偏真的。注意，这里没有定义“偏假”，因为“偏假”和偏真是等价的。因为只要互

换算法返回的 true 和 false，“偏假”就变成偏真了。

下面，我们讨论 Monte-Carlo 算法的可靠性和误差分析。

总体来说，适用于 Monte-Carlo 算法的问题，比较常见的有两类。一类是问题的解等价于某事件概率，如上述求不规则图形面积的问题；另一类是判定问题，即判定某个命题是否为真，如主元素存在性判定和素数测试问题。

先来分析第一类。对于这类问题，通常的方法是通过大量重复性实验，用事件发生的频率估计概率。之所以能这样做的数学基础，是伯努利大数法则：事件发生的频率依概率收敛于事件的概率 p。这个法则从数学生严格描述了频率的稳定性，直观意义就是当实验次数很大时，频率与概率偏差很大的概率非常小。此类问题的误差分析比较繁杂，此处从略。有兴趣的朋友可以参考相关资料。

接着，我们分析第二类问题。这里，我们只关心一致且偏真的判定问题。下面给出这类问题的正确率分析：

首先，讨论判定问题正解为假的情况
由于算法是一致且偏真的
所以这种情况下，任意次调用算法$MC(x)$均返回$false$，正确率为100%

再讨论判定问题本身为真的情况
设此时一次调用算法$MC(x)$返回$true$的概率为p，则返回$false$的概率为$1-p$
因为算法一致偏真，所以此时调用算法n次得到错误结果的概率是$(1-p)^n$
得到正确解的概率是$1-(1-p)^n$，也即此种情况的正确率

设问题本身为真的概率为q，则为假的概率为$1-q$
所以，调用n次算法$MC(x)$的总体正确率为$q\left[1-(1-p)^n\right]+1-q$

由以上分析可以看到，对于一致偏真的 Monte-Carlo 算法，即使调用一次得到正确解的概率非常小，通过多次调用，其正确率会迅速提高，得到的结果非常可靠。例如，对一个 q 为 0.5 的问题，假设 p 仅为 0.01，通过调用 1000 次，其正确率约为 0.9999784，几乎可以认为是绝对准确的。重要的是，使用 Monte-Carlo 算法解判定问题，其正确率不随问题规模而改变，这就使得仅需要损失微乎其微的正确性，就可以将算法复杂度降低一个数量级，在后面中可以看到具体的例子。

应用实例一：使用 Monte-Carlo 算法计算定积分

计算定积分是金融、经济、工程等领域实践中经常遇到的问题。通常，计算定积分的经典方法是使用 Newton-Leibniz 公式：

$$Newton-Leibniz公式：$$
$$\int_a^b f(x)dx = F(b) - F(a)$$
$$其中F(x)为f(x)的原函数$$

这个公式虽然能方便计算出定积分的精确值，但是有一个局限就是要首先通过不定积分得到被积函数的原函数。有的时候，求原函数是非常困难的，而有的函数，如 f(x) = (sinx)/x，已经被证明不存在初等原函数，这样，就无法用 Newton-Leibniz 公式，只能另想办法。

下面就以 f(x) = (sinx)/x 为例介绍使用 Monte-Carlo 算法计算定积分的方法。首先需要声明，f(x) = (sinx)/x 在整个实数域是可积的，但不连续，在 x = 0 这一点没有定义。但是，当 x 趋近于 0 其左右极限都是 1。为了严格起见，我们补充定义当 x = 0 时 f(x) = 1。另外为了需要，这里不加证明地给出 f(x)的一些性质：补充 x = 0 定义后，f(x)在负无穷到正无穷上连续、可积，并且有界，其界为 1，即|f(x)| <= 1，当且仅当 x = 0 时 f(x) = 1。

下面开始介绍 Monte-Carlo 积分法。为了便于比较，在本节我们除了介绍使用 Monte-Carlo 方法计算定积分外，同时也探讨和实现数值计算中常用的插值积分法，并通过实验结果数据对两者的效率和精确性进行比较。

1、插值积分法

我们知道，对于连续可积函数，定积分的直观意义就是函数曲线与 x 轴围成的图形中，y>0 的面积减掉 y<0 的面积。那么一种直观的数值积分方法是通过插值方法，其中最简单的是梯形法则：用以 f(a)和 f(b)为底，x 轴和 f(a)、f(b)连线为腰组成的梯形面积来近似估计积分。如下图所示。

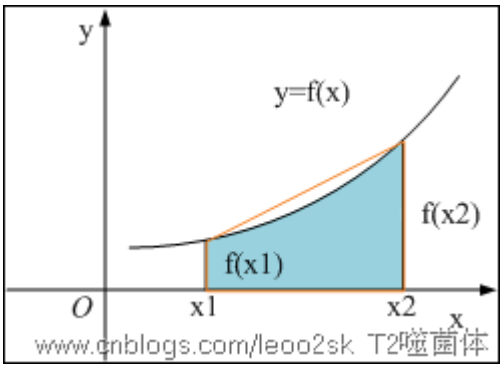

图 2、梯形插值

如图 2 所示，蓝色部分是 x1 到 x2 积分的精确面积，而在梯形插值中，用橙色框所示的梯形面积近似估计积分值。

显然，梯形法则的效果一般，而且某些情况下偏差很大，于是，有人提出了一种改进的方法：首先将积分区间分段，然后对每段计算梯形插值再加起来，这样精度就大大提高了。并且分段越多，精度越高。这就是复化梯形法则。

除了梯形插值外，还有许多插值积分法，比较常见的有 Sinpson 法则，当然对应的也有复化 Sinpson 法则。下面给出四种插值积分的公式：

梯形法则： $\int_a^b f(x)dx \approx \frac{b-a}{2}[f(a)+f(b)]$

复化梯形法则： $\int_a^b f(x)dx \approx \sum_{i=1}^{n} \frac{x_i - x_{i-1}}{2}[f(x_{i-1})+f(x_i)]$

*Sinpson*法则： $\int_a^b f(x)dx \approx \frac{b-a}{6}[f(a)+4f(\frac{a+b}{2})f(b)]$

复化*Sinpson*法则： $\int_a^b f(x)dx \approx \sum_{i=1}^{n/2} \frac{b-a}{3n}[f(x_{2i-2})+4f(x_{2i-1})+f(x_{2i})]$

下面是四种插值积分法的程序代码，用 C#编写。

```
  1: using System;
  2: using System.Collections.Generic;
  3: using System.Linq;
  4: using System.Text;
  5:
  6: namespace MonteCarlo.Integration
  7: {
  8:     /// <summary>
  9:     /// 数值法求积分
 10:     /// 被积函数为 f(x) = (sin x)/x
 11:     /// </summary>
 12:     public class NumericalIntegrator
 13:     {
 14:         /// <summary>
 15:         /// 梯形法则求积分
 16:         /// 积分公式为：((b - a) / 2) * [f(a) + f(b)]
 17:         /// </summary>
 18:         /// <param name="a">积分下限</param>
 19:         /// <param name="b">积分上限</param>
 20:         /// <returns>积分值</returns>
 21:         public static double TrapezoidalIntegrate(double a,
double b)
```

```
22:         {
23:             return ((b - a) / 2) * (Math.Sin(a) / a + Math.Sin(b)
/ b);
24:         }
25:
26:         /// <summary>
27:         /// 复化梯形法则求积分
28:         /// 积分公式为：累加((xi - xi-1) / 2) * [f(xi) + f(xi-1)]
(i=1,2,...,n)
29:         /// </summary>
30:         /// <param name="a">积分下限</param>
31:         /// <param name="b">积分上限</param>
32:         /// <param name="n">分段数量</param>
33:         /// <returns>积分值</returns>
34:         public static double ComplexTrapezoidalIntegrate(double a,
double b, int n)
35:         {
36:             double result = 0;
37:             for (int i = 0; i < n; i++)
38:             {
39:                 double xa = a + i * (b - a) / n;//区间积分下限
40:                 double xb = xa + (b - a) / n;//区间积分上限
41:
42:                 result += ((xb - xa) / 2) * (Math.Sin(xa) / xa +
Math.Sin(xb) / xb);
43:             }
44:
45:             return result;
46:         }
47:
48:         /// <summary>
49:         /// Sinpson法则求积分
50:         /// 积分公式为：((b - a) / 6) * [f(a) + 4 * f((a + b) / 2)
+ f(b)]
51:         /// </summary>
52:         /// <param name="a">积分下限</param>
53:         /// <param name="b">积分上限</param>
54:         /// <returns>积分值</returns>
55:         public static double SinpsonIntegrate(double a, double b)
56:         {
57:             return ((b - a) / 6) * (Math.Sin(a) / a + 4 * (Math.Sin(a
+ b) / (2 * (a + b))) + Math.Sin(b) / b);
58:         }
59:
```

```
60:         /// <summary>
61:         /// 复化 Sinpson 法则求积分
62:         /// 积分公式为：累加(h / 3) * [f(x2i-2) + 4*(f(x2i-1)) +
f(x2i)]  (i=1,2,...,n/2 h = (b - a) / n)
63:         /// </summary>
64:         /// <param name="a">积分下限</param>
65:         /// <param name="b">积分上限</param>
66:         /// <param name="n">分段数量(必须为偶数)</param>
67:         /// <returns>积分值</returns>
68:         public static double ComplexSinpsonIntegrate(double a,
double b, int n)
69:         {
70:             double result = 0;
71:             for (int i = 0; i < n / 2 - 1; i++)
72:             {
73:                 double xa = a + 2 * i * (b - a) / n;//区间积分下
限
74:                 double xb = xa + (b - a) / n;//区间积分限中点
75:                 double xc = xb + (b - a) / n;//区间积分上限
76:                 result += ((b - a) / (3 * n) * (Math.Sin(xa) / xa
+ 4 * (Math.Sin(xb) / xb) + Math.Sin(xc) / xc));
77:             }
78:
79:             return result;
80:         }
81:     }
82: }
```

2、Monte-Carlo 积分法

我们知道，求定积分的直观意义就是求面积，所以，用 Monte-Carlo 求积分的原理就是通过模拟统计方法求解面积。即通过向特定区域随机产生大量点，然后统计点落在函数区域内的频率，以此频率估计面积，从而得到积分值。下面给出 Monte-Carlo 求取积分的算法程序。

```
1: using System;
2: using System.Collections.Generic;
3: using System.Linq;
4: using System.Text;
5:
6: namespace MonteCarlo.Integration
7: {
8:     /// <summary>
```

```
    /// Monte-Carlo 法求积分
    /// 被积函数为 f(x) = (sin x)/x
    /// </summary>
    public class MonteCarloIntegrator
    {
        /// <summary>
        /// 用 Monte-Carlo 法求解积分值
        /// </summary>
        /// <param name="a">积分下限</param>
        /// <param name="b">积分上限</param>
        /// <param name="N">模拟次数</param>
        /// <returns>积分值</returns>
        public static double MonteCarloIntegrate(int a, int b, int N)
        {
            Random random = new Random();
            int positivePointCount = 0;//y >=0 区间内落入函数曲线内的点数目
            int negativePointCount = 0;//y < 0 区间内落入函数曲线内的点数目

            //统计 y >= 0 区间点分布
            for (int i = 0; i < N; i++)
            {
                double xCoordinate = random.NextDouble();//随机产生的 x 坐标
                double yCoordinate = random.NextDouble();//随机产生的 y 坐标
                xCoordinate = a + (b - a) * xCoordinate;//将 x 规格化到相应积分区间
                //yCoordinate = 1 * yCoordinate;//将 y 规格化到相应区间
                if (Math.Sin(xCoordinate) / xCoordinate >= yCoordinate)
                {
                    positivePointCount++;
                }
            }

            //统计 y < 0 区间点分布
            for (int i = 0; i < N; i++)
            {
                double xCoordinate = random.NextDouble();//随机产生的 x 坐标
```

```
 44:                 double yCoordinate = random.NextDouble();//随机
产生的 y 坐标
 45:                 xCoordinate = a + (b - a) * xCoordinate;//将 x
规格化到相应积分区间
 46:                 yCoordinate = -1 * yCoordinate;//将 y 规格化到相
应区间
 47:                 if (Math.Sin(xCoordinate) / xCoordinate <=
yCoordinate)
 48:                 {
 49:                     negativePointCount++;
 50:                 }
 51:             }
 52:
 53:             double positiveFrequency =
(double)positivePointCount / (double)N;//y >= 0 区间内函数内点频率
 54:             double negativeFrequency =
(double)negativePointCount / (double)N;//y < 0 区间内函数内点频率
 55:
 56:             return (positiveFrequency - negativeFrequency) *
(double)(b - a);
 57:         }
 58:     }
 59: }
```

3、积分法的测试与比较

下面对各种积分方法进行测试，对 sinx/x 在[1,2]区间上进行定积分。其中，我们分别对复化梯形和复化 Sinpson 法则做分段为 10，10000，和 10000000 的积分测试。另外，对 Monte-Carlo 法也投点数也分为 10，10000，和 10000000。测试结果如下：

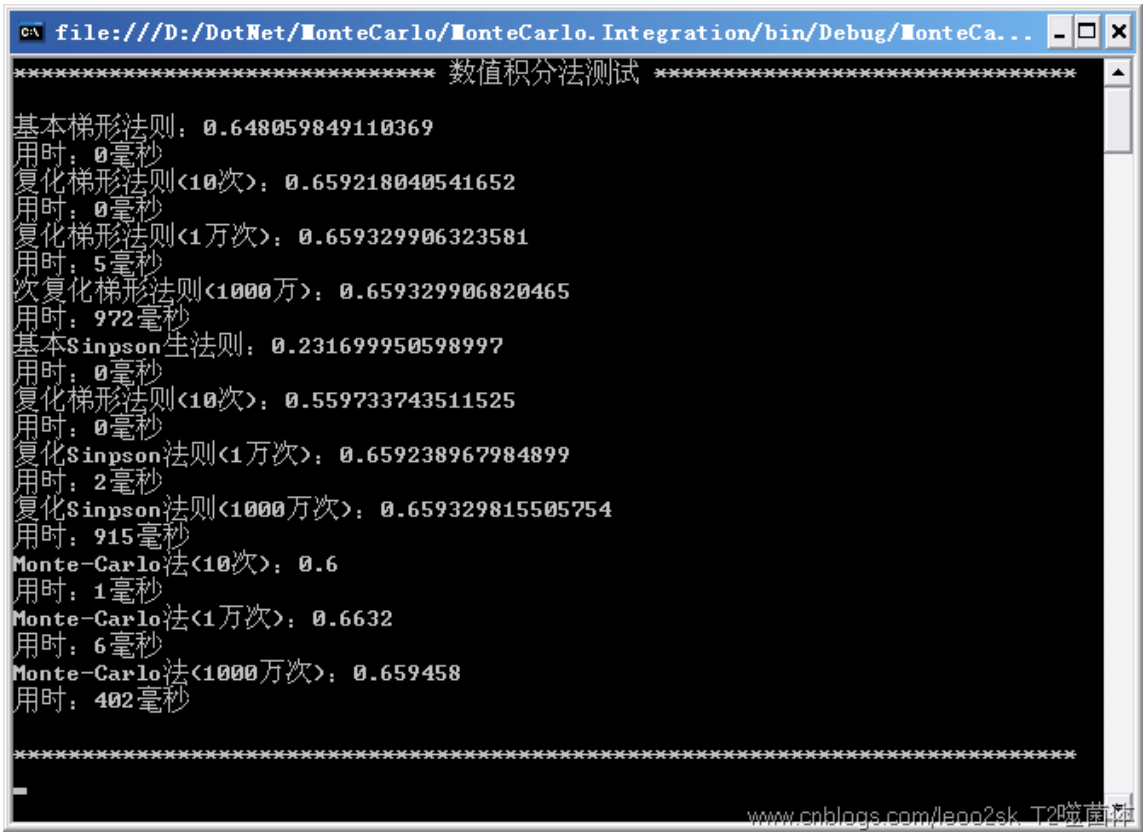

图 3、积分法测试结果

为了分析偏差，我们必须给出一个精确值。但是现在我手头没有这个积分的精确值，不过 1000 万次的梯形法则和 Sinpson 法则已经精确度很高了，所以这里就以 0.65932985 作为基本，进行误差分析。下面给出分析结果：

	绝对误差	相对误差	执行时间
梯形法则	0.01127	1.7%	<1ms
10-复化梯形法则	0.0001118	0.016958%	<1ms
1 万-复化梯形法则	0.0000005632358	0.00000854%	5ms
1000 万-复化梯形法则	0.00000005682	0.0000086179%	972ms
Sinpson 法则	0.4276298994	64.858%	<1ms
10-复化 Sinpson 法则	0.0995961	15.1%	<1ms
1 万-复化 Sinpson 法则	0.000090882	0.01378%	2ms
1000-复化 Sinpson 法则	0.000000034494	0.0000052317%	915ms
10-MonteCarlo 方法	0.05932985	6.1684%	1ms
1 万-MonteCarlo 方法	0.00402985	0.69315%	6ms
1000 万-MonteCarlo 方法	0.00006165	0.02957%	402ms

表 1、积分方法实验结果

首先看时间效率。当频度较低时，各种方法没有太多差别，但在 1000 万级别上复化梯形和复化 Sinpson 相差不大，而 Monte-Carlo 算法的效率快一倍。

而从准确率分析，当频度较低时，几种方法的误差都很大，而随着频度提高，插值法要远远优于 Monte-Carlo 算法，特别在 1000 万级别时，Monte-Carlo 法的相对误差是插值法的的近万倍。总体来说，在数值积分方面，Monte-Carlo 方法效率高，但准确率不如插值法。

应用实例二：在 O(n)复杂度内判定主元素

这次 ,我们看一个判定问题。问题是这样的 :*在一个长度为 n 的数组中 ,如果有超过[n/2]的元素具有相同的值，那么具有这个值的元素叫做数组的主元素。现在要求给出一种算法，在 O(n)时间内判定给定数组是否存在主元素。*

如果采用确定性算法，由于最坏情况下要搜索 n/2 次，而每次要比较的次数为 O(n)量级，这样，算法的复杂度就是 O(n^2)，不可能在 O(n)时间内完成。所以我们只好换一种思路：不是要一个一定正确的结果，而只需要结果在很大概率上正确就行。我们可以这样做：

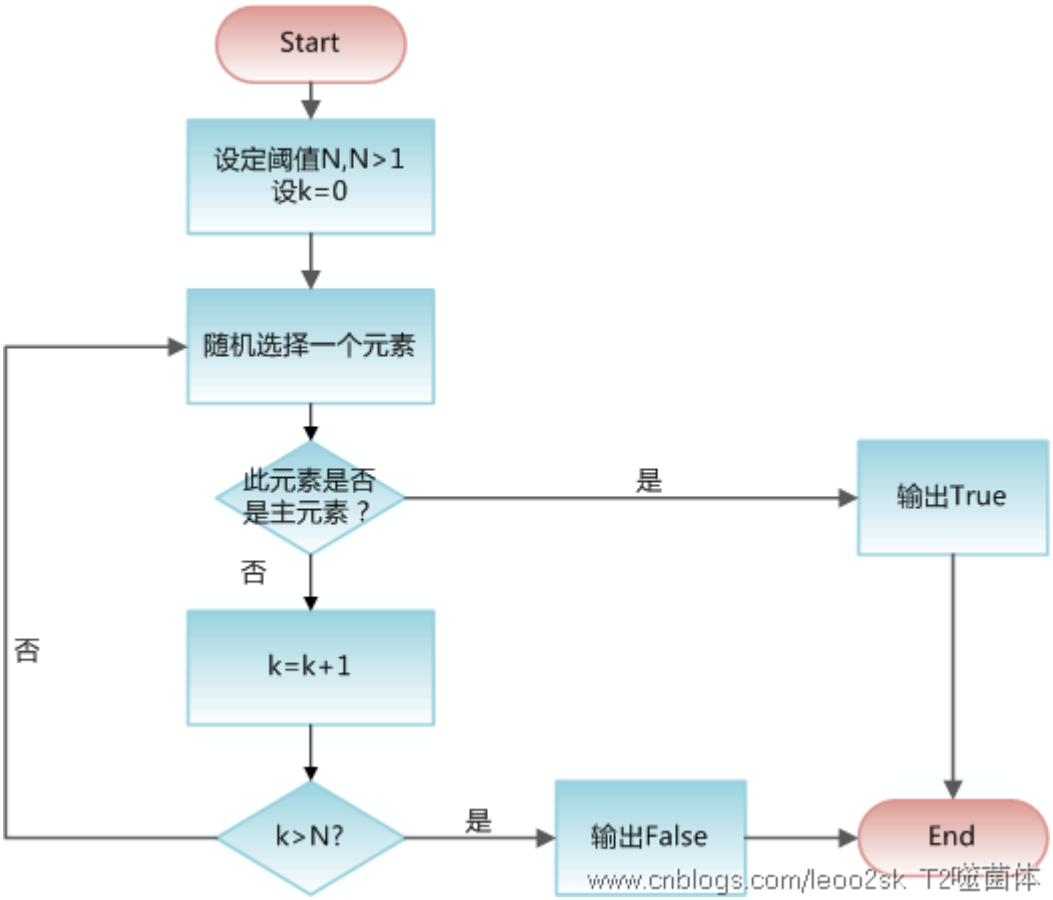

图 4、Monte-Carlo 法判定主元素

上述算法，就是用 Monte-Carlo 思想求解主元素判定问题的过程。由于阈值 N 是一个给定的常数，不随规模变化而变化，所以这个算法的时间复杂度为 O(n)，符合题设要求。但这个算法给出的解并不是 100%正确的，正确率和 N 有关。N 设得过大，影响效率，N 太小，正确率太低，那么到底 N 设多大合适呢。这就要对算法进行概率分析。

首先，这个算法是一致且偏真的，证明很简单，这里从略。所以，如果数组中不存在主元素，则结果一定正确，而如果存在，调用一次得到正确结果的概率不低于 1/2。由于偏真，在 N 次调用中只要返回一次 True，就可以认为得到正确结果，所以，调用 N 此得到正确结果的概率不低于 1 – (1/2)^N，可以看到，随着 N 的增大，这个概率增加很快，只要调用 10 次，正确率就可以达到 99.9%，重要的是，这个正确率和规模无关，即使数组的元素有 1 千万亿，只需调用 10 次，正确率依然是 99.9%，这就体现出在数组很大时，Monte-Carlo 方法的优势。

下面是使用 Monte-Carlo 算法进行主元素测试的 C#程序示例。

```
1: using System;
2: using System.Collections.Generic;
3: using System.Linq;
4: using System.Text;
5:
6: namespace MonteCarlo.Detection
7: {
8:     public class PrincipalElementDetector
9:     {
10:         /// <summary>
11:         /// 使用 Monte-Carlo 发探测主元素
12:         /// </summary>
13:         /// <param name="elements">所有元素</param>
14:         /// <param name="N">阈值</param>
15:         /// <returns>是否存在主元素</returns>
16:         public static bool DetectPrincipalElement(IList<int> elements,int N)
17:         {
18:             Random random = new Random();
19:             bool result = false;
20:             for (int i = 0; i <= N; i++)
21:             {
22:                 int index = random.Next(0, elements.Count - 1);
23:                 int element = elements[index];
24:                 int count = 0;
25:                 for (int j = 0; j < elements.Count; j++)
26:                 {
27:                     if (element == elements[j])
28:                     {
29:                         count++;
30:                     }
31:                 }
32:                 if (count >= elements.Count / 2)
33:                 {
34:                     result = true;
```

```
35:                     break;
36:                 }
37:             }
38:
39:             return result;
40:         }
41:     }
42: }
```

程序很简单，不做赘述。下面测试这个算法。我们分别将阈值设为 1、3、10，并且在每个阈值下测试 100 次，看看这个算法的准确率如何。测试数组是[4, 5, 8, 1, 8, 4, 9, 2, 2, 2, 2, 2, 5, 7, 8, 2, 2, 2, 2, 2, 1, 0, 9, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 4, 7, 8, 2, 2, 2, 2, 2, 0, 1, 2, 2, 2, 2, 2]，其中存在主元素 2。下面是测试结果：

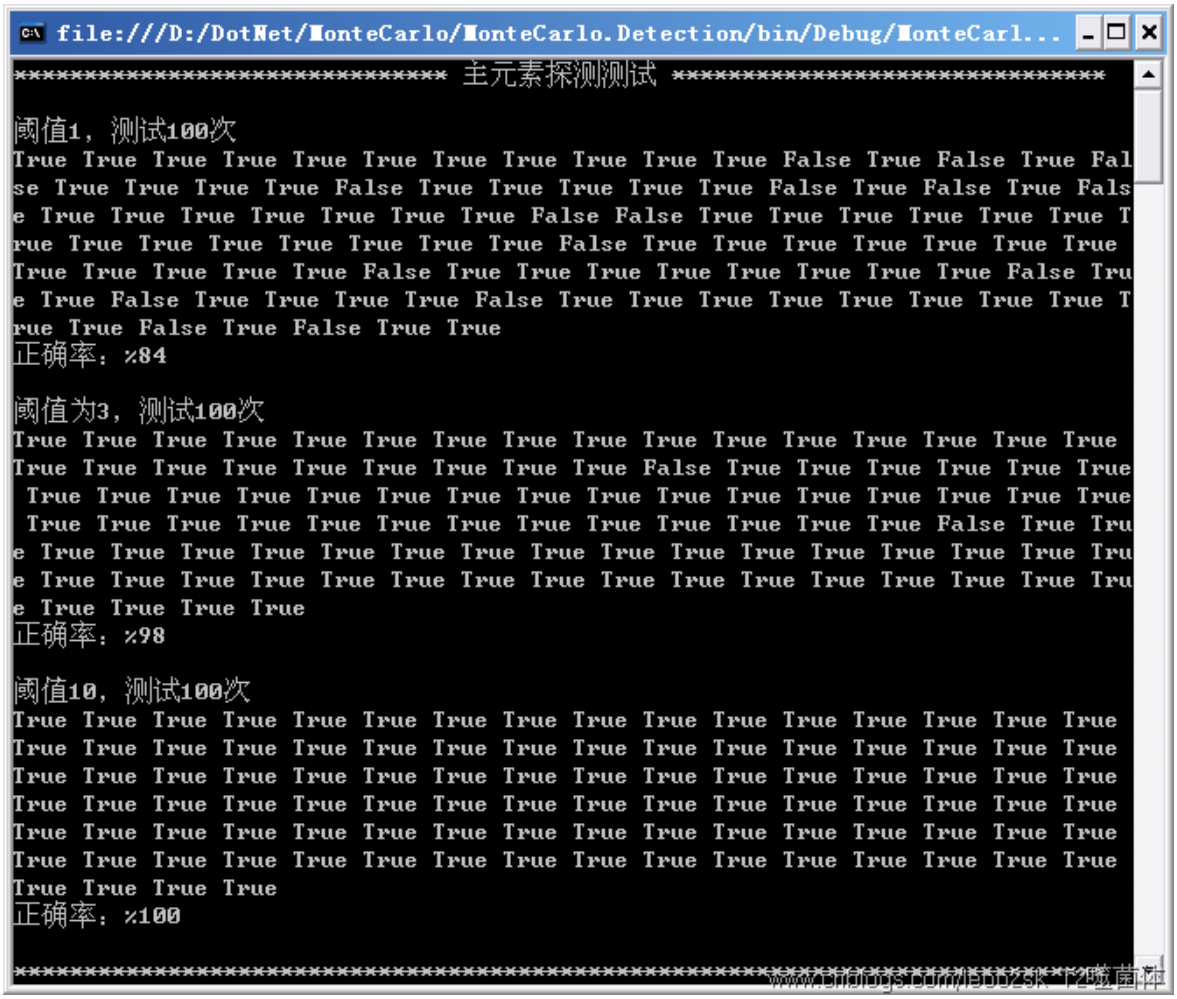

图 5、Monte-Carlo 算法判定主元素实验结果

测试数组有 49 个元素，主元素 2 有 29 个，比率为 59%。从测试结果可以看出，即使阈值为 1，正确率也高达 84%，而仅仅为 3 的阈值就使正确率升高到 98%，阈值为 10 时，100 次测试全部正确。虽然理论上来说，阈值为 10 时有 0.41^10=0.013%的概率给出错误判断，但是笔者多次试验，还没有在阈值为 10 时得到错误结果。所以，Monte-Carlo

方法求解判定问题，不论从理论上还是实践中，都是不错的方法。

另外一个与判定主元素类似的应用是素数判定问题，我们知道，对于寻找上百位的大素数，完全测试在时间效率上时不允许的。于是，结合费马小定理使用 Monte-Carlo 法进行素数判定，是广泛使用的方法。具体这里不再详述，感兴趣的朋友可以参考相关资料。

应用实例三：分布未知的概率密度函数模拟

现在我们来看看 Monte-Carlo 算法的第三种应用：模拟。在这种应用中，不再是用 Monte-Carlo 算法求解问题，而是用来模拟难以解析描述的东西。问题是这样的：

已知常数序列$a_0, a_1, \ldots, a_n$
以及相互独立的随机变量$X_1, X_2, \ldots X_n$
其中X_i的概率密度函数已知为$f_i(x)$

随机变量$Y = a_0 + a_1X_1 + a_2X_2 + \ldots + a_nX_n$
现要求Y的概率密度函数

这个问题是实验室一个师兄在开发 Six Sigma 软件开发过程管理工具时遇到的一个实际需求，最终 Y 的概率密度函数将被用来计算分位点，从而进行过程控制。其中 X 可能是正态分布（高斯分布）、泊松分布、均匀分布或指数分布等。将多个不同分布的概率密度函数相加，得到的 Y 的分布式很难解析表示出来的，但如果是为了计算分位点，我们可以采取这样一个策略：对于每一个 X，产生若干符合其分布的点，带入公式就得到若干符合 Y 分布的点，然后分段计算频率，从而模拟出 Y 的分布，这些模拟点也可以用于分位点计算。这就是 Monte-Carlo 模拟的思想。

下面我们实现这个算法，这里的 X 我们仅给出最常用的正态分布，如果要实现其他分布，只要编写相应的随机点发生器就可以了。由于 C#中只能产生符合均匀分布的随机数，所以我们需要一种算法，将均匀分布的随机数转为正态分布随机数。这种算法很多，Marc Brysbaert 在 1991 年发表的《Algorithms for randomness in the behavioral sciences: A tutorial》一文中，共总结了 5 种将均匀分布随机数转为正态分布的随机数的算法，这里笔者用到的是 Knuth 在 1981 年提出的一种算法。这个算法是将符合 u(0,1)均匀分布的随机点转换为符合 N(0,1)标准正态分布的随机点 p，由概率知识可知，要转为符合 N(e,v)的一般正态分布，只需进行 p*v+e 即可。下面是这个算法：

将符合$u(0,1)$的随机点转为符合$N(\mu,\sigma^2)$的随机点

Step1：

产生符合$u(0,1)$的随机点u_1,u_2

$V = 0.8578(2u_2 - 1)$

$Z = V / u_1$

$A = 0.25Z^2$

Step2：

如果$A < 1 - u_1$则到达Step3

如果$A > 0.259/u_1 + 0.35$则回到Step1

如果$A > -ln(u_1)$则回到Step1

Step3：

输出$Z\sigma+\mu$

下面是根据这个算法，使用 C#编写的正态分布随机点发生器：

```
 1: using System;
 2: using System.Collections.Generic;
 3: using System.Linq;
 4: using System.Text;
 5:
 6: namespace MonteCarlo.DistributingSimulation
 7: {
 8:     public class NormalDistributingGenerator
 9:     {
10:         /// <summary>
11:         /// 产生符合正态分布的随机数
12:         /// 正态分布的期望为 expectation,方差为 variance
13:         /// </summary>
14:         /// <param name="expectation">期望</param>
15:         /// <param name="variance">方差</param>
16:         /// <param name="N">产生的数量</param>
17:         /// <returns>随机数序列</returns>
18:         public static IList<double> GenerateNDRNumber(double expectation, double variance,
int N)
19:         {
20:             Random random = new Random();
21:             IList<double> randomList = new List<double>();
22:             for (int i = 0; i < N; i++)
23:             {
24:                 double u1, u2, v, z, a;
```

```
                do
                {
                    u1 = random.NextDouble();
                    u2 = random.NextDouble();
                    v = 0.8578 * (2 * u2 - 1);
                    z = v / u1;
                    a = 0.25 * Math.Exp(2);

                    if (a < 1 - u1)
                    {
                        break;
                    }

                } while (a > 0.295 / u1 + 0.35 || a > -Math.Log(u1, Math.E));

                randomList.Add(z * Math.Sqrt(variance) + expectation);
            }

            return randomList;
        }
    }
}
```

接着是利用这个正态分布发生器获得 X 的随机值，并计算出 Y 的随机值的代码。也就是 Y 的随机点发生器：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MonteCarlo.DistributingSimulation
{
    public class DistributingSimulator
    {
        /// <summary>
        /// 模拟多个正态分布之和的分布情况，产生符合复合分布的随机点
        /// y = a0 + a1*N(e1,v1) + ... + an*N(en,vn)
        /// N(e,v)表示期望为 e，方差为 v 的正态分布
        /// </summary>
        /// <param name="a">常数列</param>
        /// <param name="e">期望列</param>
        /// <param name="v">方差列</param>
        /// <param name="N">产生模拟点的个数</param>
        /// <returns>模拟点序列</returns>
```

```
20:         public static IList<double> Simulate(IList<double> a,IList<double>
e,IList<double> v,int N)
21:         {
22:             IList<double> result = new List<double>();
23:             IList<IList<double>> randomLists = new List<IList<double>>();
24:             int count = a.Count - 1;
25:
26:             //产生各个自变量的随机序列
27:             for (int i = 1; i <= count; i++)
28:             {
29:                 randomLists.Add(NormalDistributingGenerator.GenerateNDRNumber(e[i],
v[i], N));
30:             }
31:
32:             //带入公式
33:             for (int j = 0; j < N; j++)
34:             {
35:                 double y = 0;
36:                 for (int k = 1; k <= count; k++)
37:                 {
38:                     y += a[k] * randomLists[k - 1][j];
39:                 }
40:                 y += a[0];
41:                 result.Add(y);
42:             }
43:
44:             return result;
45:         }
46:     }
47: }
```

这样，我们就可以产生任意多个符合 Y 分布的随机点，从而借此模拟 Y 的概率密度分布。

接着，我们测试一下这个模拟程序的效果，首先我们将初始值设为仅有一个符合标准正态分布的 X，这样 Y=X，我们看看直接模拟一个标准正态分布的效果。这里，我们产生 100 万个随机点。

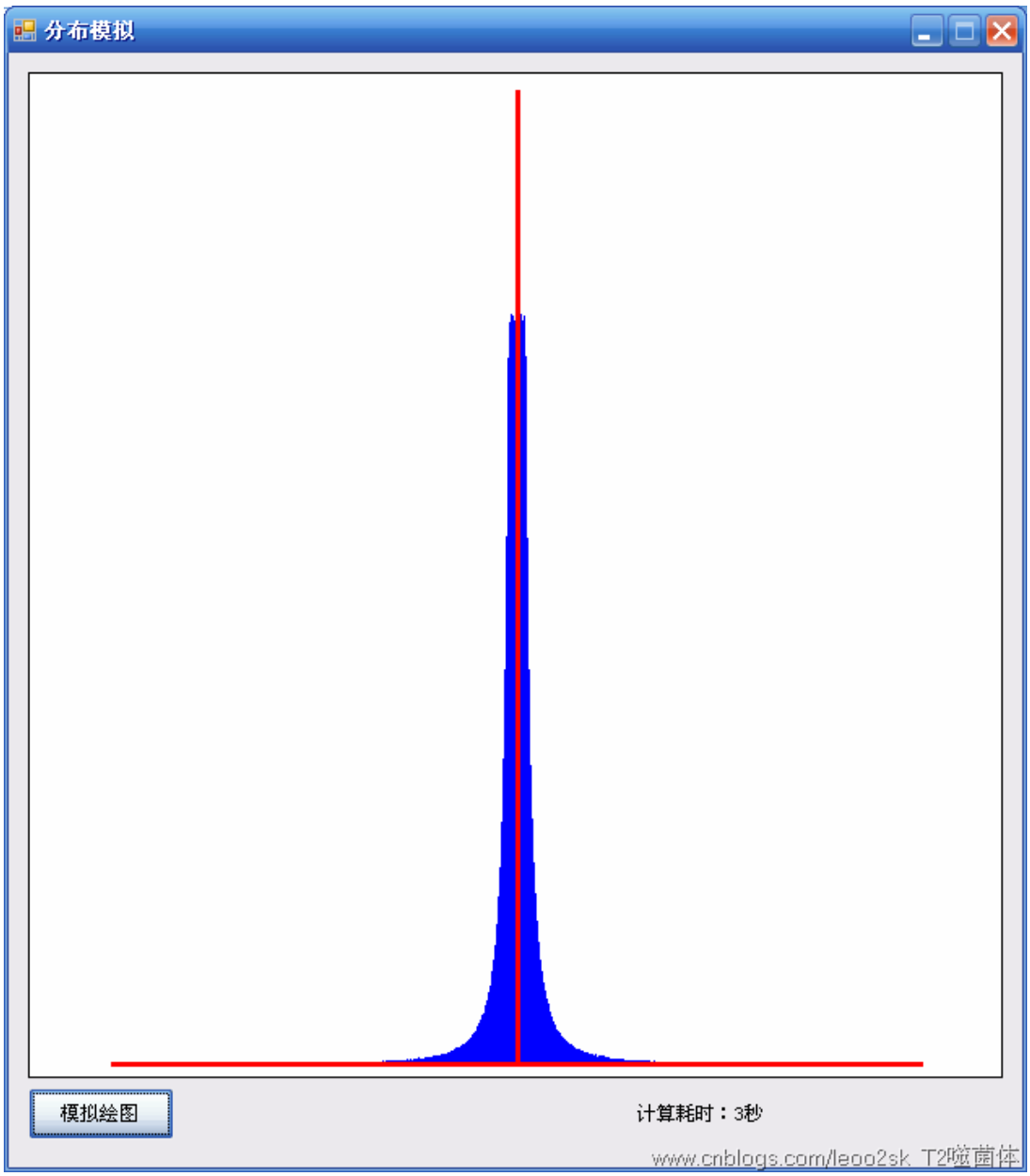

图 6、使用 Monte-Carlo 算法模拟标准正态分布

可以看到，模拟效果基本令人满意。接下来，我们实际应用这个程序模拟一个分布未知的 Y，其中 Y = 15 + 2*N(2,8) + 5*N(-10,9) + 7*N(0,0.5)。模拟结果如下：

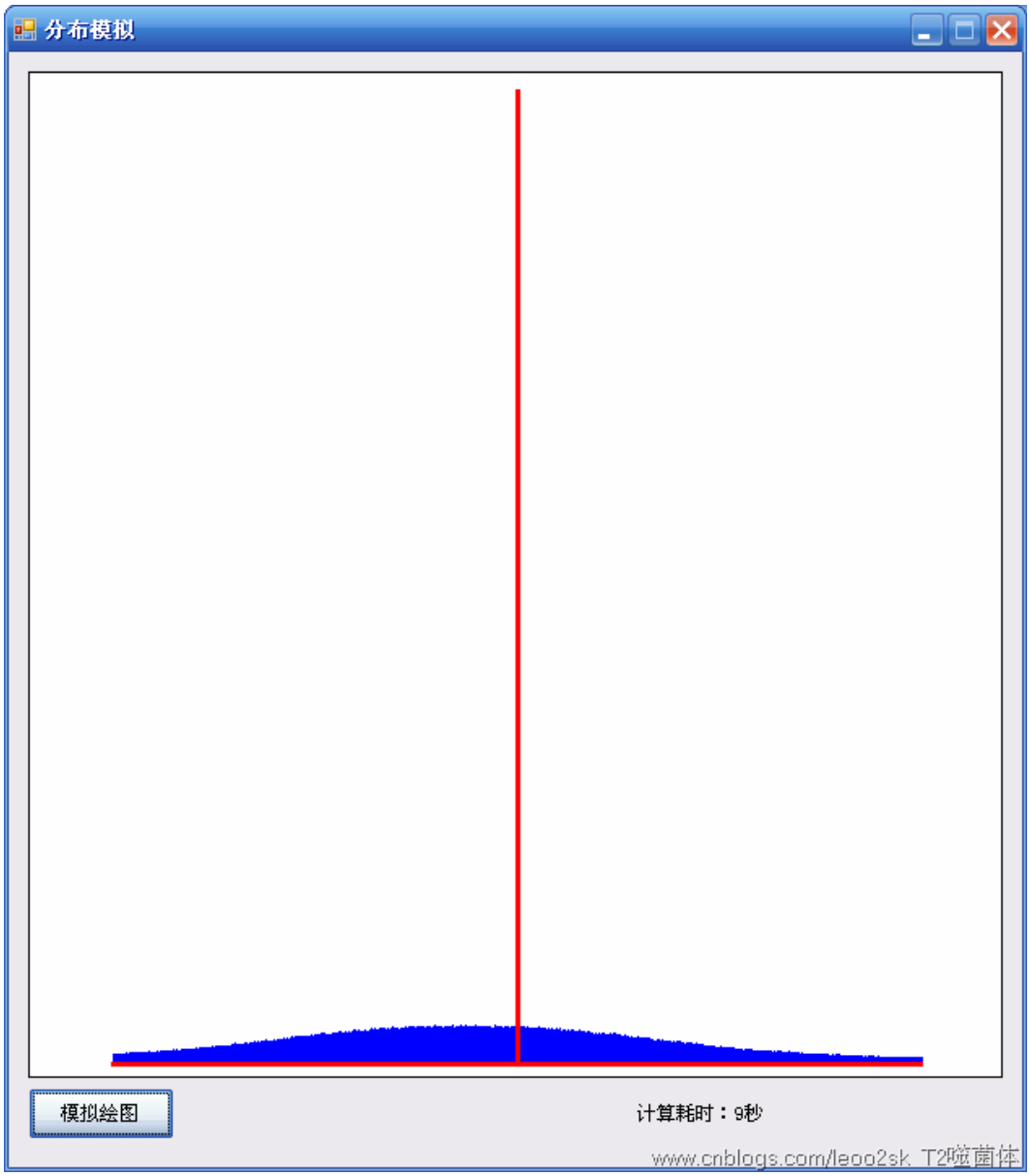

图 7、使用 Monte-Carlo 算法模拟未知分布

有了符合 Y 分布的大量随机点以及频率统计，就可以随心所欲绘出分布模拟图，并进行分位点计算。这样就用 Monte-Carlo 算法解决了本节开头提到的问题。

总结

本文首先通过一个不规则图形面积计算的例子直观介绍了 Monte-Carlo 算法，然后给出了 Monte-Carlo 算法在应用过程中需要了解的数理基础。然后大篇幅介绍了三个应用：计算、判定和模拟。

总体来说，当需要求解的问题依赖概率时，Monte-Carlo 方法是一个不错的选择。但这个算法毕竟不是确定性算法，在应用过程中需要冒一定“风险”，这就要求不能滥用这个

算法，在应用过程中，需要对其准确率或正确率进行数理分析，合理设计实验，从而得到良好的结果，并将风险控制在可容忍的范围内。

其实，不确定性算法不只 Monte-Carlo 一种，Sherwood 算法、Las Vegas 算法和遗传算法等也是经典的不确定算法。在很多问题上，不确定性算法具有很好大的应用价值。有兴趣的朋友可以参考相关资料。

本文用到的实验程序完整版和本文的 PDF 版可以在这里下载：

参考文献

[1] 孙海燕，周梦等 著，应用数理统计。北京航空航天大学出版社，2008.8

[2] 盛骤，谢式千，潘承毅 著，概率论与数理统计。高等教育出版社，2006.12

[3] David Kincaid ,WardCheney 著 ,王国荣等 译 ,数值分析(原书第三版)。机械工业出版社 ,2005.9

[4] Thomas H. Cormen 等 著，算法导论(第二版，影印版)。高等教育出版社，2002.5

[5] 王晓东 著，计算机算法设计与分析。电子工业出版社，2001.1

[6] Marc Brysbaert，Algorithms for randomness in the behavioral sciences: A tutorial。Behavior Research Methods, Instruments, & Computers 1991, 23 (1) 45-60

[7] Patrick Smacchia 著，施凡等 译，C#和.NET2.0 平台、语言与框架。2008.1

[8] Google。www.google.com

[9] Wikipedia。www.wikipedia.org

作者：EricZhang(T2 噬菌体)

出处：http://leoo2sk.cnblogs.com