Python下浅浅的探究随机数的产生方法

#python #算法

前言

我们对于随机数肯定不会陌生，随机数早已成为了我们经常要用到的一个方法，比如用于密码加密，数据生成，蒙特卡洛算法等等都需要随机数的参与.那么我们的电脑是怎么才能够产生随机数的呢？是电脑自己的物理存在还是依靠算法？它到底是如何工作的呢？所以我也对这些问题有着好奇心，所以找到了许多资料学习了一下，现在就整理下将自己的理解分享给大家.

随机数的概念

这里我们先来看一下数学上对随机数这个词的定义，定义如下：
在连续型随机变量的分布中，最简单而且最基本的分布是单位均匀分布.由该分布抽取的简单子样称为随机数序列，其中每一个体称为随机数.单位均匀分布即［0，1］上的均匀分布.
由随机数序列的定义可知，ξ1，ξ2，…是相互独立且具有相同单位均匀分布的随机数序列.也就是说，独立性、均匀性是随机数必备的两个特点.

随机数的真伪

为什么说随机数会分成真和伪呢？那是因为，真正的随机数，是物理中量子力学的概念，而我们如果想要真正做到真随机，那么就需要真正的独立于电脑的特殊设备来实现生成真正的随机数（比如英特尔曾经的芯片组是通过用热噪声放大后，影响电压控制的振荡器并用另外一个高频振荡器来接受数据得到随机数）以及往大了来说自然界的一切都可以用于作为随机数发生器的因子.而另一种通过算法实现的被称为伪随机的方法是指根据算法和指定的不确定因素(也就是种子seed)，我们在电脑中常会用到电脑时间来做为seed的值，但是这样的话，除非是毫秒级甚至以下的单位，否则在很短的时间内生成大量的随机数，很容易发生时间重合，造城最终随机数相同的情况.所以，在伪随机数中，种子(seed)的指定就显得相当的重要了.据说UNIX系统中，将系统时间，连入WIFI，甚至按下的键盘次数都量化为了seed，参考指标越多，伪随机数就越接近真正的随机生成.

产生伪随机数的方法

产生伪随机数有很多的方法，而在本文中，将分别对其中的几种伪随机数生成方法进行介绍和实践.方法如下：

- 线性同余方法

- 平方取中方法
- 梅森旋转随机生成法
- BOX-MULLER法

随机数产生方法好坏的判定

对于这个问题，我在网上搜到了相当之多的判定方法，这之中有许多的判定标准和判断工具.比如像NIST测试方法，TestU01的测试工具等等，这里我就贴上德联信息安全工作室的一套方法（其他方法作为参考内容会放在文末的文献参考大家也可以去看看了解一下相关的知识）
德国联邦安全办公室（下简称德联安全办）总共公布了四个等级，这里就稍微翻译一下大致的意思：

1. 包含相同随机数序列的概率很低.
2. 根据指定的统计检验区别于"真随机数"的数字序列.
3. 它不可能被任何人能够通过计算得到，或以其他的方法进行猜测，得到任何给定的一个子序列，以及任何工作中产生的一切序列中的值，以及迭代器的工作状态.
4. 对于任何目的而言，任何人不能从随机数生成器的内部状态计算或猜测得到序列中的任何一个之前的数字 / 序列以及任何之前的随机数的状态.

产生随机数的方法

我们经常会对随机数有一定的要求范围，那么随机数产生的一般步骤是怎样的呢？因为真随机涉及到了物理的量子.故本文全文只讨论伪随机数的生成方法，我们在python 中的random库中，经常会用到如randint之类的方法来生成一定范围内的随机数.这之中主要用到的方法步骤为->指定一个特定的seed种子，根据seed再通过特定的随机数产生方法，就可以做到在[0，1]这个范围中取到随机分布的随机数，然后一定的变换，我们就可以得到一定范围内的随机数了.

伪随机随机数算法

○平方取中法生成随机数

平方取中法的思路是：

- 选择一个m位数N作为SEED种子->做平方运算（记（N* N））
- 判断N**N的长度，若结果不足2**m个位，在最前面补0.
- 在这个N*N的数选中间m个位的数作为随机数.

公式表示为：

$$N_{I+1} = \left[\frac{N_I}{10^{\frac{m}{2}}} \right] \bmod(10^m)$$

随机数为：

$$\Re = \frac{N_{I+1}}{10^m}$$

我们来看看平方取中的代码实现：

```
#平方取中 -xlxw
from time import time

def rander(seed,n):
    if n ==1:
        return 0
    seed = int(seed)
    length = len(str(seed))
    seed = int(seed**2/pow(10,(length/2))) % int(pow(10.0,length))
    print(str(seed) +" " ,end='')
    rander(seed,n-1)
def main():
    seed = time()
    rander(seed,100)

main()
```

生成的随机数的截图：

```
============ RESTART: /Users/xulvxiaowei/Desktop/Random/middle.py ============
8463539114 4943342079 6309100120 7443241840 8490887265 1665469391 7882923579 484
1523541 3501980571 8679196614 4538644690 2956220651 2405373988 8240221470 249874
6489 7340162898 9913691757 2842528097 9659822344 1677176416 9207303866 444480858
5 3233572897 9936802129 365508989 702426436 769617813 536712432 264448587 477380
597 585217439 151166785 624618801 583491700 371601624 718605983 829771364 930479
411 744921747 744657057 276535785 258249387 9804101 5938227 994217 467443 502958
 966749 603629 367969 401184 948601 843857 94636 21271 30789 97720 97216 86530 7
```

平方取中法的缺点：

- 周期很短
- 分布并不均匀

但是它因为计算迅速所以也常被用于随机数据的生成.用于试验等地方.

和它相近的方法还有：常数取中法和乘法取中法等

O线性同余法计算随机数

先来解释一下名字的含义，我们把线性同余拆开来理解.线性的意思是满足可加性和齐次性的映射；同余的意思就是如果给定一个数M,使两个整数a，b可以满足（a-b）能被M整除，则称a，b对M同余.另一种理解方法就是a和b同时被M除后取余数相等则为同余.

根据这种方法的两个特性，我们可以作如下两个式子

```
(a +/- b) mod M =  ((a mod M) +/- (b mod M))  mod m
    a*b mod M   =  ((a mod M) * (b mod M) mod M
```

我们来看一下线性同余的标准的式子：

$$X_{N+1} = (A * X_N + C) MOD(M)$$

我们来看一下这个式子的组成部分的含义：

- 我们可以从这个式子中看出，这是一个递归公式，所以X0为它的基，X0就是我们要给出的Seed种子.X0又叫做初始值.
- A的定义是系数
- C的定义是是增数
- M的定义是模数，也可以看作循环周期（最大）
- C,M,A是影响生成随机数质量的重要因素.

我们来看一下下面用Python写的模拟线性同余方法的代码：(由于每次只生成一个随机数，所以没有用到递归,seed用时间来表示)

这里M / A / C的值用了glibc (used by GCC) 编译器的参数值

M = 2^32 A = 1103515245 C = 12345

```
#Random - LCG Ver by xlxw
#import
from time import time,clock

#define
m = 2**32
a = 1103515245
```

```
c = 12345


def LCG(seed):
    seed = (a * seed + c) % m
    return seed/float(m-1)
def main():
    br = input("请输入随机数产生的范围(用,隔开):")
    mi = eval(br.split(',')[0])
    ma = eval(br.split(',')[1])
    seed = time()
    rd = LCG(seed)
    ourd = int((ma-mi)*rd) + mi
    print("随机生成的数字式：{}".format(ourd))
main()
```

效果截图：

通过一个seed生成多个随机数的代码：

```
#Random - LCG Ver by xlxw
#import
from time import time,clock


#define
m = 2**32
a = 1103515245
c = 12345
rdls = []
```

```
def LCG(seed,mi,ma,n):
    if n == 1:
        return 0
    else:
        seed = (a * seed + c) % m
        rdls.append(int((ma-mi)*seed/float(m-1)) + mi)
        LCG(seed,mi,ma,n-1)


def main():
    br = input("请输入随机数产生的范围(用,隔开):")
    co = eval(input("请输入需要产生的随机数的个数:"))
    mi = eval(br.split(',')[0])
    ma = eval(br.split(',')[1])
    seed = time()
    LCG(seed,mi,ma,co)
    print("随机生成的数字",rdls)


main()
```

运行结果截图:

```
>>>
============ RESTART: /Users/xulvxiaowei/Desktop/Random/rd-lcg.py ============
请输入随机数产生的范围(用,隔开):0,100
请输入需要产生的随机数的个数:100
随机生成的数字 [40, 41, 63, 86, 65, 29, 16, 34, 43, 64, 89, 16, 83, 27, 42, 69, 72, 54
, 64, 30, 97, 84, 40, 38, 47, 54, 60, 29, 9, 1, 55, 46, 23, 99, 72, 87, 50, 81, 49,
84, 63, 73, 55, 16, 4, 35, 20, 38, 5, 53, 1, 22, 92, 50, 85, 65, 20, 11, 63, 73, 25,
6, 47, 61, 39, 27, 88, 50, 46, 70, 61, 7, 39, 73, 43, 61, 31, 72, 20, 2, 53, 80, 32,
85, 25, 71, 15, 49, 53, 76, 43, 74, 42, 67, 91, 49, 70, 74, 70]
```

上面我们已经可以用线性同余来得到随机数了.那么我们来通过下面来看看生成的随机数的概率是不是相同的，我们这里顺便测试一下生成随机数的速度.

请看下面代码:

```
#Random - LCG Ver by xlxw
#import
from time import time,clock
import sys
sys.setrecursionlimit(20000)

#define
m = 2**32
a = 1103515245
```

```
c = 12345
rdls = []

def LCG(seed,mi,ma,n):
    if n == 1:
        return 0
    else:
        seed = (a * seed + c) % m
        rdls.append(int((ma-mi)*seed/float(m-1)) + mi)
        n = n-1
        LCG(seed,mi,ma,n)

def POSCheck():#得到各个数字出现次数
    counts = {}
    for i in rdls:
        if i in counts:
            counts[i] = counts.get(i,0) + 1
        else:
            counts[i] = 1
    print(counts)

def main():
    br = input("请输入随机数产生的范围(用,隔开):")
    co = eval(input("请输入需要产生的随机数的个数:"))
    mi = eval(br.split(',')[0])
    ma = eval(br.split(',')[1])
    seed = time()
    LCG(seed,mi,ma,co)
    POSCheck()
    #print("随机生成的数字",rdls)

main()
```

程序运行结果截图：（由于IDLE的递归深度限制在了1000以内，我通过修改了这个限制使得能递归20000次，再高则会失去响应，可能有什么限制，所以这里就拿生成20-29的随机数20000运行三次来大致看看是否均匀）

```
================================ RESTART: Shell ================================
>>>
========== RESTART: /Users/xulvxiaowei/Desktop/Random/rd-lcg-pos.py ==========
请输入随机数产生的范围(用,隔开):20,30
请输入需要产生的随机数的个数:20000
{20: 1920, 21: 1967, 22: 2009, 23: 2053, 24: 1936, 25: 1955, 26: 1985, 27: 2106,
 28: 2128, 29: 1940}
>>>
========== RESTART: /Users/xulvxiaowei/Desktop/Random/rd-lcg-pos.py ==========
请输入随机数产生的范围(用,隔开):20,30
请输入需要产生的随机数的个数:20000
{20: 1923, 21: 1954, 22: 2040, 23: 2055, 24: 1945, 25: 1982, 26: 1962, 27: 2077,
 28: 2118, 29: 1943}
>>>
========== RESTART: /Users/xulvxiaowei/Desktop/Random/rd-lcg-pos.py ==========
请输入随机数产生的范围(用,隔开):20,30
请输入需要产生的随机数的个数:20000
{20: 1752, 21: 1885, 22: 2203, 23: 2195, 24: 1589, 25: 1874, 26: 1965, 27: 2190,
 28: 2283, 29: 2063}
```

我们可以看到出现次数大致上都分布在2000次左右，所以差不多可以看作产生的随机数的概率是差不多一样的.

M/A/C的选定(为了生成高质量随机数)

- M:对于随机数的生成而言，随机数序列周期越长, 它就能够具有在[0, 1] 上均匀分布及相互独立.M越大周期就越大. 所以我们使得M接近于计算机能表示的最大的整数, 所以我们选择2^32作为M的值
- A：对于乘数而言，1.对于M的任何一个质因子P，a-1能被P整除.2.如果4是M的因子，则a除以4余1(资料中提到，证明在数论中有)
- C：增量C和M互质(证明同样在数论中)

以上条件可以用于生成高质量随机数，也就是将随机数的循环达到M的值(其他情况下都是小于循环M)

线性同余法的缺点

- 线性同余法生成的随机数周期太短.
- 如果被知道一定长度的序列就能破解出所有的随机数生成序列（梅森旋转也同样有这种缺点）

O梅森旋转法

- 梅森旋转法的产生：

> 梅森旋转算法是一个伪随机数发生算法.由松本真和西村拓士开发，是一个基于有限二进制字段上的矩阵线性递归的算法.可以快速产生高质量的伪随机数，修正了朴素随机数生成的缺陷.

梅森旋转法是通过线性反馈移位寄存器来生成随机数的.线性反馈移位寄存器-LFSR，是指给定前一状态的输出，将该输出的线性函数再用作输入的移位寄存器也就是对寄存器的某些位进行异或操作后作为输入，再对寄存器中的各比特进行整体移位.而LFSR由两个部分组成：

- 级数-一级一比特
- 反馈函数

解释：

- 一：异或操作

  异或逻辑运算（半加运算）

  异或运算通常用符号"⊕"表示,其运算规则为：

  0⊕0=0 0同0异或,结果为0

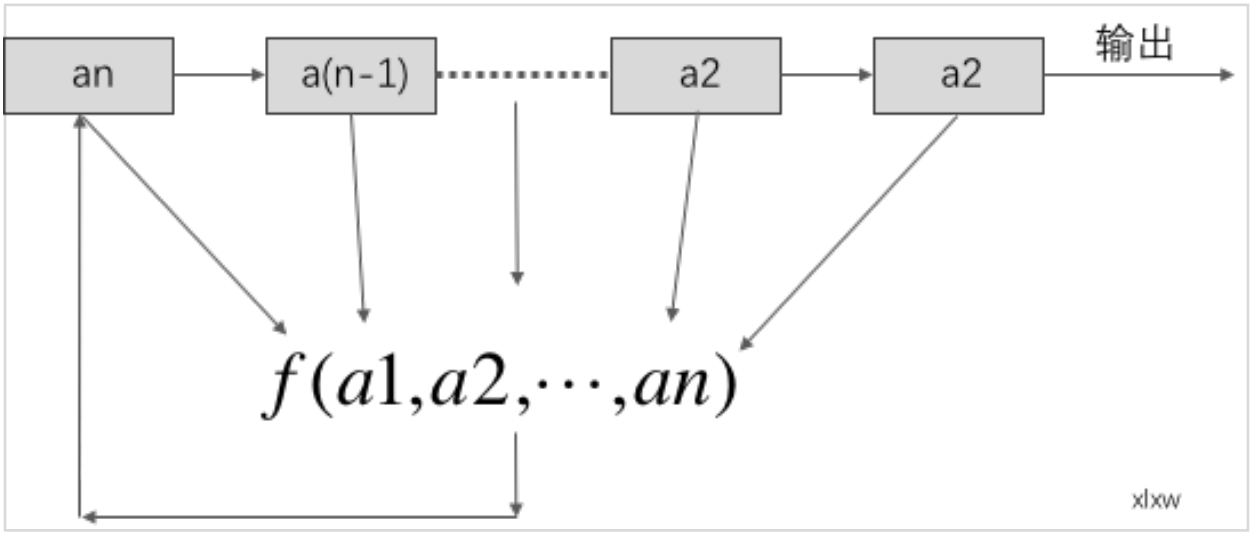
  0⊕1=1 0同1异或,结果为1

  1⊕0=1 1同0异或,结果为1

  1⊕1=0 1同1异或,结果为0

  即两个逻辑变量相异,输出才为1

- 二：线性反馈移位寄存器的图示：



- 三:某些位的说明

  在LFSR线性反馈移位寄存器的解释中说的"某些位"进行异或处理是由特征多项式来决定的.

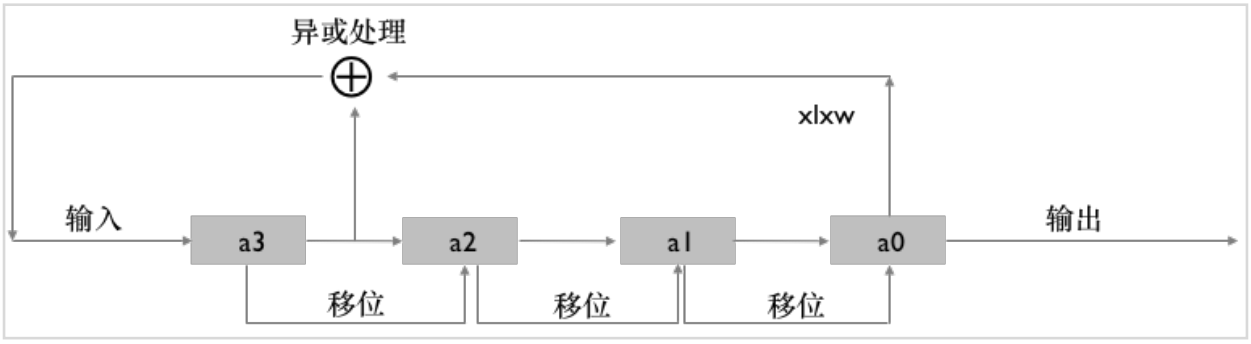
- 四：特征多项式的要求

  一个n阶的本原多项式，在LFSR中要求使用本原多项式的原因是本原多项式才能使线性反馈移存器的周期达到最大.而本原多项式指的是：一个n次不可约多项式，如果只能整除1＋x^2^n-1而不能整除其它1＋x^L(L<2^n-1)，则这种不可约多项式就称为本原多项式.

线性反馈移位寄存器的工作原理

例1.以一个4位的线性反馈移位寄存器为例说明它的工作原理

反馈函数为：$f(x)=x^4+x+1$，可根据反馈函数的多项式作出线性移位寄存器逻辑框图（如下图）：



设输入的初始状态为：

$$f(a_3,a_2,a_1,a_0)=(1,0,0,0)$$

那么计算步骤为：

（范例）

A3 A2 A1 A0

1 0 0 0

进行第一次异或判断

$$a_3'=a_3\oplus a_0=1\oplus 0=1$$

进行移位处理，并将a3'的状态给a3

A3 A2 A1 A0

1 1 0 0

之后再重复进行这个操作产生周期序列：

A3 A2 A1 A0

1 0 0 0

1 1 0 0

1 1 1 0

1 1 1 1

到这时，再进行异或判断时

$$a_3'=a_3\oplus a_0=1\oplus 1=0$$

所以接下去的序列为：

A3 A2 A1 A0

1 0 0 0

1 1 0 0

1 1 1 0

1 1 1 1

0 1 1 1

再继续进行：

1 0 1 1

0 1 0 1

1 0 1 0

1 1 0 1
0 1 1 0
0 0 1 1
1 0 0 1
0 1 0 0
0 0 1 0
0 0 0 1
1 0 0 0

我们可发现最后一行的序列和我们的初始状态序列是一样的.这就是一次移位寄存器的完整周期.可以看出这个序列的周期为15.在这一个周期里有1-15所有整数，而且并没有以固定的顺序出现，说明了有很好的随机性.

梅森素数Python的实现

```
#梅森旋转算法   -xlxw
#参考:mersenne twister from wikipedia

#import
from time import time
import numpy as np

#var
index = 624
MT = [0]*index
# MT[0] ->seed

def inter(t):
    return(0xFFFFFFFF & t) #取最后32位->t

def twister():
    global index
    for i in range(624):
        y = inter((MT[i] & 0x80000000) +(MT[(i + 1) % 624] & 0x7fffffff))
        MT[i] = MT[(i + 397) % 624] ^ y >> 1
        if y % 2 != 0:
            MT[i] = MT[i] ^ 0x9908b0df
    index = 0

def exnum():
```

```
    global index
    if index >= 624:
        twister()
    y = MT[index]
    y = y ^ y >> 11
    y = y ^ y << 7 & 2636928640
    y = y ^ y << 15 & 4022730752
    y = y ^ y >> 18
    index = index + 1
    return inter(y)

def mainset(seed):
    MT[0] = seed    #seed
    for i in range(1,624):
        MT[i] = inter(1812433253 * (MT[i - 1] ^ MT[i - 1] >> 30) + i)
    return exnum()

def main():
    br = input("请输入随机数产生的范围(用,隔开):")
    mi = eval(br.split(',')[0])
    ma = eval(br.split(',')[1])
    so = mainset(int(time())) / (2**32-1)
    rd = mi + int((ma-mi)*so)
    print("产生的随机整数为：",rd)
main()
```

运行程序截图：

```
========== RESTART: /Users/xulvxiaowei/Desktop/Mersenne twister.py ==========
请输入随机数产生的范围(用,隔开):100,1000
产生的随机整数为：  385
>>>
```

另一类伪随机数生成方法

这一类随机数还是伪随机数.但是和上面几种算法生成的伪随机数不同在于上面几种方法生成的伪随机数追求的都是分布均匀.而下面讲的是用于一些特殊情况下随机数的生成方法：

先让我们来看看以下几个例子：

- 当我们要对一个班的成绩进行模拟实验的时候，最贴近实际的是一种接近正态分布的成绩分布情况.并且数据要随机.
- 还有像模拟大学生体重情况，也和上面的一样遵循类正态分布.

- 还有像产品的良品率，降雨量等等都是

- 所以在现实中我们需要符合正态分布的随机数.而下面我们对其中的一种生成正态分布随机数的方法进行分享-BOX MULLER法

BOX-MULLER 的Python实现

```
#import
import numpy as np

def boxmuller():
    summa = 1
    size = 1
    x = np.random.uniform(size=size)
    y = np.random.uniform(size=size)
    z = np.sqrt(-2 * np.log(x)) * np.cos(2 * np.pi * y)
    q =  z * summa
    return q

print(boxmuller()[0])
```

这里对于BOX-MULLER不作详细的介绍

参考文献&拓展

- 随机数好坏的判定
  1.德联安全办的官方网址：
  BSI - Startseite
  2.NIST测试方法
  NIST.gov - Computer Security Division - Computer Security Resource Center
  3.Diehard测试程序
  http://www.stat.fsu.edu/pub/diehard/
  4.TestU01测试程序
  http://simul.iro.umontreal.ca/testu01/tu01.html
- 线性同余法
  [一种基于线性同余算法的伪随机数产生器]论文 发表于[纯粹数学与应用数学]21卷第三期

- 梅森旋转法
  线性反馈移位寄存器与梅森旋转算法 - ACdreamer - 博客频道 - CSDN.NET

https://en.wikipedia.org/wiki/Mersenne_Twister