Random Number Generation

1 Introduction

"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." This famous statement concerning the use of sequences of numbers generated using recursive formulas as random sequences is attributed to John von Neumann (1951). To generate a random number u from the Uniform (0,1) distribution, one must randomly select an integer $x \in (0, m)$ and set

$$u = \frac{x}{m}.$$

Machine-generated sequences of real numbers depend on recursions of the form:

$$x_j = f(x_{j-1}, x_{j-2}, \ldots, x_{j-k}),$$

or more expeditiously,

$$x_j = f(x_{j-1}).$$

Such sequences are obviously *deterministic*. We want to be able to generate sequences whose properties closely resemble those of random sequences in important ways. In addition, since such sequences are also necessarily *periodic*, we want such generators to have *long* periods, as well.

2 Linear Congruential Generators

`Linear Congruential Generators` are defined by the recursion

$$x_{i+1} \equiv (a\, x_i + c) \bmod m$$

where $x_0, x_1, \ldots$ is a sequence of integers and depends upon

$$\begin{aligned} x_0 &: \text{a seed} \\ a &: \text{a multiplier} \\ c &: \text{a shift} \\ m &: \text{a modulus} \end{aligned}$$

all of which are also integers "$\equiv$" defines an equivalence relation. Two numbers a and b are said to be *congruent modulo* m or $a \equiv b \bmod m$ where m is an integer, if their difference is exactly divisible by m. If $0 \leq a < m$ and $a \equiv b \bmod m$, then a is said to be a *residue* of b modulo m. a can be easily calculated using $a = b - \lfloor b/m \rfloor \times m$, where the *floor* function $\lfloor x \rfloor$ computes the greatest integer less than x.

The pseudo-random sequence $\{u_i\}$ is obtained by setting $u_i = x_i/m$ for i = 1, 2, ... If $c = 0$, the above defines a *multiplicative congruential generator* (MCG). Some theoretical discussions on MCG's are given in Ripley(1987), Gentle(1998, 2003) and Fishman(2006). A typical choice for m on binary computers is 2^k where k is the typical machine length

for integer type storage. In early generators this was thought to be a good choice because then the *modulo reduction* can be achieved simply via standard fixed-point overflow in the integer multiplication operation ($a\,x_i$). Recall that if fixed-point overflow is not trapped, the result of an integer arithmetic multiplication is modulo reduced by 2^k. This is because if the product results in a number greater than $2^{31}-1$, only the lower order 32 bits are retained to represent the result. However, since the first bit in fixed-point representation is the sign bit, instead of integers in the range $(0, 2^{32}-1]$, integers in the range $[-2^{31}+1, 2^{31}-1]$ are produced by such multiplications.

The multiplier "a" is chosen such that the generator will have maximal period. On computers using 32-bits to store integers, a popular early choices were $m = 2^{32}$ and $a = 65539$. Since the speeds of modern-day computers allow the modulo reduction to be done efficiently many implementations of random number generators use extended-precision floating point computations. Other values of a such as 1099087573, 2396548189, 2824527309, 3934873077, and 2304580733 have been shown to produce sequences that perform well on some tests.

Another choice for m is a large prime p, whence the period is $p-1$ if the multiplier is chosen to be its primitive root. Primes of the form 2^p-1, called *Mersenne primes*, are of particular interest. On 32-bit machines, popular choices are $m = 2^{31}-1$ and its primitive root $a = 7^5 = 16807$. The Unix `drand48` generator is defined on 48-bit unsigned integers with $a = 25214903917$, $c = 11$ and $m = 2^{48}$. It is implemented as the function `gsl_rng_uniform` in the GNU Scientific Library.

3 Linear Feedback Shift-Register Generators

Another type of random number generator employs the so-called *linear feedback shift-register* (FSR) technique, based on random sequences of bits. The i^{th} bit is computed from the previous p bits using a recurrence relation; thus for example: $b_i \equiv (a_1\,b_{i-1} + \cdots + a_d\,b_{i-p})\,mod\,2$, where $a_1, a_2, \ldots, a_d$ are 0 or 1. This recurrence relation can easily be performed by hardware circuitry using shift-registers. A sequence of bits is shifted to the right (or left) one place at a time and the leftmost (or rightmost) bit i.e., b_i is computed using an *XOR* operation with the coefficients $a_1, \ldots, a_p$. The pseudo-random number is formed by taking a bit-string of length L from this sequence (a process called *L-wise decimation*):

$$u_i = \left(b_{(i-1)L+1}\,b_{(i-1)L+2}\ldots b_{iL}\right)_2 / 2^L$$

for i = 1,2,...

The `random()` function on Unix (C or Fortran) implements an FSR with

$$x_i^{31} + x_i^3 + 1 \quad i.e. \quad p = 31 \quad q = 3$$

`random()` family of functions is a set of linear feedback shift register generators originally used in BSD Unix, and a version is available on GNU/Linux systems. The uniform generator in R is an example of a coupled generator: combination of 2 or more types of generators. The R routine `runif()` uses a combination of an MCG and FSR. There are several GFSR and mixed type generators available in the GNU Scientific Library (GSL) one of which is the generator `taus2` which is one of the fastest simulation quality generators available.

4 Portable Generators

The Fortran routine uni() from CMLIB is an example of `portable` random number generator. It uses an FSR and uses a table of machine constants to adapt the generator to different computers. The general idea is to make generators machine independent. That is, a generator should be able to produce the same sequence of numbers on different machines.

Wichmann and Hill (1982) uses 3 simple MCG's each with a prime for modulus and a primitive root for multiplier. If the results from each are added, the fractional part is $U(0,1)$. This follows from the fact that if X and U are independent and $U \sim U(0,1)$, then the fractional part of $X+U$ is $U(0,1)$ irrespective of the distribution of X. The generator is defined by the relations:

$$\begin{aligned} x_{i+1} &\equiv 171\, x_i \ mod\ 30269 \\ y_{i+1} &\equiv 172\, y_i \ mod\ 30307 \\ z_{i+1} &\equiv 170\, z_i \ mod\ 30323 \end{aligned}$$

and returns u_i where

$$\begin{aligned} u_i &= (x_{i+1}/30269 + y_{i+1}/30307 + z_{i+1}/30323) \\ u_i &= u_i \ mod\ 1.0 \end{aligned}$$

This generator requires three seeds x_0, y_0, and $z)0$ and has a period in the order of 10^{12}. Studies have shown that higher-order autocorrelations in sequences from the Wichman-Hill generator compare favorably with those from other good generators. A C implementation of the Wichmann-Hill generator is given in Figure 1.

5 Recent Innovations

5.1 Add-With-Carry and Multiply-With-Carry

Marsaglia and Zaman(1991) introduced the *add-with-carry* (AWC) generator of the form

$$x_i \equiv (x_{i-s} + x_{i-r} + c_i)\ mod\ m$$

where r and s are lags and $c_i = 0$ and $c_{i+1} = 0$ if $x_{i-s}+x_{i-r}+c_i < m$ and $c_{i+1} = 1$, otherwise. The c is the "carry." The *subtract-with-carry*(SWC) is given by

$$x_i \equiv (x_{i-s} - x_{i-r} - c_i)\ mod\ m.$$

Since there is no multiplication these can be implemented to be very fast and have large periods. Research has shown that the sequences resulting from these generators are similar to sequences obtained from linear congruential generators with very large prime moduli. However, other work show that the lattice structure of these may be very poor. Marsaglia also describes a *multiply-with-carry* (MWC) generator

$$x_i \equiv (ax_{i-} + c_i)\ mod\ m$$

and suggests $m = 2^{32}$ and an implementation in 64-bits. (This generator is referenced in Hofmann notes as Marsaglia-Multicarry-RNG and C code for using it provided there.)

```
/*****************************************************************/
/* Function  WICHHILL                                            */
/* This function generates and returns one Uniform(0,1) random   */
/* variate.  It uses the Wichmann-Hill algorithm.  Three seeds   */
/* addresses from 1-30,000 are passed to the subroutine and used */
/* to generate the variate.                                      */
/*****************************************************************/

#include <stdlib.h>
#include <math.h>
double fmod(double, double);
double wichhill(int *ix, int *iy, int *iz)
{
  double rand;

  *ix = (171**ix)%177 - (2*(*ix/177));
  *iy = (172**iy)%176 - (35*(*iy/176));
  *iz = (170**iz)%178 - (63*(*iz/178));

  if (*ix < 0)
    *ix += 30269;
  if (*iy < 0)
    *iy +=  30307;
  if (*iz < 0)
    *iz +=  30323;

  rand = fmod(((*ix)/30269.0f + (*iy)/30307.0f + (*iz)/30323.0f),1.0f);
  return rand;
}
```

Figure 1: A C Language Implementation of the Wichmann-Hill Generator

5.2 Mersenne twister

The GFSR generators have been modified by "twisting" the bit pattern in x_{i-q} by pre-multiplying the "vector" of bits in x_{i-q} by the $L \times L$ matrix C. The recurrence then becomes

$$x_i = x_{i-p} \oplus Cx_{i-q}.$$

The famous *Mersenne twister* is a twisted GFSR proposed by Matsumoto and Nishmura (1998), where the matrix C has special properties leading to the generator to have a period of $2^{19937} - 1$ and 623-variate Uniformity. They gave a C program named `mt19937`. The initialization procedure for this generator is rather complicated and described on the website `http://www.math.keio.ac.jp/~matumoto/emt.html` by the main author Matsumoto.

Several choices of basic unform generators are available in R. The R function `RNGkind()` may be used to select the generator. The default is the Mersenne twister. Wichmann-Hill and several Marsaglia generators, including the MWC, are available.