Pseudo-Random Number Generators

Part of the postgraduate journal club series, Mathematics, UQ

Vivien Challis

21 October 2008

1 Introduction

Random numbers are being used more and more as part of statistical simulations. There are two ways of generating random numbers:

1. By observing the outcomes of a truly random physical process.
2. By using a deterministic algorithm which generates numbers that *look* like random numbers.

The first option here is not easy. Some pieces of data of this kind exist, for example from the decay of radioactive elements, but generally generation is slow and you cannot get the quantity of random numbers you require for modern applications.

The second option here is what I'll talk about today. Algorithms of this type are called "Pseudo-Random Number Generators" (PRNGs). Many *bad* algorithms exist, in the sense that they generate numbers which can fairly readily be distinguished as having a pattern, and not being from a truly random process. I will discuss some of these issues and also outline the algorithms for some famous PRNGs, both *good* and *bad*. I'll focus on PRNGs for generating pseudo-random numbers from the uniform distrubution on the interval $[0, 1]$. It is fairly easy to transform these for a different distribution if required.

2 Some Properties of *Good* PRNGs

Good PRNGs pass a number of theoretical and statistical tests. In particular, a good generator should have a long period, meaning that there are many numbers in the sequence before it repeats. Various statistical tests have been proposed for PRNGs, many of these are based on goodness-of-fit of the points to the distribution expected if you were sampling from a uniform distribution. For example, the *equidistribution* test checks that there are the same number of points in intervals of the same length. This can be tested not just on the real line but also for tuples of numbers generated by the PRNG. Another test is the *run-test* which counts sequences of generated numbers which are increasing or decreasing, or remain above a certain value. The number and length of these sequences should exhibit certain behaviour.

I don't have time to talk about these tests in detail, but it is safe to say that there are many. There are sets of tests which are used on a new PRNG before it gains credibility. For example the *DIEHARD* test suite (see references).

No matter how many statistical tests a PRNG passes, the numbers generated are not random. Thus no PRNG is 'perfect', and it could be the case that a PRNG which is generally accepted to be

good is a poor choice for your particular problem. In such a case the model/simulation picks out a weakness in the generator, or the dynamics of the simulation and the number generator somehow interact to give false results. Thus the golden rule is:

ALWAYS run your simluation with more than one pseudo-random number generator.

Many PRNGs which are good for statistical purposes (e.g. Monte Carlo simulations) are not good for cryptographic purposes. For cryptographic purposes the generated numbers need to be unpredictable: from any number of generated numbers it should be impossible to gain information on how they were generated. For most generators this is not the case — here I'll focus on PRNGs for statistical rather than cryptographic purposes.

3 Linear Congruential Generators

Linear congruential generators are a very common, simple type of PRNG. A sequence of uniform pseudo-random numbers $\{u_i\}$ are calculated from:

$$x_{i+1} = ax_i + c \pmod{m} \quad (1)$$

$$u_i = \frac{x_i}{m}. \quad (2)$$

The sequence is started with some integer x_0, called the *seed*. When $c > 0$ a generator of this type is called a *mixed linear congruential generator*, and it generates numbers on the interval $[0, \frac{m-1}{m}]$. For $c = 0$ the generator is called a *multiplicative linear congruential generator*, and it generates numbers on the interval $[\frac{1}{m}, \frac{m-1}{m}]$.

Since these generators have such a simple form, they are easy to analyse number-theoretically and theorems exist which detail how to choose the parameters a, c and m to give a generator with a long period. Note that the longest possible period is $m - 1$ when $c = 0$ or m when $c > 0$.

3.1 RANDU

`RANDU` is a famously poor PRNG of this type implemented by IBM in 1968. It has parameters $a = 65539$, $c = 0$, $m = 2^{31}$. These parameters were chosen to make the generator easy to implement and fast. Since computers use binary representations of integers, operations with powers of 2 are particularly simple. In particular, arithmetic modulo 2^{31} can be done quickly, and $a = 65539 = 2^{16} + 3$ so multiplication by a can be done with a shift and an addition.

These choices made the generator computationally efficient but led to the very undesirable property

$$x_{k+2} = 6x_{k+1} - 9x_k \text{ for all } k. \quad (3)$$

The manifestation of this is demonstrated in Fig. 1. The following is the Matlab code used to generate the figure, which shows how easy it is to implement one of these generators:

```
% Define parameters
a=65539;
c=0;
x0=1;
m=2^31;
```

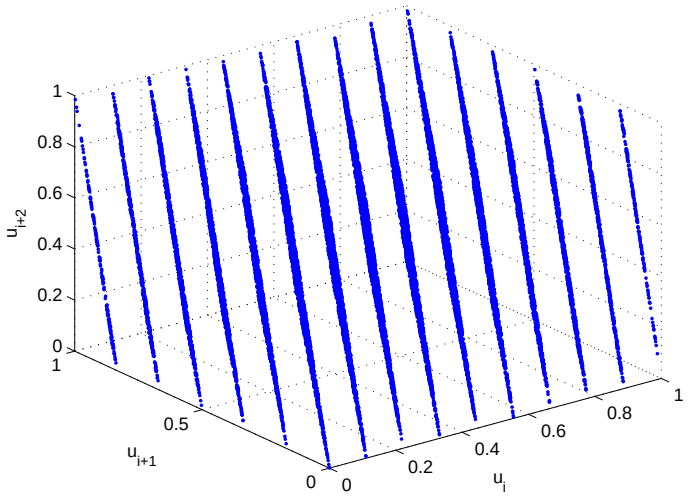

Figure 1: 3-tuples of 20,000 numbers generated by the `RANDU` PRNG.

```
% Calculate sequence using recursion relation
xn=zeros(20000,1);
for i=1:20000
    xn(i)=mod(a*x0+c,m);
    x0=xn(i);
end

% Divide by m to give real numbers between 0 and 1
un=xn/m;

% Plot 3-tuples of the u_i in 3D space
plot3(un(1:end-2),un(2:end-1),un(3:end),'b.');
xlabel('u_i'); ylabel('u_{i+1}'); zlabel('u_{i+2}'); grid('on');
```

So even though the generated random numbers look fine on the real line or as 2-tuples, they appear *very* non-random when you plot them as 3-tuples!

3.2 Quality of Linear Congruential Generators

All linear congruential generators suffer from the problem that all the generated pseudo-random numbers lie on a lattice. Even if this is not as apparent as for the `RANDU` case above the lattice will still be present. Despite this, these generators have been and still are widely used. Also, some generators use a more general form of the recursion where more than just the previous value in the sequence is used — these are called *multiple recursive congruential generators*.

Linear congruential generators are not suitable for cryptographic purposes. From enough numbers in the sequence it is possible to predict how the sequence continues, due to the linear recursion relation. This is the problem with all linear generators, and leads to the use of non-linear generators when cryptographic security is required. These generators are slower and more difficult to implement. However, since fewer numbers will be generated than are needed for statistical purposes, this is of little consequence.

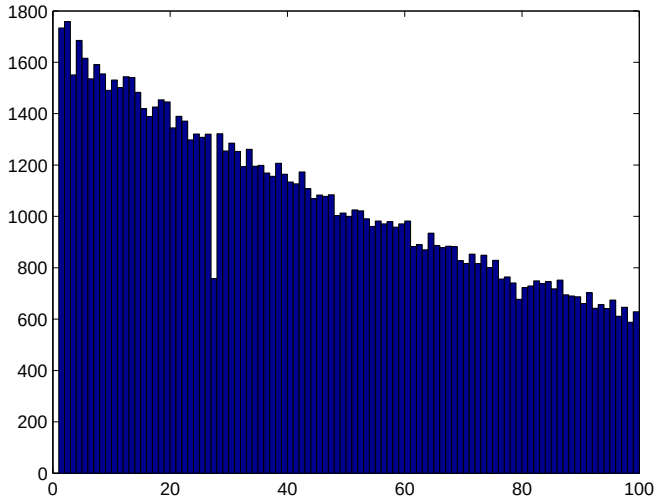

Figure 2: Histogram of run lengths for runs which stay above 0.01, for Matlab's implementation of a subtract-with-borrow algorithm.

3.3 Matlab's PRNG Choice

For many years the default Matlab PRNG was a linear congruential generator, with parameters $a = 7^5 = 16807$, $c = 0$, $m = 2^{31} - 1 = 2,147,483,647$. This generator has a period of $m - 1$, and each number of the form $\frac{k}{m}$ between $\frac{1}{m}$ and $\frac{m-1}{m}$ is generated as part of the sequence. A few years ago this period of just over 2 billion was considered adequate for any statistical simulations because a computer could not exhaust this number easily. This generator has a much less obvious lattice structure than `RANDU`, but nevertheless it is still there.

Today a standard PC could exhaust this a sequence of length 2 billion in about a minute. In 1995 the default in Matlab was changed to a completely different algorithm called subtract-with-borrow (I won't talk about the details, see references), which generates floating point numbers using bit-wise arithmetic (instead of actually generating integers first). This generator is still the default in the version of Matlab on my office computer, and has a period of $2^{1492} \approx 10^{450}$. It can theoretically generate all floating point numbers between 2^{-53} and $1 - 2^{-53}$.

The subtract-with-borrow algorithm fails a test called the *run-test*, which counts the length of sequences in the generated pseudo-random numbers which remain above some small value. For true random numbers generated from an uniform distribution the number of runs of a certain length should decrease as the length of the run increases. This behaviour is observed, but as seen in Fig. 2, runs of length 27 are under-represented. This is a very subtle fault, caused by the parameters of the generation algorithm. The following is the code used to generate the figure:

```
% set seed, 'state' here means that we are using the subtract-with-borrow
% algorithm (Matlab's default between 1995 and 2007).
rand('state',0)

% generate random numbers, set tolerance
x = rand(1,2^24);
delta = .01;

% generate run lengths, histogram data & plot it
```

```
k = diff(find(x<delta));
t = 1:99;
c = histc(k,t);
bar(t,c,'histc')
```

The default in Matlab changed with version 7.4 in 2007 to the *Mersenne Twister*. This is currently the PRNG of choice for most statistical applications. I'll discuss this next but will just note that it has a period of $2^{19937} - 1 \approx 10^{6000}$.

4 Mersenne Twister

The Mersenne Twister (MT) is a *Twisted Linear Feedbacked Shift Register* PRNG. Before I describe the generation algorithm, here are some properties of the MT:

Computations are all bit-wise: 32-bits are used to represent each random number and all of the computations are on these bits. So, all computations are essentially done modulo 2.

Not suitable for cryptographic purposes: Since the generator is linear, it is not suitable for cryptographic purposes unless the output is converted via a secure hasing algorithm.

Period length of $2^{19937} - 1$: As I said above, this is about 10^{6000}. The number $2^{19937} - 1$ is a Mersenne prime, this is why the algorithm has the word "Mersenne" in the name.

The generated numbers are 623-dimensionally equidistributed: This means that all k-tuples up to $k = 623$ dimensions generated by the MT satisfy the equidistribution property you would expect of uniform random numbers. Note that this property is actually satisfied for all 32-bits of the generated numbers. This is an important improvement over previous generators which tended to have poor equidistribution properties for some of the bits.

The recursion relation for the MT is (I got this direct from the original paper):

$$\mathbf{x}_{k+n} = \mathbf{x}_{k+m} \oplus (\mathbf{x}_k^u | \mathbf{x}_{k+1}^l) A, \tag{4}$$

where the notation/parameters are:

- $\mathbf{x}_k$ is the k^{th} value in the sequence, and is a "word" of length w (i.e., a w-bit number). For the MT $w = 32$.
- n defines the degree of the recursion, and m the middle term used in the recursion. For the MT, $n = 624$, $m = 397$. This means that 624 words (i.e., 32-bit numbers) are needed to seed the sequence and need to be stored in memory.
- The notation $\oplus$ represents bit-wise addition of the words, which is modulo 2.
- The notation $\mathbf{x}_k^u$ is the upper $w - r$ bits of $\mathbf{x}_k$, while $\mathbf{x}_{k+1}^l$ is the lower r bits of $\mathbf{x}_{k+1}$. The notation $|$ is just a concatenation of these two partial words to form a w-bit word. The parameter r is 31 in MT.
- A is a matrix, whose form is chosen so that multiplcation by A is very fast.

It is all the bit-wise operations which make generation using this generator *so* fast.

There is one additional aspect of the algorithm which is called *tempering*. To improve the equidistribution properties, each word is multiplied on the right by an invertible $w \times w$ matrix T. This then gives the final number output by the MT.

One final note, for the MT the 624 words needed to seed the generator (which have no real constraints except that they should be non-zero) are usually generated using some other pseudo-random number gerator (the standard is a linear congruential generator).

5 Possibilities for future discussions

Related ideas that I think might be interesting:

1. Cryptographically secure (i.e., non-linear) pseudo-random number generators. I get the impression these are even harder to design than PRNGs for statistical purposes.
2. Low discrepancy sequences and quasi-Monte Carlo as an alternative to pseudo-random number generators and Monte Carlo simulations. Low discrepancy sequences aren't designed to look like random sequences, instead they are designed to evenly cover the area of interest. When you generate random numbers you see random clumping of the numbers, low discrepancy sequences don't have the clumping. This makes them good for things like numerical integration, and the quasi-Monte Carlo simulations can have improved convergence properties over standard Monte Carlo.

6 Useful references

A book written by a Matlab developer, which has a really nice, introductory and interactive chapter on PRNGs (as a note, the book seems to be really useful for anyone wanting to teach students how to use Matlab on some fairly simple, but interesting problems):
Moler, C.B., "Numerical Computing with MATLAB," SIAM, (2004).
Available online at http://www.mathworks.com/moler

A following paper was written for users of PRNGs as opposed to designers of PRNGs. It was written before the Mersenne Twister was published. The author also keeps a really useful website on random number generators, which inclues a lot more references: http://random.mat.sbg.ac.at/
Hellekalek, P.: Good random number generators are (not so) easy to find. Mathematics and Computers in Simulation, 46: 485–505, 1998.

The original Mersenne Twister paper:
Matsumoto, M. and Nishimura, T. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudorandom Number Generator," ACM Transactions on Modeling and Computer Simulation, (1998), 8(1):3-30.

The Marsaglia Random Number CDROM including the *Diehard Battery of Tests of Randomness*, downloadable at:
http://www.stat.fsu.edu/pub/diehard/