Chapter 3

Pseudo-random numbers generators

3.1 Basics of pseudo-random numbers generators

Most Monte Carlo simulations do not use true randomness. It is not so easy to generate truly random numbers. Instead, pseudo-random numbers are usually used. The goal of this chapter is to provide a basic understanding of how pseudo-random number generators work, provide a few examples and study how one can empirically test such generators. The goal here is not to learn how to write your own random number generator. I do not recommend writing your own generator. I also do not recommend blindly using whatever generator comes in the software package your are using. From now on we will refer to pseudo random number generators simply as random number generators (RNG).

The typical structure of a random number generator is as follows. There is a finite set S of states, and a function $f : S \to S$. There is an output space U, and an output function $g : S \to U$. We will always take the output space to be $(0, 1)$. The generator is given an initial value S_0 for the state, called the seed. The seed is typically provided by the user. Then a sequence of random numbers is generated by defining

$$\begin{aligned} S_n &= f(S_{n-1}), \quad n = 1, 2, 3, \cdots \\ U_n &= g(S_n) \end{aligned} \tag{3.1}$$

Note that there is nothing random here. If we generate a sequence of numbers with this procedure and then generate another sequence using the same seed, the two sequences will be identical. Since the state space is finite, S_n must eventually return to a state it was in some time earlier. The smallest integer p such that for some state the function f returns to that state after p iterations is called the period of the generator. Obviously, longer periods are better than short periods. But a long period by itself certainly does insure a good random

number generator.

We list some of the properties we want in our generator. Of couse we want it to produce an i.i.d. sequence with the uniform distribution on $(0, 1)$. That by itself is a rather abstract mathematical requirement; the first two properties below make it more practical.

1. **Pass empirical statistical tests** These are tests where you generate a long sequence of random numbers and then perform various statistical tests to test the hypothesis that the numbers are uniformly distributed on $[0, 1]$ and are independent.

2. **Mathematical basis** There is a lot of mathematical theory behind these random number generators (at least some of them) including properties that should produce a good random number generator. Obviously, we want a large period, but there are more subtle issues.

3. **Fast (and not a lot of memory)** Most Monte Carlo simulations require a huge number of random numbers. You may want to generate a large number of samples, and the generation of each sample often involves calling the random number generator many times. So the RNG needs to be fast. With most generators memory is not really an issue, although some can take up a lot of cache.

4. **Multiple streams** It is easy to use parallel computation for Monte Carlo. But this requires running multiple copies of your random number generator. So you need to be sure that these different streams are independent of each other.

5. **Practical concerns** Ease of installation and ease of seeding. Some random number generators are quite short and only take a few lines of code. Others are considerably more complicated. Some like the Mersenne twister require a rather large seed. Many of the better generators use 64 bit integers. So be sure you are working on a 64 bit system and type your variables appropriately.

6. **Reproducibility** For debugging and testing purposes you want to be able to generate the same stream of random numbers repeatedly. For any random number generator of the form we are considering this is easy - just start with the same seed.

3.2 Some examples

We do not attempt to give all the different types of generators. We discuss a few different types with some specific examples.

3.2.1 Linear congruential generators

The state space is $\{0, 1, 2, \cdots, m-1\}$ where m is a positive integer.

$$f(X) = aX + c \bmod m \tag{3.2}$$

where mod m means we do the arithmetic mod m. The constants a and c are integers and there is no loss of generality to take them in $\{0, \cdots, m-1\}$. For the output function we can take

$$g(X) = \frac{X}{m} \tag{3.3}$$

The quality of this generator depends on the choice of the constants a and c. There is a lot of mathematics behind how these constants should be chosen which we will not go into.

Example: Lewis, Goodman, and Miller, proposed the choice $a = 7^5 = 16807, c = 0$ and and $m = 2^{31} - 1 = 2147483647$. It has period $2^{31} - 2$.

drand48(): This is a linear congrutial generator which uses $m = 48$,
$a = 5DEECE66D_{16} = 273673163155_8$, $c = B_{16} = 13_8$. It is part of the standard C library. It is not recommended.

A bunch of examples of (not necessarily good) LCG is at :

random.mat.sbg.ac.at/results/karl/server/node4.html

3.2.2 Multiple-recursive generators

We take the state space to be $\{0, 1, 2, \cdots, m-1\}^k$ and define the function f by

$$X_n = (a_1 X_{n-1} + a_2 X_{n-2} + \cdots a_k X_{n-k}) \bmod m \tag{3.4}$$

The notation is inconsistent here. Before X_n was the state at time n. Now the state at time n is $(X_n, X_{n-1}, \cdots, X_{n-k+1})$. Then the output is $U_n = X_n/m$. With suitable choices of m and a_i we can get a period of $m^k - 1$.

There are more complicated linear generators, e.g., matrix multiplicative recursive generators, generalized feedback shift registers, and twisted generalized feedback shift registers, but we do not discuss them. A widely used example of the latter is the Mersenne twister, MT19937, invented by Matsumoto and Nishimura. It is implemented in MATLAB and SPSS. It has a very large period, $2^{19937} - 1$. Code for it can be found at

http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/emt.html

3.2.3 Combining generators

A common trick in designing random number generators is to combine several not especially good random number generator. An example is the Wichman-Hill generator which combines three linear congruential generators. The state space is
$\{0, 1, 2 \cdots, m_1 - 1\} \times \{0, 1, 2 \cdots, m_2 - 1\} \times \{0, 1, 2 \cdots, m_3 - 1\}$. We denote the state at step n by (X_n, Y_n, Z_n). Then the generator is

$$\begin{aligned} X_n &= 171 X_{n-1} \bmod m_1 \\ Y_n &= 172 Y_{n-1} \bmod m_2 \\ Z_n &= 170 Z_{n-1} \bmod m_3 \end{aligned} \tag{3.5}$$

with $m_1 = 30269, m_2 = 30307, m_3 = 30323$. The output function is

$$U_n = \frac{X_n}{m_1} + \frac{Y_n}{m_2} + \frac{Z_n}{m_3} \bmod 1 \tag{3.6}$$

The period is approximately 7^{12}, large but not large enough for large scale Monte Carlo.

We can also combine multiple-recursive generators. L'Ecuyer's MRG32k3a is an example which employs two MRGs of order 3:

$$\begin{aligned} X_n &= (1403580 X_{n-2} - 810728 X_{n-3}) \bmod m_1 \\ Y_n &= (527612 Y_{n-1} - 1370589 Y_{n-3}) \bmod m_2 \end{aligned} \tag{3.7}$$

with $m_1 = 2^{32} - 209$, $m_2 = 2^{32} - 22853$. The output function is

$$U_t = \begin{cases} \frac{X_n - Y_n + m_1}{m_1 + 1}, & \text{if } X_n \le Y_n, \\ \frac{X_n - Y_n}{m_1 + 1}, & \text{if } X_n > Y_n, \end{cases} \tag{3.8}$$

The period is approximately 3×10^{57}. This generator is implemented in MATLAB.

Some very simple and fast RNG Marsagalia's KISS generators. He posted some of these in the forum:

http://www.thecodingforums.com/threads/64-bit-kiss-rngs.673657/

Some more examples like this can be found in the article by David Jones (reference at end of section).

Stop - Mon, 1/25

3.3 A little statistics

We briefly explain two statistical tests - χ^2 goodness of fit and the Kolmogorov-Smirnov goodness of fit tests. These are worth discussing not just because there are used in testing random number generators but also because they are used to analyze what comes out of our Monte Carlo simulation.

The χ^2 distribution: Let $Z_1, Z_2, \cdots Z_k$ be independent random variables, each of which has a standard normal distribution. Let

$$X = \sum_{i=1}^{k} Z_i^2 \tag{3.9}$$

The distribution of X is called the χ^2 distribution. Obviously, it depends on the single integer parameter k. k is called the "degrees of freedom."

The multinomial distribution: This is a discrete distribution. Fix an integer k and probabilities $p_1, p_2, \cdots p_k$ which sum to one. Let $X_1, X_2, \cdots, X_n$ be independent random variables with values in $\{1, 2, \cdots, k\}$ and $P(X_j = l) = p_l$. Let O_j be the number of $X_1, X_2, \cdots, X_n$ that equal j. (O stands for observed, this is the observed number of times j occurs.) There is an explicit formula for the joint distribution of $O_1, O_2, \cdots, O_k$, but we will not need it. We will refer to the parameter n as the number of trials. Note that the mean of O_j is $p_j n$.

Suppose we have random variables $O_1, O_2, \cdots, O_k$ and we want to test the hypothesis that they have the multinormal distibution. We let $e_j = np_j$, the mean of O_j, and consider the statistic

$$V = \sum_{j=1}^{k} \frac{(O_j - e_j)^2}{e_j} \tag{3.10}$$

Theorem 1 *If $O_1, O_2, \cdots, O_k$ has a multinomial distribution then as $n \to \infty$ the distribution of V defined above converges to the χ^2 distribution with $k-1$ degrees of freedom.*

Any software package with statistics (or any decent calculator for that matter) can compute the χ^2 distribution. As a general rule of thumb, the number of observations in each "bin" or "class" should be at least 5.

The Kolmogorov-Smirnov goodness of fit test is a test that a random variable has a particular continuous distribution. We start with an easy fact:

Theorem 2 *Let X be a random variable with a continuous distribution function $F(x)$. Let $U = F(X)$. Then the random variable U is uniformly distributed on $[0, 1]$.*

Proof: We assume that the range of X is an interval (possibly half-infinite or infinite) and F is strictly increasing on the range of X. So we can define an inverse F^{-1} that maps $[0,1]$ to the range of X. For $0 \leq t \leq 1$,

$$P(U \leq t) = P(F(X) \leq t) = P(X \leq F^{-1}(t)) = F(F^{-1}(t)) = t \tag{3.11}$$

QED.

Now fix a distribution with continuous distribution function (CDF) $F(x)$. (This test does not apply to discrete distributions.) We have a random variable and we want to test the null hypothesis that the CDF of X is $F(x)$. Let $X_1, X_2, \cdots, X_n$ be our sample. Let $X_{(1)}, X_{(2)}, \cdots, X_{(n)}$ be the sample arranged in increasing order. (The so-called order statistics.) And let $U_{(i)} = F(X_{(i)})$. Note that $U_{(1)}, U_{(2)}, \cdots, U_{(n)}$ are just $U_1, U_2, \cdots, U_n$ arranged in increasing order where $U_i = F(X_i)$. Under the null hypothesis we expect the $U_{(i)}$ to be roughly uniformly distributed on the interval $[0,1]$. In particular $U_{(i)}$ should be roughly close to $(i - 1/2)/n$. (The difference should be on the order of $1/\sqrt{n}$.)

$$\begin{aligned} D &= \frac{1}{2n} + \max_{1 \leq i \leq n} |F(X_{(i)}) - \frac{i - \frac{1}{2}}{n}| && (3.12) \\ &= \frac{1}{2n} + \max_{1 \leq i \leq n} |U_{(i)} - \frac{i - \frac{1}{2}}{n}| && (3.13) \end{aligned}$$

Note that if the null hypothesis is true, then the U_i are uniform on $[0,1]$ and so the distribution of D does not depend on $F(x)$. It only depends on n. There is a formula for the distribution of D involving an infinite series. More important, software will happily compute p-values for this statistic for you.

3.4 Tests for pseudo-random numbers generators

In the following we let U_n be the sequence of random numbers from our generator. We want to test the null hypothesis that they are independent and each U_n is uniformly distributed on $[0,1]$ In the following the null hypothesis will always mean the hypothesis that the U_n are i.i.d. and uniform on $[0,1]$.

3.4.1 Equidistribution tests

One way to test that the U_n are uniformly distributed is to just use the Kolmogorov-Smirnov test. Here $F(x) = x$. Note that this does not test the independence of the U_n at all.

Another way to test the uniform distribution is the following. Fix a positive integer m. Let

$$Y_n = \lfloor mU_n \rfloor \tag{3.14}$$

where $\lfloor x \rfloor$ is the floor function which just rounds x down to the nearest integer. So Y_n takes on the values $0, 1, 2, \cdots, m-1$. Under the null hypothesis the Y_n are independent and uniformly distributed on $\{0, 1, \cdots, m-1\}$. So we can do a χ^2 test using V above, where O_j is the number of times Y_i equals $j+1$. Note that this only tests that the U_n are uniformly distributed on $[0, 1]$.

To test the independence (as well as the uniformity) we can do the following. Fix another integer d. Use the random number generator to generate random d-tuples of numbers: $(U^1, U^2, \cdots, U^d)$. Generate n such d-tuples, call them $(U_i^1, U_i^2, \cdots, U_i^d)$ where $i = 1, 2, \cdots, n$. (So we call the RNG nd times.)

$$Y_i^j = \left\lfloor mU_i^j \right\rfloor \tag{3.15}$$

Then the $(Y_i^1, Y_i^2, \cdots, Y_i^d)$ should be uniformly distributed over $\{0, 1, 2, \cdots, m-1\}^d$. We can test this with χ^2 test. This tests the independence to some extent, but it only tests if d consecutive calls to the RNG are independent. Note that the number of cells in the χ^2 test is m^d, so d cannot be too large.

3.4.2 Gap tests

Fix an interval $(\alpha, \beta) \subset (0, 1)$. Let T_n be the times when the random number is in (α, β). Let Z_n be the gaps between these times, i.e., $Z_n = T_n - T_{n-1}$. (We take $T_0 = 0$.) If the random numbers are i.i.d. and uniform, then the Z_n should be i.i.d. with a geometric distribution with parameter $p = \beta - \alpha$. We can test this with a χ^2 test. Fix an integer r and take the classes to be $Z = 0, Z = 1, \cdots, Z = r-1$ and $Z \geq r$.

A special case is $(\alpha, \beta) = (0, 1/2)$ which is called runs above the mean. With $(\alpha, \beta) = (1/2, 1)$ it is called runs below the mean.

3.4.3 Permutation tests

Use the RNG to generate random d-tuples of numbers: $(U^1, U^2, \cdots, U^d)$. For each d-tuple let π be the permutation which puts them in increasing order, i.e., $U^{\pi(1)} < U^{\pi(2)} < \cdots < U^{\pi(d)}$. Under the null hypothesis that the random numbers are i.i.d. uniform, the permutations will have the uniform distribution on the set of $d!$ permutations. We can test this with a χ^2 test.

3.4.4 Rank of binary matrix test

This one looks pretty crazy, but it has been useful in showing some RNG's that pass some of the simpler tests are not good.

Convert the i.i.d. sequence U_n to an i.i.d. sequence B_n which only takes on the values 0 and 1 with probability 1/2. For example let B_n be 0 if $U_n < 1/2$, $B_n = 1$ if $U_n \geq 1/2$. Fix integers r and c with $r \leq c$. Group the B_n into groups of size rc and use each group to form an $r \times c$ matrix. Then compute the rank of this matrix using arithmetic mod 2. There is an explicit formula for the distribution of this rank under the null hypothesis. (See Kroese review article for the formula.) We can then do a χ^2 test. For example, take $r = c = 32$. The rank can be at most 32. Note that it is very unlikely to get a rank a lot less than 32. So you don't want to take classes running over all possible values of the rank. With $r = c = 32$ we could take the classes to be $R \leq 30$,$R = 31$, and $R = 32$, where R is the rank.

3.4.5 Two-stage or second order tests

Consider one of the above tests (or many others). Let T be the test statistic. We assume that the distribution of T under the null hypothesis is known. The simple test is to generate a sample, compute the value of T and then compute its p-value. (Explain what a p-value is). A small p value, e.g., less than 5% means we got a value of T that is pretty unlikely if the null hypothesis is true. So we reject the null hypothesis (and so conclude our RNG is suspect). Now suppose we repeat this 50 times. So we get 50 values $T_1, T_2, \cdots T_{50}$ of the test statistic. We then get 50 p-values. Even if the null hypothesis is true, we expect to get a few p-values less than 5%. So we shouldn't reject the null hypothesis if we do. But we can use our 50 values of T to carry out a second order test. If the null hypothesis is true, then $T_1, T_2, \cdots T_{50}$ should be a sample from the known distribution of the test statistic. We can test this with the KS statistic and compute a single p-value which tests if the T_i do follow the known distribution.

3.4.6 Test suites

We end our discussion of testing with a few of the well known packages for testing a RNG.

George Marsaglia developed a suite of 15 tests which he called the diehard suite. Marsaglia passed away in 2011, but google will happily find the diehard suite. Robert Brown at Duke expanded the suite and named it dieharder:

http://www.phy.duke.edu/ rgb/General/dieharder.php

A comprehensive set of tests is the TestU01 software library by LEcuyer and Simard:

http://www.iro.umontreal.ca/ lecuyer/myftp/papers/testu01.pdf

Another important use of random numbers is in cryptography. What makes a good RNG for this (unpredictability) is not exactly the same as what make a good one for Monte Carlo. In particular, speed is not so important for some crytography applications. So in looking at test suites you should pay attention to what they are testing for.

3.5 Seeding the random number generator

How you seed the random number generator is important.

For parallel computation we certainly don't want to use the same seed.

MT and bad seeds

The seed of a RNG is a state for the RNG and so is typically much larger than a single random number. For the MT the seed requires ?? So it is nice to have an automated way to generate seeds.

A cheap way to generate a seed is to use some output from the operating system and then run it through a hash function. For example, on a unix system

ps -aux | md5sum

will produce a somewhat random 128 bit number. (The output is in hexadecimal, so it is 32 characters long.)

A more sophisticated approach on unix system is to use /dev/urandom. This uses "noise" in the computer to generate a random number. Note that it tests how much entropy has accumulated since it was last called and waits if there is not enough. So this is too slow to be used for your primary RNG. On a unix system

od -vAn -N4 -tu4 < /dev/urandom

will give you some sort of random integer. (Need to find out just what this returns.)

3.6 Practical advice

- Don't use built in RNG unless you know what it is and how it has been tested.

- Design your code so that it is easy to change the RNG you use.
- Use two different RNG and compare your results. This is not a waste of time as you can always combine the two sets of data. Or if you do not want to go to the trouble of a second generator, try things like using only every fifth random number from the generator.
- Don't use too many random numbers from your generator compared to your period. There are many ad hoc rules of thumb : use at most $P/1000$, $\sqrt{P}/200$ or even $P^{1/3}$.

3.7 References

I have followed the chapter on random number generation in Kroese's review article and also taken some from "Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications" by David Jones. It is a nice practical discussion of picking a RNG. It can be found at

http://www0.cs.ucl.ac.uk/staff/d.jones/GoodPracticeRNG.pdf

For an interesting example of how supposedly good RNG lead to wrong results, see

Monte Carlo simulations: Hidden errors from "good" random number generators Alan M. Ferrenberg, D. P. Landau, and Y. Joanna Wong, Phys. Rev. Lett. 69, 3382 (1992).