A few notes about (pseudo-)random number generators

1 The basic idea

All of the Monte Carlo sampling and integration schemes that we have discussed in class have depended on having a reliable source of iid uniform random deviates

$$U_1, U_2, \ldots, \stackrel{iid}{\sim} Unif[0,1]$$

Modern simulation methods, and especially MCMC methods, generally require the use of thousands, billions, or more, of these numbers in a single simulation.

Pseudo-random number generators (RNG's for short) are computer algorithms for producing sequences $U_1, U_2, \ldots$ that *pass many statistical tests* of randomness and uniformity.

At some fairly broad level, all RNG's are obtained by iterating some deterministic function, e.g. perhaps there is some function $g_{RNG}(u)$ such that

$$U_{i+1} = g_{RNG}(U_i)$$

or perhaps the iteration works on blocks of U_i's at a time. Thus, because there are only finitely many numbers U_i that a digital computer can represent, eventually some future U_{i^*} will have to equal a past U_i, and when that happens the RNG enters a "loop": it will from that point on forever produce the sequence of numbers leading from U_i to U_{i^*}. The length of the loop, $i^* - i$, is the *period* of the RNG.

Iterative RNG's have to be given an initial value U_0 and then they can generate the rest of a sequence from that. This initial value is called the *seed* of the RNG. Clearly, the same RNG will produce different sequences $U_1, U_2, \ldots$, starting from different seeds U_0. It is preferable to work with RNG's that produce approximately the same period, no matter what the seed.

(In Splus and R, the function that sets the seed of the RNG is `set.seed()`. The value of the seed is maintained in the variable `.Random.seed` in both packages; if you save the value `.Random.seed` in another variable when you stop an RNG in R or Splus, you can restart it again "at the same place" by resetting `.Random.seed` to that saved value. The form of the seed is different depending on the RNG in use. There is only one standard RNG in Splus, but in R there are several, chosen with the `RNGkind()` function. Similarly, in the IMSL C statistics subroutine library, you choose which RNG to use with the `imsls_random_option` function, and select the seed with the `imsls_random_seed_set` function. Some details on the RNG's provided by these packages will be given below. The function `imsls_random_seed_get` can be used to get the value of the RNG seed when you stop simulation in IMSL, so that it can be restarted later "in the same place", just as in Splus or R.)

Finally, most RNG's do not naturally produce reals (or floats or doubles) between 0 and 1. Instead, it is usually more natural (and much faster) to generate integers between 0 and some large number, say M_{RNG} (note that the period of the RNG must be less than or equal to M_{RNG} as well!). Then these integers can be divided by M_{RNG} to produce $Unif[0,1]$ deviates (Monahan, pp. 315ff discusses fancier ways to get from integers to $Unif[0,1]$ deviates, as well).

In what follows we will use U_i for continuous $Unif[0,1]$ and V_i for discrete $Unif\{0,1,2,\ldots, M_{RNG}\}$ RNG's.

2 What we want from a random number generator

Of course, we want a RNG with a long period. Apart from the modern need for truckloads and truckloads of pseudorandom numbers, statistical tests of randomness tend to fail as the period of the RNG is exhausted, because if you know the maximum value M_{RNG}, then the last few numbers in the period are very predictable from the list of numbers visited so far.

We also want the RNG to pass some set of statistical and geometric tests of randomness and uniformity. These tests typically fall into several categories:

- *Uniformity.* For example, is it true that for every subinterval $(a, b) \subseteq (0, 1)$,

  $$\frac{\#\{i \le n : \, U_i \in (a, b)\}}{n} \approx b - a$$

  as n grows? Etc.

- *Uncorrelatedness.* Is the autocorrelation function $\texttt{acf}(s) = Corr(U_i, U_{i+s})$ reasonably close to zero for all s? Is it possible to predict future "draws" from the record of draws so far? Etc.

- *Multidimensional behavior.* If you plot successive pairs (U_i, U_{i+1}), triples (U_i, U_{i+1}, U_{i+2}), etc., many RNG's exhibit strong lattice structure and other spatial patterns. These patterns may (a) undermine the justification for treating the U_i's as truly random, and (b) limit the usefulness of the U_i in Monte Carlo integration, since local detail between lattice points may not be be picked up by the RNG.

Despite these general approaches to "vetting" RNG's, the true test of an RNG is that, if you are working on an important problem, then you should choose an RNG that behaves appropriately in that problem. Several of the RNG's outlined below have lots of experience behind them, as general-purpose generators in Splus, R or IMSL. Except for extremely complicated or high-dimensional problems, these should work well in many Monte Carlo situations, especially based on iid sampling. Testing RNG's for use in MCMC sampling, for example, is still in its infancy however, and so some extra precautions—especially in large MCMC problems—are advisable. These precautions might include

(a) Estimating the number of U_i's you will need for your simulation and making sure the period of your RNG can handle that many;

(b) Simulating a system related to your "real" problem, where you can analytically get an answer, and verify that the simulation answer and the analytic answer for the related system are the same;

etc.

3 Some Common Types of Generators, and Variations

3.1 Linear congruential generators (LCG's)

This is a basic type that is conceptually easy, and usually easy to compute. It is an iterative scheme based on modular arithmetic:

$$V_i = (aV_{i-1} + c) \bmod m,$$

that is, compute $aV_{i-1} + c$ and then V_i is its remainder after division by m.

For example, the function `imsls_f_random_uniform` in IMSL for C generates Unif[0,1] random numbers starting from the LCG

$$V_i = aV_{i-1} \bmod (2^{31} - 1),$$

where a can be chosen (using `imsls_random_option`) to be either 16807, 397204094, or 950706376. All three have (maximal) periods of $2^{31} - 2$, and have a long and successful history of use. The larger a, the slower to compute the next U_i; to offset this, we note that the generator with $a = 950706376$ passes the a statistically sound set of tests for RNG's (cf. Monahan, p. 260).

3.2 Shift register generators (SRG's)

LCG's are conceptually easy, and do not require much storage to compute. It tends to be difficult to get long periods with LCG's and their multidimensional properties can be surprisingly bad [LCG methods are not hopeless however, and many good ones exist such as the IMSL generators mentioned above]. Shift register methods overcome these difficulties, at some expense in storage and complexity in setting the algorithm up (indeed, sometimes an LCG is used to simulate the many seed values needed to set up a shift register generator!).

The seed for a SRG consists of p binary integers of length d,

$$\begin{aligned} V_1 &= (v_{11}, \ldots, v_{1d}) \\ V_2 &= (v_{21}, \ldots, v_{2d}) \\ &\vdots \\ V_p &= (v_{p1}, \ldots, v_{1d}) \end{aligned}$$

an offset $q < p$ is also chosen, and then V_{p+1}, V_{p+2}, etc., are computed from the recursion

$$V_k = V_{k-p} \oplus V_{k-p+q}$$

where $\oplus$ is bitwise exclusive-or:

v_{kj}	$v_{k'j}$	$v_{kj} \oplus v_{k'j}$
0	0	0
0	1	1
1	0	1
0	0	0

Two common SRG's are:

- Tausworthe generators, in which $d = 1$ and an M-bit $Unif[0, 1]$ deviate is formed by

$$U_k = \sum_{j=1}^{M} 2^{-j} v_{(k \cdot M+j)1}$$

- Generalized feedback shift register (GFSR) generators, in which $d = M$,

$$U_k = \sum_{j=1}^{M} 2^{-j} v_{kj}$$

  and the seed is formed from an initial set of bits $z_1, \ldots, z_{p+M}$ via

$$v_{ij} = z_{i+s(j)}$$

  where $0 = s(1) < s(2) < \ldots < s(M) \leq p$ are previously chosen "shift" constants.

SRG's, and GFSR generators in particular, tend to have very long periods and can have excellent multidimensional uniformity properties. However the multidimensional properties turn out to be heavily dependent on the initial seed, and a fair amount of mathematics has gone into characterizing good seeds for SRG's.

R offers several GSFR's as options (see `help(set.seed)`), for example.

3.3 Combining Generators

3.3.1 Shuffling

Two RNG's V_{1k} and V_{2k} can be combined by "shuffling" as follows:

- Fill a table $T_1, \ldots T_{128}$ with the first 128 values from $U_1.$: $T_j = U_{1,j}, j = 1, \ldots, 128$;
- Now for each k:
  1. Generate new values U_{1k}, U_{2k};
  2. Use the lower 7 bits of U_{2k} to define an index j;
  3. Output $U_k = T_j$
  4. Set T_j equal to U_{1k}

For example IMSL's `imsls_random_option` function offers the option of shuffling any one of its three LCG generators in this way, by taking $U_{1k} = U_{2k}$ =(output from the LCG). This cannot improve the period of the LCG (why?) but it may improve its correlational and spatial propertes.

3.3.2 Bitwise Exclusive-Or

A simpler way to combine two RNG's is to exclusive-or (XOR) them

$$U_k = U_{1k} \oplus U_{2k}$$

For example Splus's `runif()` generator is a modification of Marsaglia's "Super-Duper" RNG, which XOR's an LCG generator and a Tausworthe generator. The resulting generator has period of around 4.6×10^{18}, for most seeds (see `help(set.seed)`).

R also offers a modified "Super-Duper" RNG as one of several stock RNG's (see `help(set.seed)`). R also offers several other RNG's including a couple of GFSR generators and a newer multiplicative generator of Marsaglia's.

4 Choosing/Using a Random Number Generator

The library `<stdlib.h>` for C contains at least the random number generator `rand()`, which produces integers between 0 and `RAND_MAX`. In many versions of C, `rand()` is (I believe) an LCG with a defective period, whose low order bits are not very random (e.g. the 1's bit cycles from 0 to 1 to 0 and so forth; this sort of thing can happen when LCG's have less than the maximum possible period, for their "divisor" m).

Some versions of C also have a function `random()` and/or `rand48()` but what these do varies greatly from implementation to implementation. On our current Linux workstations, random() and rand() are the same, but what is implemented seems to be better than bad old `rand()`. In other versions of C, `random()` will simply be an interface for calls to bad old `rand()`; and in still others, `rand()` and `random()` are distinct RNG's.

If you are unsure of the quality of your RNG, try to use its higher-order bits rather than its lower-order bits. For example, to produce a random integer between 1 and 10, this code

```
j = 1 + floor(10.0*rand()/(RAND_MAX+1.0));
```

is better than this code

```
j = 1 + (rand() % 10);
```

The first example uses the high-order bits of `rand()` to take a fraction of 10. The second example uses modular arithmetic "`rand() mod 10`" to extract the low-order bits that add up to less than 10.

For serious work in C, I would probably opt to use a shuffled version of one of the LCG's in IMSL, and I would probably shuffle the LCG with the biggest multiplier, unless it was just too slow.

I have not had any experience with the RNG's in R. The modified Super-Duper RNG in Splus seems to work reasonably well for the Monte Carlo integration and MCMC work that I have done.

5 Quasi-Monte Carlo Integration

The usually-quoted error bound for numerical integration using iid sequences,

$$\left| \frac{1}{n} \sum_{i=1}^{n} h(X_i) - \int h(x) f(x) dx \right| = O(1/\sqrt{n})$$

is based on an argument that is ultimately *heuristic* in nature: We imagine that our deterministic sequence $X_1, X_2, \ldots$ *really is* random, so that the laws of probability theory give us, for example,

$$P \left[\left| \frac{1}{n} \sum_{i=1}^{n} h(X_i) - \int h(x) f(x) dx \right| \leq 2 * \sqrt{Var(h(X))/n} \right] \overset{n \to \infty}{\approx} 0.95$$

The heuristic can be very effective, especially if the underlying pseudo RNG is a "good" one, and in fact all of modern MC and MCMC is based on this heuristic.

If one were uncomfortable with the heuristic and instead wanted to formalize the rate of convergence of Monte Carlo integration, one would have to throw away the probabily heuristic and think of Monte Carlo methods as producing an irregular grid of points $U_1, U_2, \ldots$ that is *uniformly distributed* in the sense that, for any interval $(a, b) \subseteq (0, 1)$

$$\frac{\#\{i \leq n : \ U_i \in (a, b)\}}{n} \to b - a$$

and then derive error bounds based on what amount to Riemann-like sums of the form

$$\frac{1}{n} \sum_{i=1}^{n} h(X_i)$$

where the X_i's are an appropriate transformation of the U_i's.

However, pseudorandom uniform RNG's try to reproduce properties of random uniforms that may not be be advantageous for estimating error bounds, such as the fact that some sequences may place all their output in $[0, \frac{1}{2}]$ instead of spreading it out over the whole range from 0 to 1 (after all, there is some small probability that a long sequence of observations of a real $Unif[0, 1]$ would behave in this way).

Quasi-random numbers are numbers that are also uniformly distributed, but are constrained so that the empirical distribution is never far from uniform in any subinterval of $(0, 1)$. Thus, a quasi-random sequence $U_1, U_2, \ldots$ would tend not to place successive entries U_i, U_{i+1} close together, but rather space them apart to preserve uniformity. Thus, another way of thinking about them is that quasi-random numbers are like pseudo-random numbers with very strong negative correlations. These negative correlations tend to reduce the error in approximating the integral with the Riemann-like sum

$$\left| \frac{1}{n} \sum_{i=1}^{n} h(X_i) - \int h(x) f(x) dx \right|$$

(where again the X_i's are an appropriate transformation of the U_i's) in a way that is analogous to the fact that negatively correlated summands in the expression above would reduce the overall variance of the sum.

When error bounds for this method, called *Quasi-Monte Carlo integration*, can be found, they are generally better than the $1/\sqrt{n}$ rates of Monte Carlo integration methods. For this reason they are sometimes recommended in place of pseudo RNG based Monte Carlo methods.

However, the probability heuristic is very powerful; moreover, starting with uncorrelated uniforms $U_1, U_2, \ldots$ is much more flexible (e.g. there would be no MCMC if we had to use quasi-random numbers as the basic inputs, becuase the correlations between quasi-random nubmers would affect the behavior of the simulated Markov chain. Thus, use of pseudo-random numbers remains very popular.