

Simulation

Simulation means (here) the use of computer-generated data from specified stochastic mechanisms: an earlier term was *Monte Carlo methods*. This is often done to try things out, for example to find out if the approximate (asymptotic?) distribution of a test statistic or the coverage property of a confidence interval procedure holds for a realistic simulation of the problem of interest. The advantage of a simulation is that you do know¹ the true stochastic mechanism which generated the data.

Random Numbers

The fundamental building block from stochastic simulations is *random numbers*, meaning the generation of independent identically-distributed $U(0, 1)$ random variates.

Occasionally genuine random numbers are used, from physical sources (e.g. electronic noise). In 1955 RAND published a book of 10^6 random digits and 10^5 normally-distributed random numbers, and in the 1980's Marsaglia distributed a CD-ROM of random numbers. However, physical random numbers have drawbacks:

1. They can be far too slow to generate or read in.
2. If generated on the fly the simulation is not repeatable.
3. You rely on the physical mechanism being implemented perfectly, which in the case of the RAND tables was not the case.

As a result, the vast majority of random numbers used are *pseudo-random*², generated from a *seed* by a deterministic algorithm which makes a new number out of the fine details of the last number (or the last few numbers).

Not so long ago, users needed to know how this was done³ as it was usually done insufficiently well. Nowadays you can probably assume that the PRNG in a major statistical system is good enough, although you should be aware that all PRNGs have some systematic departures from randomness, and it is a good idea in critical studies to compare results from more than one PRNG. (This is not easily possible in S-PLUS, but it is in R, for example.)

To get pseudo-random numbers in S-PLUS, call the function `runif`. S-PLUS saves the *seed* as variable `.Random.seed` in the workspace, so each session starts using the pseudo-random number stream where the last one left off. If you want reproducible results, it is recommended to call the function `set.seed(i)` with $1 \leq i \leq 1000$ to choose one of a thousand pre-selected

¹unless you or someone else made a mistake in the implementation!

²as distinct from *quasi-random*, which is a different concept.

³the references will tell you if you are interested, and for S-PLUS see the appendix.

seeds. Then if you want to re-run the results (for example to collect more details or to use a different analysis), just call `set.seed(i)` again for the same `i`.

Other Random Variables

You will probably find that the system you are using has functions to generate random variable from the common non-uniform distributions: **S-PLUS** has `rnorm`, `rpois` However, it is helpful to understand the basic principles, as they are the same as are needed to simulate from more complex problem.

The basic task is to turn a stream $(U_i, i = 1, 2, \dots)$ of random numbers into samples from the specified stochastic mechanism. This should be done fast enough⁴ and without undue sensitivity to the fine details of the random numbers⁵.

Perhaps the simplest way to specify a univariate random variable is via its CDF F . Then if $X \sim F$ and F is continuous, $F(X) \sim U(0, 1)$. Inverting this shows that $F^{-1}(U)$ has CDF F if $U \sim U(0, 1)$. For a discrete distribution this still holds if we define

$$F^{-1}(x) = \min\{x \mid F(x) \geq u\}$$

This is known as *inversion*, and is often a good general method if F^{-1} is known⁶ or a fast approximation is available. For example, the exponential distribution has $F(x) = 1 - \exp -\lambda x$, so $F^{-1}(U) = -(1/\lambda) \log(1 - U)$, which can be simplified slightly as $1 - U$ and U have the same distribution. For a discrete distribution inversion amounts to searching a table of cumulative probabilities.

A great deal of ingenuity has been expended using stochastic mechanisms to get a desired distribution. For example, counting events in a Poisson process gives a Poisson-distributed random variable, and a Poisson process can easily be simulated by adding up exponential random variables⁷, at least provided the Poisson mean is not very large. Another ingenious idea is due to Box & Muller (1958):

1. Generate $U_1 \sim U(0, 1)$ and set $\Theta = 2\pi U_1$.
2. Generate $U_2 \sim U(0, 1)$ and set $E = -\log U_2$, $R = \sqrt{2E}$.
3. Return $X = R \cos \Theta$, $Y = R \sin \Theta$.

Then X, Y are independent standard normal variates: if we only want one we can throw Y away or keep it in our pocket for next time we are asked.

One other general principle is worth knowing, that of *rejection sampling*. Suppose we know how to simulate from probability density (or mass function) g with the same support⁸ as f such that f/g is bounded by $M < \infty$. Then we can create a sample from f by

⁴which will include not using an excessive number of the U_i

⁵so simulating coin tossing by readings the bits of U_i in some binary representation is not a good idea.

⁶it is the quantile function, given in **S-PLUS** as `qnorm` etc.

⁷and adding up $-\log U_i$ can be replaced by multiplying U_i .

⁸ $g > 0$ wherever $f > 0$.

Repeat
 Generate Y from g .
 Generate $U \sim U(0, 1)$.
until $MU \leq f(Y)/g(Y)$.

The expected number of tries is M , so this works best if g and f are closely matched: it is often used to make approximate methods of simulation exact.

Another advantage of rejection sampling is that we don't need to know f , only a function $f' \propto f$ and a bound on f'/g . This allows us to ignore normalizing constants which may be awkward to compute, or even not known explicitly.

The references (especially Devroye, 1986) cover a plethora of techniques and applications of them to standard distributions. It is unlikely these days that you will need to actually implement them: look around for existing implementations in whichever programming language you are using.

One important technique is nowadays known as *Markov Chain Monte Carlo* and will be covered in a module in Trinity Term.

Simulation Experiments

The most important thing to remember when you are using simulation is that you are **performing an experiment**. So, the experiment should be designed, and the data it produces should be analysed.

The main point in the design of a simulation experiment is to eliminate inessential variation and so make the results as precise as possible. The first and most important idea is to compare ideas on the same simulation runs, sometimes known as using *common random numbers* although it is just an application of *blocking*. Other ideas are ways to reduce variability:

control variates: estimate the difference between the result of interest and a similar one which can be obtained analytically and also estimated from the data.

antithetic variates: deliberately introduce negative correlations between runs: rarely useful.

importance sampling: simulate from a different distribution and adjust. Useful to make rare events less rare, for example.

The more complicated the problem the less likely these are to be useful, but they should be borne in mind if a simulation experiment is taking too long.

The main difficulty with *analysis* is dependence in the results. Many simulations are of events through time, and so the results from a time series – there are specialized methods of analysis for such time series.

Various uses of simulation methods in statistical inference will be covered in the *Computer-Intensive Statistics* module.

Reference books

Bratley, P., Fox, B. L. and Schrage, L. E. (1987) *A Course in Simulation*. Second Edition. Springer.

Devroye, L. (1986) *Non-uniform Random Variate Generation*. Springer.

Ripley, B. D. (1987) *Stochastic Simulation*. Wiley.

Ross, S. M. (1996) *Simulation*. Second Edition. Academic Press.

Appendix: the PRNG in S-PLUS

The current generator is based on George Marsaglia's "Super-Duper" package from about 1973. The generator produces a 32-bit integer whose top 31 bits are divided by 2^{31} to produce a real number in $[0, 1)$. The 32-bit integer is produced by a bitwise exclusive-or of two 32-bit integers produced by separate generators. The C code for the random number generator is, (effectively,

```
unsigned int congrval, tausval; # assume 32-bit
static double xuni()
{
  unsigned int n, lambda = 69069, res;
  do {
    congrval = congrval * lambda;
    tausval ^= tausval >> 15;
    tausval ^= tausval << 17;
    n = tausval ^ congrval;
    res = (n>>1) & 017777777777;
  } while(res == 0);
  return (res / 2147483648.);
}
```

The integer `congrval` follows the congruential generator

$$X_{i+1} = 69069X_i \bmod 2^{32}$$

as unsigned integer overflows are (silently) reduced modulo 2^{32} ; that is, the overflowing bits are discarded. As this is a multiplicative generator (no additive constant), its period is 2^{30} and the bottom bit must always be odd (including for the seed).

The integer `tausval` follows a 32-bit Tausworthe generator (of period $4\,292\,868\,097 < 2^{32} - 1$):

$$b_i = b_{i-p} \text{ xor } b_{i-(p-q)}, \quad p = 32, \quad q = 15$$

This follows from a slight modification of Algorithm 2.1 of Ripley (1987, p. 28). (In fact, the period quoted is true only for the vast majority of starting values; for the remaining 2 099 198 non-zero initial values there are shorter cycles.)

For most starting seeds the period of the generator is $2^{30} \times 4\,292\,868\,097 \approx 4.6 \times 10^{18}$, that is quite sufficient for calculations that can be done in a reasonable time in S-PLUS. The current values of `congrval` and `tausval` are encoded in the vector `.Random.seed`, a vector of 12 integers in the range $0, \dots, 63$. If x represents `.Random.seed`, we have

$$\text{congrval} = \sum_{i=1}^6 x_i 2^{6(i-1)} \quad \text{and} \quad \text{tausval} = \sum_{i=1}^6 x_{i+6} 2^{6(i-1)}$$