A Pseudo-Random Number Generator for Spreadsheets

Research Note, Jan 2013

Michael Lampton, Space Sciences Lab, UC Berkeley

Abstract

Setting up a spreadsheet to simulate noisy data collection from an experiment requires a generator of pseudo-random numbers. The function RAND built into popular spreadsheets is unsuitable because it rerandomizes every time the spreadsheet is recalculated. Unlike measured data, RAND changes for each data analysis activity. Moreover, spreadsheet rootfinding plugins necessarily recalculate the entire spreadsheet for each internal iteration, and RAND cannot keep its output constant during a sequence of iterations. Beyond that there is no way to enforce a common seed, to verify agreement between instructor and student statistical results. Here, I introduce a well-tested random number generator that overcomes these limitations and I show how to make it portable to spreadsheets and high-level computer languages.

Keywords: random numbers, spreadsheets.

Goals

For decades, physics instructors have used spreadsheets to organize computational work in the classroom and laboratory. To statistically model measurement errors, some kind of pseudo-random number (PRN) generator is essential. Desirable features of a simple spreadsheet-based PRN generator (PRNG) are:

- Occupies a single spreadsheet cell;
- Specifies a single input cell for its argument or seed;
- Output spans 0<x<1 for use with distribution generators NORMINV, TINV, etc.;
- Has a period that far exceeds the likely number of rows in an experiment simulator;
- Has correct means, variances, uniformities, and autocorrelations;
- Has a white (flat) power spectrum;
- Pseudo-random sequences (PRSs) from different seeds should be statistically orthogonal;
- For the same seed, different spreadsheets should yield the same PRS;
- Should also be portable between spreadsheets and high level languages.

Generators designed for cryptography must pass exceedingly demanding tests; see L'Ecuyer (2010) and references therein for examples of current approaches. Those generators are complex. In contrast, a simple seed-controlled PRNG good for thousands (not billions!) of iterations would be a useful spreadsheet component. In this Note, I write the Lewis-Goodman-Miller (LGM) generator in spreadsheet form and show how to seed it, as an aid to those who need a fully controlled source of PRNs to model experimental noise or measurement errors. L'Ecuyer (2001) offers test results comparing a variety of popular RNG implementations, including this one.

A fully portable generator

The most popular simple PRN generators are based on the multiplicative linear algorithm originally by Lehmer (1949); see also Lewis Goodman and Miller (1969), Wichmann and Hill (1982), Park and Miller (1988), Press, Flannery, Teukolsky, and Vetterling (1988), Knuth (1997), L'Ecuyer (2001), and Wichmann and Hill (2006). This generator receives an integer Zin and produces an integer Zout:

$$\text{Zout} = (\text{A}\cdot\text{Zin}) \bmod \text{M} \qquad (1)$$

Briefly, the low order bits of the input integer Zin are hoisted to higher significance by an integer multiplier A, and the highest order bits are discarded by the modulus operation. The modulus M sets the finesse of the comb of output values. If M and A are properly chosen, the period of the sequence will have the maximum possible length of M- 1. Owing to its speed and simplicity it has enjoyed a long life and has been ported to a variety of environments. Obviously it must never be given a seed of zero, or the whole sequence would collapse. Indeed, seed values very near zero cause the first few iterates to be substandard in size (see Wichmann and Hill (2006)), and seeds must be randomized if the PRS is to have a random looking startup sequence. I offer a seeder in the next section.

Exact integer arithmetic is essential if a generator is to be portable. The IEEE-754 double precision specification requires exact integer arithmetic in the range -2^{52} to $+2^{52}-1$ and this standard is widely obeyed by high-level languages. However, spreadsheets typically deliver exact integer arithmetic only for numbers whose size is less than about 10^{15} i.e., fifteen digits accuracy or about 2^{50} (see Almiron et al. 2010). To avoid overflow, a PRN algorithm internal integer product A·M should be comfortably smaller than this fifteen-digit bound.

A second requirement is that the generator must deliver variates in the range $0 < Z < 1$ so that these can be fed into the appropriate inverse distribution generator, for example NORMINV or TINV spreadsheet functions. This requirement is customarily met by moving Equation 1 into a floating point environment, with M·Zin playing the role of the integer Zin:

$$\text{Zout} = ((M\cdot A\cdot \text{Zin}) \bmod M)/M \qquad (2)$$

I adopt the Lewis-Goodman-Miller (1969) “minimal standard generator" (see Park and Miller 1988) defined by the constants

$$M = 2^{31}-1 = 2147483647 \qquad (3a)$$
$$A = 7^{5} = 16807 \qquad (3b)$$

This generator has been exhaustively tested and has been found to be generally acceptable within its limited sequence length of M-1. This generator will be portable among systems whose arithmetic correctly handles numbers of magnitude A·M without loss of integer accuracy. Here, the largest number that has to be handled is the MA product, about 3.6E13. This is comfortably within the 15-digit accuracy of all popular spreadsheets. Features of this PRNG are:

- Successive iterates are uncorrelated;
- Beyond that, they pass the Knuth spectral test for dimensions 2,3,4,5, and 6;
- The PRNG is maximal length (here equal to M-1);
- The PRNG populates the variate axis uniformly;
- The PRNG is portable: it delivers the same PRS on every platform.

In a spreadsheet cell, Equation 2 is implemented with the expression

```
=MOD(ROUND(M*A*Z,0),M)/M                    (4)
```

Again, M and A are the constants from Equation 3. Z stands for the address (column,row) of whichever cell contains the previous RN in your sequence, or, for the initial RN, it is the address of your seed generator. The ROUND(X,0) operation is essential for portability: it reestablishes the correct integer product by removing the floating point division errors (order of 10^{-15}) that arise in the previous iteration division. Without ROUND(), the PRS would depend on the chain of division errors which would introduce fractional terms into the sum. These fractional terms differ among spreadsheets and can also differ from the IEEE-754 oating point specifications, causing the various platform PRSs to diverge after some number of iterations.

Portable seeding

How should this function be seeded? In experiment modeling, users will want a selection of seeds that deliver independent statistics. The entire PRS period is about two billion, so if each segment has 1000 iterates, any randomly chosen seed will have only one chance in a million of overlapping any other given seed segment. Very good odds! but of course the RNG is totally deterministic and the seeder must be verified for freedom from overlap. Fractional seeds are essential since all integer inputs are equivalent to zero seed and yield the same null PRS.

A good seeder will accept an integer run number and deliver a prerandomized seed value. (A simple list of seeds like 0.1, 0.2, etc., would fail the portability test because they are unlikely to be found among the comb of PRS values.) A prerandomized seed makes the PRNG self-starting, requiring no warm-up iterations before use. I address this issue here by offering an explicit seed() function that accepts an integer run number IRUN=1, 2, ... and delivers a starting point for a PRS. For portability, I precondition the seed by applying the same modulus treatment that each RN has:

```
=MOD(ROUND(MOD(IRUN*EXP(1),1)*M*A,0),M)/M          (5)
```

Here, the constant EXP(1)=2.71828... supplies some fractional digits that are boosted by the integer run number IRUN. That fractional part is then boosted by the MA product to fill the working span of double precision integers. That product is then reduced modulo M, and normalized to unit span in the same way that PRS numbers are reduced, so that each seed is a member of the PRS comb. These actions make every seed compute the same way on all platforms.

Portability Check

Park and Miller (1988) emphasize that exhaustive statistical testing is exceedingly demanding of resources (see for example Fishman and Moore (1986)) and recommend that any portable RNG should be tested for correctness rather than for its statistics. For the constants in Equation 3 above and a seed derived from IRUN=1, iteration 10000 should yield Z = 0.785320384794.   I verified this result for Microsoft Excel 2007 (PC edition), Gnome Gnumeric 1.10.16, Open Office Calc 3.3.0, Java 1.5.0, Gnu C, and Python 2.6.6. Portions of these runs are listed in Table 1 below. All platforms tested are in agreement over the range of parameters tested, giving good evidence of portability.

Table 1: Iterations on Various Platforms

IRUN=	**1**	**2**	**3**	**999**
Seed=	0.162690911052	0.325381822570	0.488072733622	0.528220262159
Excel iter=1	0.346142053300	0.692291932969	0.038433986268	0.797946102357
Excel iter=2	0.609489807212	0.350517402566	0.960007209778	0.080142321568
Excel iter=3	0.695189804628	0.145984931451	0.841174736079	0.951998594195
Excel iter=4	0.055046384714	0.568742901352	0.623789286066	0.240372629482
Excel iter=10000	0.785320384794	0.056613301419	0.841933686213	0.887922685076
Gnumeric iter=10000	0.785320384794	0.056613301419	0.841933686213	0.887922685076
Calc iter=10000	0.785320384794	0.056613301419	0.841933686213	0.887922685076
Java iter=10000	0.785320384794	0.056613301419	0.841933686213	0.887922685076

Table 1: Demonstration of portability among platforms. For run numbers listed, PRS iterations 1, 2, 3, 4, and 10000 are shown to 12 digits accuracy. Iteration 10000 is shown for Excel 2007 (PC); Gnumeric (PC); OpenOffice3 Calc(Mac); and Java which is IEEE-754 compliant.

Testing the seeder

Any PRS run should not overlap any portion of any other PRS run. The seeder generates a list of random-appearing seeds that start the PRSs. Figure 1 below shows the log of the maximum nonoverlapping PRS length as a function of the log of the number of runs (blue) and also (red) the maximum possible PRS length if the segments were to be uniformly arranged.

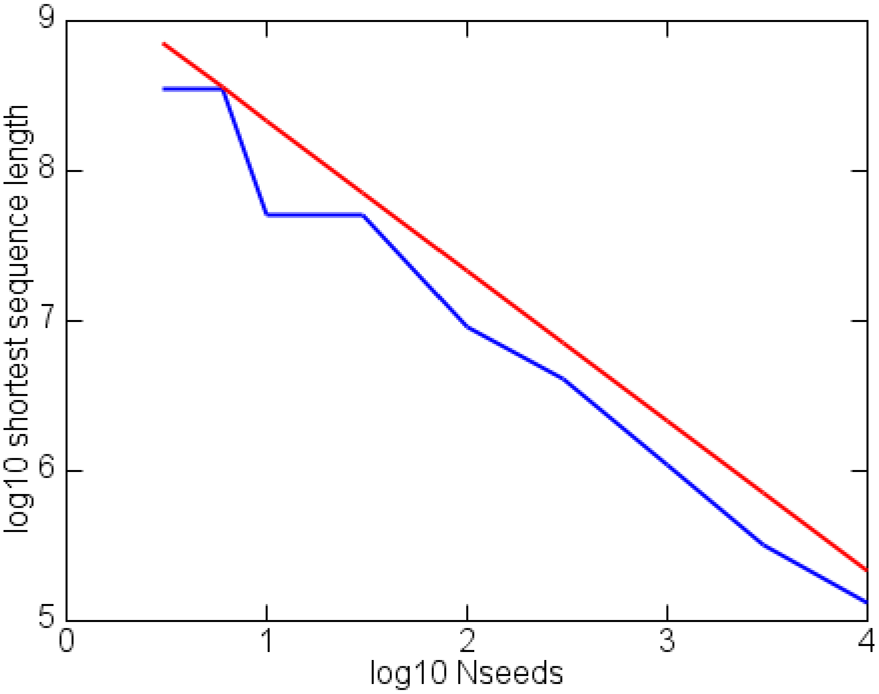

Figure 1: Plot showing the log of the maximum available sequence length without overlapping any other sequence as a function of the log of the number of seeds chosen. For 100 runs, individual sequences can have millions of iterations without overlap.

A handy on-off switch

I noted that IRUN=0 and its seed value of zero is prohibited because zero collapses the entire PRS. If however each PRN is used solely to feed an inverse cumulative probability function such as NORMINV, this all-zero off state can be recognized and employed to switch off the random deviations throughout the worksheet. Use the expression

=IF(Z=0,0,NORMINV(Z,0,1)) (6)

where Z is again the (column, row) address of the uniform PRN being converted.

Conclusions

In Equations 4 and 5, I have presented a portable uniform random number generator that is under user control: its variates are e_ectively random and independent for a wide range of seed values, but then | like real measurements | they become constant during subsequent data analysis. Background theory shows good performance for small scale simulation situations found in the classroom and laboratory.

Acknowledgment

The author greatfully acknowledges the support by the Director, Office of Science, U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

References

Almiron MG, et al. (2010). “On the Numerical Accuracy of Spreadsheets." J. Stat. Software, 34(4), 1-24.

Fishman GS, Moore LR (1986). “An Exhaustive Analysis of Multiplicative Congruential Random Number Generators with Modulus 2^{31}-1" SIAM J. Scientific and Statistical Computing, 7, 24-45.

Knuth DE (1997). “The Art of Computer Programming, volume 2: Seminumerical Algorithms.” 3rd edition. Addison Wesley, Reading, Massachusetts.

L’Ecuyer P (2001) “Software for Uniform Random Number Generation: Distinguishing the Good and the Bad,” Proceedings of the 2001 Winter Simulation Conference, IEEE Press, Dec. 2001, 95-105.

L'Ecuyer P (2010), ``Pseudorandom Number Generators", in Encyclopedia of Quantitative Finance, R. Cont, Ed., in volume Simulation Methods in Financial Engineering, E. Platen and P. Jaeckel Eds., John Wiley, Chichester, UK, 1431-1437.

Lehmer DH (1949). “Mathematical Methods in Large-scale Computing Units." Proc. 2nd Symposium on Large-Scale Digital Calculating Machinery, pp. 141-146; also Annals of the Computation Laboratory of Harvard University, v.26 (1951).

Lewis PAW, Goodman AS, and Miller JM (1969) “A pseudo-random number generator for the System/360,” IBM Systems Journal v.8 136-143.

Park SK, Miller KW (1988). “Random Number Generators: Good Ones are Hard to Find." Communications of the ACM, pp. 1192-1201.

Press WH, Flannery BP, Teukolsky SA, Vetterling WT (1988). “Numerical Recipes” 2nd edition. Cambridge University Press.

Wichmann BA, Hill ID (1982). “Algorithm AS183: An Effcient and Portable Pseudo-Random Number Generator." Applied Statistics, 31(2), 188-190.

Wichmann BA, Hill ID (2006). “Generating Good Pseudo-Random Numbers." Computational Statistics and Data Analysis, 51(3), 1614-1622.