

A Pseudo-Random Number Generator for Spreadsheets

Author Information will be provided here
3 January 2012

Setting up a spreadsheet to simulate noisy data collection from an experiment requires a supply of pseudo-random numbers (PRNs). The function RAND built into Microsoft Excel® and other popular spreadsheets is unsuitable because it rerandomizes every time the spreadsheet is recalculated: unlike measured data, RAND changes for each data analysis activity. Solver® plugins necessarily recalculate the entire spreadsheet for each of its internal iterations, and RAND cannot keep its output constant during a fit and is therefore incompatible with Solver. Beyond that there is no way to check agreement between instructor and student statistics. Here, I introduce a well-tested random number generator (RNG) that overcomes these limitations and I show how to make it portable to spreadsheets and high-level computer languages.

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 random number generator is essential. Desirable features of a spreadsheet-based PRN generator are:

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

A Fully Portable Generator

The most popular PRN generators are based on the *multiplicative linear algorithm* (originally by Lehmer¹; see also references²⁻⁸) which receives an integer Z_{in} and produces an integer Z_{out} :

$$Z_{out} = (A \cdot Z_{in}) \bmod M \quad (\text{eqn 1}).$$

Briefly, the low order bits of the input integer Z_{in} 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=0 or the whole sequence would collapse. Indeed, seed values very near zero cause the first few iterates to be substandard in size (see ref.⁷), 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 ref⁹). To avoid overflow, a PRN algorithm’s internal integer product $A \cdot M$ 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 eqn 1 into a floating point environment, with $M \cdot Z_{in}$ playing the role of eqn 1’s integer Z_{in} :

$$Z_{out} = ((A \cdot M \cdot Z_{in}) \bmod M) / M \quad (\text{eqn 2})$$

I adopt the Park & Miller⁴ “minimal standard generator” defined by the constants

$$M = 2^{31} - 1 = 2147483647 \quad (\text{eqn 3a})$$

$$A = 7^5 = 16807 \quad (\text{eqn 3b})$$

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 $A \cdot M$ without loss of integer accuracy. Here, the largest product $A \cdot M \approx 3 \cdot 10^{13}$, comfortably within the 15-digit accuracy of all popular spreadsheets. Features of this PRS are:

1. Successive iterations are uncorrelated;
2. Beyond that, they pass Knuth’s spectral test for dimensions 2,3,4,5, and 6;
3. Deliver a period of maximal length (here equal to $M-1$) before repeating;
4. Populate the variate axis uniformly;
5. Be portable: given the same seed, deliver the same PRS on every platform.

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

$$=\text{MOD}(\text{ROUND}(A \cdot M \cdot Z, 0), M) / M \quad (\text{eqn 4}).$$

Here, M and A are the constants from eqn 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 the 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 $\sim 10^{-15}$) that arise in the previous iteration’s 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 floating point specifications, causing the various platform PRSs to diverge after some 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's 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 of course required 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 prandomized seed value. This feature helps make the PRS 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. A simple list of seeds like 0.1, 0.2, etc will fail the portability test because the numerators in equation 2 are likely to lack an exact binary integer representation, and the denominator certainly lacks that. For portability, precondition the seed by applying the same modulus treatment that each RN has:

$$\text{MOD}(\text{ROUND}(\text{MOD}(\text{IRUN} \cdot \text{EXP}(1), 1) \cdot A \cdot M, 0), M) / M \quad (\text{eqn } 5) .$$

Here, the constant $\text{exp}(1)=2.71828\dots$ supplies some fractional digits that are boosted by the run number IRUN. That fractional part is then boosted by the $M \cdot A$ 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. This action makes the seed value one of the ~ 2E9 valid seeds that compute the same way on all platforms.

Testing for Portability

Park & Miller⁴ emphasize that exhaustive statistical testing is exceedingly demanding of resources (see for example Fishman & Moore⁵) and recommend that any portable RNG should be tested for correctness rather than for its statistics. For the constants in eqn 2 above and $\text{seed}=\text{seeder}(1)$, iter 10000 should yield $Z=0.785320384794$. I verified this result for Microsoft Excel 2007 (PC edition), 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 Figure 1 below. All platforms tested are in agreement over the range of parameters tested, giving good evidence of portability.

IRUN=	1	2	3	4	999
seed=	0.162690911052	0.325381822570	0.488072733622	0.650763644674	0.528220262159
iter=1	0.346142053300	0.692291932969	0.038433986268	0.384576039568	0.797946102357
iter=2	0.609489807212	0.350517402566	0.960007209778	0.569497016989	0.080142321568
iter=3	0.695189804628	0.145984931451	0.841174736079	0.536364540707	0.951998594195
iter=4	0.055046384714	0.568742901352	0.623789286066	0.678835670780	0.240372629482
Excel 10000	0.785320384794	0.056613301419	0.841933686213	0.617254071006	0.887922685076
Gnumeric 10000	0.785320384794	0.056613301419	0.841933686213	0.627254071006	0.887922685076
OpenOffice 10000	0.785320384794	0.056613301419	0.841933686213	0.627254071006	0.887922685076
IEEE-754 10000	0.785320384794	0.056613301419	0.841933686213	0.627254071006	0.887922685076

Figure 1: 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 (Mac); and Java.

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 2 shows the log of the maximum PRS length as a function of the log of the number of runs (blue) and also (red) the maximum PRS length if the segments were to be uniformly arranged, head-to-tail, without gaps.

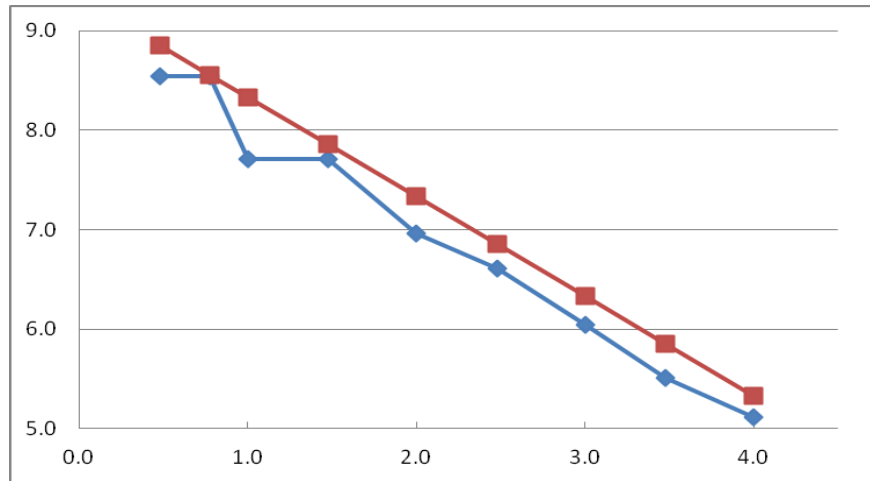


Figure 2. Plot showing the log of the maximum available sequence length without overlapping any other sequence as a function of the log of the number of seeds chosen. For 100 runs, individual sequences can have millions of iterations without overlap.

Demonstrating the RNG Statistics by Histogram

The statistical uniformity and scatter of the RNG can be shown with a simple histogram showing the accumulated hits in a number of equal size bins that span the variate range $0 < z < 1$. In Figure 3, I have set up 100 equal bins in Java, and overplotted 100 runs of 60000 iterations. Each run should average about 600 hits per bin. The accumulated statistics show good homogeneity and a Gaussian distribution of hit densities.

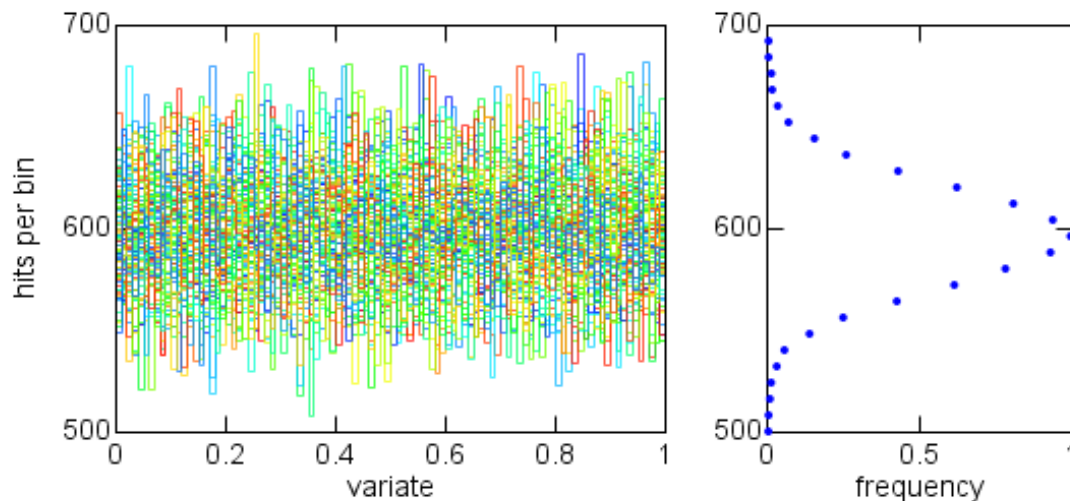


Figure 3: For a set of 100 equal bins spanning $0 < z < 1$, the number of RNs falling into each bin is plotted. 60000 iterations per run; 100 runs. Right: bin probabilities.

Demonstrating the RNG Statistics by Power Spectrum

A Fourier analysis of the each PRS can in principle reveal subtle periodicities that would escape binning tests. In Fig. 4, I show Java power spectra of 4096-length PRS vectors for IRUN=1 and 2. The Fourier power values are expected to be distributed with a negative exponential distribution. Fig. 4 shows that this is the case, without strongly grouped or peaked frequency features and with the expected power distribution.

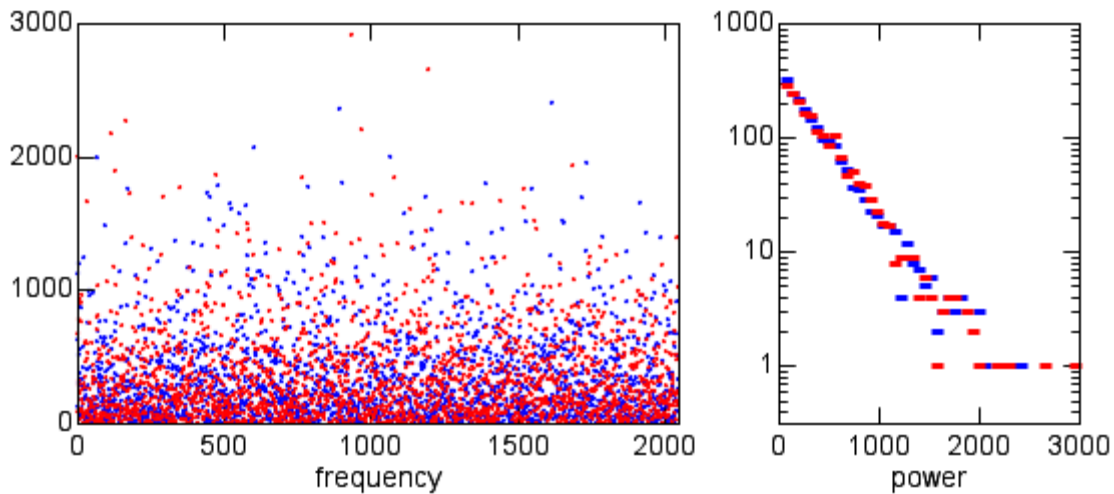


Figure 4: Left: Fourier power at frequencies 1 to 2048 cycles per sequence, for seeds=1(red) and 2 (blue). Right: histogram of power values; a negative exponential distribution would average to a straight line.

A Handy On-Off Switch

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

$$=IF(Z=0, 0, NORMINV(Z,0,1)) \quad (\text{eqn 6}),$$

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

Conclusions

In eqn 4, I have shown portable uniform random number generator that is under user control: its variates are apparently random for any choice of seed, but then – like real measurements – they become constant during data analysis. Background theory and two simple demonstrations show good performance for small scale simulation situations found in the classroom and laboratory.

Acknowledgment

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

References

1. D. H. Lehmer, "Mathematical methods in large-scale computing units," Proc. 2nd Symposium on Large-Scale Digital Calculating Machinery pp.141-146, (1949); MR44899.
2. Donald E. Knuth, "The Art of Computer Programming, Volume II : Seminumerical Algorithms," 3rd edition. Addison-Wesley (1998); section 3.2.1.
3. William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, "Numerical Recipes," 2nd Edition, Cambridge University Press (1988).
4. Stephen K. Park and Keith W. Miller, "Random Number Generators: Good Ones are Hard to Find," Commun. ACM, pp.1192-1201 (1988).
5. George S. Fishman and Louis R. Moore III, "An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$," SIAM J. Sci. & Stat. Comput., v.7#1, 24-45, (1986).
6. Pierre L'Ecuyer, "Random number generation," Chapter 4 of *Handbook on Simulation*, Ed. J Banks; Wiley 1998.
7. B. A. Wichmann and I. D. Hill, "Algorithm AS183: An efficient and portable pseudo-random number generator," Applied Statistics v.31 #2 pp.188-190 (1982).
8. B. A. Wichmann and I. D. Hill, "Generating Good Pseudo-Random Numbers," Computational Statistics & Data Analysis, v.51(3), 1614-1622 (2006).
9. M. G. Almiron, B. Lopez, A. L. C. Oliviera, A. C. Medeiros, and A. C. Frery, "On the numerical accuracy of spreadsheets," J. Stat. Software, v.34#4, pp.1-24 (2010).