Conversion of High-Period Random Numbers to Floating Point

JURGEN A DOORNIK
University of Oxford

Conversion of unsigned 32-bit random integers to double precision floating point is discussed. It is shown that the standard practice can be unnecessarily slow and inflexible. It is argued that simulation experiments could benefit from making better use of the available precision.

Categories and Subject Descriptors: G.3 [**Probability and Statistics**]: Random number generation; G.4 [**Mathematical Software**]: Algorithm design and analysis, Efficiency

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Conversion to floating point, Precision

1. INTRODUCTION

Recent research into random number generation has resulted in several very high period generators. The emphasis is on efficient implementation using unsigned integer arithmetic, primarily written in C. Conversion to a uniform $[0, 1)$ random number is mostly done through multiplication by an appropriately small number.

Two issues are addressed in this note. First, conversion to floating point can be implemented more efficiently, without affecting the properties of the RNG. Allowance is made to exclude zero, which is important in many statistical applications. Secondly, most simulations are implemented in double precision, but rarely are uniform random numbers generated that use the full precision.

These issues have received limited attention in the literature. Press et al. [1993, p.285] consider bit-wise conversion for single precision. The code for the Mersenne Twister has a double precision conversion attributed to Isaku Wada. An alternative approach would be to implement a uniform RNG in double precision throughout.

2. CONVERSION TO FLOATING POINT

2.1 Conversion of 32-bit integers

Many recently proposed random number generators are implemented in C using 32-bit unsigned integers for maximum computational efficiency. Multiple integers are used to achieve very high periods. Conversion to uniform random numbers is usually done by multiplying the random integer by the floating-point value that

This research was supported by ESRC grant RES-000-23-0539.
Author's address: Nuffield College, University of Oxford, Oxford OX1 1NF, UK.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2006 ACM 0098-3500/2006/1200-0001 $5.00

corresponds to 2^{-32}, as in function `Ran32Dbl0`:[1]

```
#define M_RAN_INVM32    2.32830643653869628906e-010
double Ran32Dbl0(unsigned int uiRan)
{   return uiRan * M_RAN_INVM32;    }
```

Provided that the RNG never returns zero, the uniform random numbers range from 0.999999999767169 to $2.3283064365387 \times 10^{-10}$ as `uiRan` ranges from 0x00000001 to 0xFFFFFFFF.[2] If the RNG can also return zero, that will obviously result in zero from `Ran32Dbl0`.

Surprisingly, there is another implementation which can be more efficient on some of the most commonly used hardware:

```
double Ran32Dbl1(unsigned int uiRan)
{   return (int)uiRan * M_RAN_INVM32 + 0.5;    }
```

The unsigned random integer is cast to a signed integer (which is costless), and multiplied by 2^{-32}. This gives a value in $[-0.5, 0.5)$ to which 0.5 must be added. The values are now generated in a different order: for the unsigned integer running from 0 to the maximum, the uniform values run from 0.5 to 0.999999999767169 and then from 0.0 to 0.4999999997671694. In `Ran32Dbl0`, because a floating point value is always signed, effectively 64 bits need to be loaded and then translated to floating point.[3] `Ran32Dbl1` can be faster because it uses some additional information about the outcome, and only 32 bits need to be loaded and converted to floating point.

A second benefit of `Ran32Dbl1` is that it is costless to create a version that excludes zero, by adding $2^{-32}/2$ to all values (half the step size of the grid of numbers). Many rejection methods and statistical applications take logarithms, in which case it is important to avoid returning zero.

2.2 Implications

Consider the underlying RNG as iteratively updating a vector of bits of size k from an initial seed $\mathbf{s}_0 = (x_0, \ldots, x_{k-1})'$. At iteration j the uniform random number is $u_j = \sum_{i=1}^{L} x_{jk+i-1} 2^{-i}$ where $u_j \in [0, 1)$ and L is the word length (32 here).

When $\tilde{u}_j = 2^{-L-1} + \sum_{i=1}^{L/2} x_{jk+i-1} 2^{-i-L/2} + \sum_{i=L/2+1}^{L} x_{jk+i-1} 2^{-i+L/2}$ is used instead:

$$\tilde{u}_j - 2^{-L-1} = \tilde{\mathbf{x}}_j' \mathbf{c} = \mathbf{x}_j' \begin{pmatrix} 0 & \mathbf{I}_{L/2} \\ \mathbf{I}_{L/2} & 0 \end{pmatrix} \mathbf{c} = \mathbf{x}_j' \mathbf{P}_L \mathbf{c}.$$

Here, $\mathbf{x}_j = (x_{jk}, \ldots, x_{jk+L-1})'$, $\mathbf{c} = (2^{-1}, \ldots, 2^{L-1})'$ and $\mathbf{P}_L$ is a permutation matrix which is full rank and orthogonal (as well as symmetric).

The constant term has no impact other than excluding zero as a possible outcome. Statistical tests, while numerically different, will have the same expected outcome.

Equidistribution refers to the distribution of t-dimensional vectors of successive u_j's on the unit hypercube [L'Ecuyer 1996]. Careful consideration is given

[1] For clarity, the code is presented as a separate function, but will normally form part of the RNG.
[2] 0x... indicates hexadecimal notation, as used in C-style languages.
[3] Some compilers may do this automatically for `Ran32Dbl0` when optimization is switched on: convert as a signed integer, but add 2^{32} if it is negative. This test still carries significant overhead.

to equidistributional performance of modern generators [Panneton et al. 2006]. Using $\tilde{u}$ instead of u amounts to a rotation of the unit hypercube, and has no impact on the equidistribution properties.[4]

Conversion which shifts right by one bit and multiplies the signed integer (now non-negative) by 2^{-31} would be as fast as `Ran32Dbl1`. However, the transformation now reduces rank. The preference here is to increase precision, not reduce it.

2.3 Increased precision

When simulations are implemented in double precision, as often the case, the practice is to use 32 random bits as in `Ran32Dbl0`. However, the IEEE 754 definition of a 64-bit floating point value provides for a mantissa of 53 bits (normalized, so taking 52 bits). In decimal terms, this corresponds roughly to only using about 9–10 digits of the available 15–16. One can envisage situations, such as randomized searches or maximization, where this matters.

I propose the following 32 and 52 bit C macros to convert to floating point:[5]

```
#define M_RAN_INVM52    2.22044604925031308085e-016
#define RANDBL_32new(iRan1)                    \
    ((int)(iRan1) * M_RAN_INVM32 + (0.5 + M_RAN_INVM32 / 2))
#define RANDBL_52new(iRan1, iRan2)             \
    ((int)(iRan1) * M_RAN_INVM32 + (0.5 + M_RAN_INVM52 / 2) + \
        (int)((iRan2) & 0x000FFFFF) * M_RAN_INVM52)
```

This avoids the potential slowness of converting an unsigned integer, and creates an attractive symmetry: $\min(\tilde{u}_j) = \min(1 - \tilde{u}_j)$. This is 1.16415321826935e-010 for 32 bits and $1.11022302462516e-016$ for 52 bits.

2.4 Comparison

Two integer-based random number generators are used to compare the procedures for conversion to floating point. MWC8222 is a multiply-with-carry generator [Marsaglia 2003b; 2003a], with a period of $2^{8222} - 1$. The Mersenne twister [Matsumoto and Nishimura 1998] uses a binary linear recurrence on an array of 624 integers that is updated once every 624 steps. The version for which computer code is provided[6] is called MT19937, with period $2^{19937} - 1$. Panneton et al. [2006] extend this type of generators to a class labelled WELL. DLF202 [Marsaglia and Tsang 2004] has period $2^{202} - 1$ and uses double precision variables throughout.

Table I provides the timings to sum 10^9 floating point random numbers for a range of hardware and compilers. The '32def' versions return u_j using a macro along the lines of `Ran32Dbl0`, while the versions with '32new' use the macro `RANDBL_32new` of §2.3. In most cases the differences are quite astounding: about 50% speed-

[4]Linear generators on $\mathbb{F}_2$ can be rewritten to obtain the original output. Write it as $\mathbf{x}_j = \mathbf{A}\mathbf{x}_{j-1}$, $\mathbf{y}_j = \mathbf{B}\mathbf{x}_j$, $u_j = \mathbf{y}_j'\mathbf{c}$, with $\mathbf{A}$ a $k \times k$ transition matrix, and $\mathbf{B}$ an $L \times k$ selection matrix. Let $\mathbf{Q}$ be the $k \times k$ permutation matrix, block diagonal with $\mathbf{P}_L$ along the diagonal: $\tilde{\mathbf{x}}_j = \mathbf{QAQQ}\mathbf{x}_{j-1} = \tilde{\mathbf{A}}\tilde{\mathbf{x}}_{j-1}$, $\tilde{\mathbf{y}}_j = \mathbf{P}_L\mathbf{BQQ}\mathbf{x}_j = \tilde{\mathbf{B}}\tilde{\mathbf{x}}_j$. Applying the new procedure to this routine gives $\tilde{\mathbf{y}}_j'\mathbf{P}_L\mathbf{c} = u_j$.

[5]A 48 bit version was also considered, using 3 integers to create 2 doubles. However, this was usually slower than the 52-bit version.

[6]For mt19937ar.c see www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html. For `RANDBL_52new` versions, the required two random integers are generated inside the function to improve efficiency.

up from this seemingly trivial change. When producing 64-bit executables (row A-3), there is no penalty for the 64-bit integer to floating point conversion, and the new implementation is slightly slower. However, it still has the benefit of returning $U(0,1)$ instead of $U[0,1)$; making a similar adjustment to the 32def and 52def versions to exclude zero makes them slightly slower. Finally, while DLF202 appears to be faster when compared to the default versions, this benefit disappears in most cases when using the new conversion.

Table I. Speed comparison: seconds of CPU time to generate and sum 10^9 random numbers

	MWC8222				MT19937				DLF202
	32def	32new	52def	52new	32def	32new	52def	52new	
P-1	33.7	17.5	63.4	28.5	33.0	15.0	54.2	22.8	29.2
P-2	29.3	14.7	63.2	23.9	31.7	15.2	85.1	23.4	28.2
A-2	17.9	9.3	31.2	16.7	20.2	12.0	38.3	18.7	22.5
A-3	7.7	7.1	10.6	9.9	11.1	11.3	19.6	18.2	16.3

A: AMD Athlon 64 3400 (2.4Ghz)/Linux Fedora Core 3 64-bit
P: Intel Pentium 4 (2.8GHz)/Microsoft Windows XP SP2
1: Microsoft Visual C++ 6 SP6 with settings: /O2
2: GCC 3.4.2: -O2 -m32 (32-bit executable)
3: GCC 3.4.2: -O2 -m64 (64-bit executable)

3. CONCLUSION

Perhaps because it seems so trivial, conversion of integer random number generators to floating point is almost always implemented as an afterthought. However, the implementation put forward here saw speed increases of around 50% in some cases. These will be hardware dependent, and shrink with more clever compilers. Nonetheless, the results here are relevant when benchmarking different random number generators. Conversion to $U(0,1)$ instead of $U[0,1)$ was also addressed without speed penalties. The alternative suggestion frequently found in the literature is to test for zero, which is much less attractive. There is a speed penalty of using 52-bit precision in double precision, but that may well be worth it, considering that it could be less costly than inefficient use of 32-bit precision. The results are also relevant when comparing the speed of old-style linear congruential generators with period 2^{31} to that of the new style RNGs.

REFERENCES

L'Ecuyer, P. 1996. Maximally equidistributed combined tausworthe generators. *Math. Comput. 65*, 203–213.

Marsaglia, G. 2003a. Re: good C random number generator. Posting, Usenet newsgroup sci.lang.c. 13-May-2003.

Marsaglia, G. 2003b. Seeds for random number generators. *Commun. ACM 46*, 90–93.

Marsaglia, G. and Tsang, W. W. 2004. The 64-bit universal RNG. *Statist. & Probab. Letters 66*, 183–187.

Matsumoto, M. and Nishimura, T. 1998. Mersenne Twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. Model. Comput. Simul. 8*, 3–30.

Panneton, F., L'Ecuyer, P., and Matsumoto, M. 2006. Improved long-period generators based on linear recurrences modulo 2. *ACM Trans. Math. Soft. 32*, 1–16.

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. 1993. *Numerical Recipes in C*, 2nd ed. Cambridge University Press, New York.