Random Number Generation

Chris Lomont, www.lomont.org

Introduction

This article is an introduction to random number generators (RNGs). The main goal is to present a starting point for programmers needing to make decisions about RNG choice and implementation. A second goal is to present better alternatives for the ubiquitous Mersenne Twister (MT). A final goal is to cover various classes of random number generators, providing strengths and weaknesses of each.

Random Number Generation

Background

Random number generators (RNGs) are essential to many computing applications. For some problems algorithms employing random choices perform better than any known algorithm not using random choices[1]. It is often easier to find an algorithm to solve a given problem if randomness is allowed.

Most random numbers used in computing are not considered truly random, but are created using Pseudo-Random Number Generators (PRNGs). PRNGs are deterministic algorithms, and are the only type of random number that can be algorithmically generated without an external source of entropy, such as thermal noise or user movements.

Designing good RNGs is hard and best left to professionals. Like cryptography, the history of RNGs is littered with bad algorithms and the consequences of using them. A few historical mistakes are covered near the end of this article.

Uses

Random numbers are used in many applications[2], including

- AI algorithms like genetic algorithms and automated opponents.
- Random game content and level generation.
- Simulation of complex phenomena such as weather and fire.
- Numerical methods such as Monte-Carlo integration.
- Until recently primality proving used randomized algorithms.
- Cryptography algorithms such as RSA use random numbers for key generation.
- Weather simulation and other statistical physics testing.

---

[1] The class of problems efficiently solvable on a (Turing) machine equipped with a random number generator is BPP, and it is an open problem if BPP=P, P being the class of problems efficiently solvable on a computer without random choice.

[2] Robert R. Coveyou of Oak Ridge National Laboratory humorously once titled an article, “The generation of random numbers is too important to be left to chance.”

- Optimization algorithms use random numbers significantly: simulated annealing, large space searching, and combinatorial searching.

Hardware RNGs

Since an algorithm cannot create "true" random numbers[3], many hardware based RNGs have been devised. Quantum mechanical events cannot be predicted, and are considered a very good source of randomness. Such quantum phenomena include:

- Nuclear decay detection, similar to a smoke detector.
- Quantum mechanical noise source in electronic circuits called "shot noise".
- Photon streams through a partially silvered mirror.
- Particle spins created from high energy x-rays.

Other sources of physical randomness are

- Atmospheric noise[4].
- Thermal noise in electronics.

Other physical phenomena are often used on computers, like clock drift, mouse and keyboard input, network traffic, add-on hardware devices, or images gathered from moving scenery. Each source must be analyzed to determine how much entropy the source has, and then how many high-quality random bits can be extracted.

Here are a few websites offering random bits and the method used to obtain them:

- http://random.org/ - atmospheric noise.
- http://www.fourmilab.ch/hotbits/ - radioactive decay of Cæsium-137.
- http://www.lavarnd.org/ - noise in CCD image.

Pseudo-random Number Generators (PRNGs)

PRNGs generate a sequence of "random" numbers using an algorithm, operating on an internal state. The initial state is called the *seed*, and selecting a good seed for a given algorithm is often difficult. Often the internal state is also the returned value. Due to the state being finite, the PRNG will repeat at some point, and the *period* of a RNG is how many numbers it can return before repeating. A PRNG using n bits for its state has a period of at most 2^n. Starting a PRNG with the same seed allows repeatable random sequences, which is very useful for debugging among other things. When a PRNG needs a "random" seed, often sources of entropy from the system or external hardware are used to seed the PRNG.

---

[3] "Anyone who considers arithmetical methods of producing random numbers is, of course, in a state of sin." - John von Neumann

[4] See www.freewebs.com/pmutaf/iwrandom.html for a way to get random numbers from WiFi noise.

Due to computational needs, memory requirements, security needs, and desired random number "quality," there are many different RNG algorithms. No one algorithm is suitable for all cases, in the same way that no sorting algorithm is best in all situations. Many people default to C/C++ `rand()` or the Mersenne Twister, both of which have their uses. Both are covered below.

Common Distributions

Most RNGs return an integer selected uniformly from the range $[0, m]$ for some maximum value m. C/C++ implementations provide the `rand()` function, with m being `#defined` as `RAND_MAX`, quite often the 15 bit value 32767. `srand(seed)` sets the initial seed, often using the current time using `srand(time(NULL))` as an entropy source. Most C/C++ `rand()` functions are Linear Congruential Generators, which are poor choices for cryptography. Most C/C++ implementations (as well as other languages) generate poor quality random numbers exhibiting various kinds of bias.

The most common distribution used in games is a *uniform distribution*, where equally likely random integers are needed in a range $[a,b]$. A common **mistake** is to use C code like `(rand()%(b-a+1)) + a`. The mistake is that not all values are equally likely to occur due to modulus wrapping around. This only works if $b-a+1$ divides `RAND_MAX+1`. For example, if `RAND_MAX` is 32767, then trying to generate numbers in the range [0,32766] using this method causes 0 to be twice as likely as any other value. A valid (although slower) solution is to chop to a multiple of the range in $[a,b]$, using:

```
int z, c = RAND_MAX / (b-a+1); // must ensure these operations
int c *= b-a+1;                // do not overflow!
do
   {
   z = rand( );
   } while( z >= c); // require z uniformly in [0,b-a]
return (z % (b-a+1)) + a;
```

The second most commonly used distribution is a *Gaussian Distribution*, which can be generated from a uniform distribution. Let `randf()` return uniformly distributed real numbers in [0,1]. Then the polar form of the Box-Muller transformation gives two Gaussian values `y1` and `y2` per call.

```
float x1, x2, w, y1, y2;
do {
   x1 = 2.0 * randf() - 1.0;
   x2 = 2.0 * randf() - 1.0;
   w  = x1 * x1 + x2 * x2;
   } while ( w >= 1.0 );
w = sqrt( (-2.0 * log( w ) ) / w );
y1 = x1 * w;
y2 = x2 * w;
```

Boost [Boost07] documents techniques for generating other distributions starting with a uniform distribution.

Randomness Testing

To test if a sequence is "random," a definition of "random" is needed. However "randomness" is very difficult to make precise. In practice (since many PRNGs are useful) tests have been designed to test the quality of RNGs by detecting sequence behavior that does not behave like a random sequence should.

The most famous randomness-testing suite is DIEHARD [Marsaglia95], made of twelve tests[5]. DIEHARD has been expanded into the open source (GPL) set of tests DieHarder [Brown06], which includes the DIEHARD tests as well as adding many new ones. Also included are many RNGs and a harness to add new ones easily. A third testing framework is TestU01 [L'Ecuyer06]. Each framework provides some assurance a tested RNG is not clearly bad.

Software Whitening

Many sources of random bits have some bias or bit correlation, and methods to remove the bias and correlation are known as whitening algorithms. Some choices:

- John von Neumann: Take bits two at a time, discard 00 and 11 cases, and output 1 for 01 and 0 for 10, removing uniform bias, at the cost of needing more bits.
- Flip every other bit, removing uniform bias.
- XOR with another known good source of bits, such as Blum Blum Shub.
- Apply cryptographic hashes like Whirlpool or RIPEMD-160. Note MD5 is no longer considered secure.

These whitened streams should still not be considered a secure source of random bits without further processing.

Non-cryptographic RNG Methods

Non-cryptographically secure methods are usually faster than cryptographic methods, but should **not** be used when security is needed, hence the classification. Each of the following is a PRNG with output sequence X_n. Some have a hidden internal state S_n from which X_n is derived. Either X_0 or S_0 is the seed, as appropriate.

Middle Square Method

This was suggested by John von Neumann in 1946: take a 10 digit number as a seed, square it, and return the middle 10 digits as the next number and seed. It was used in ENIAC, is a poor method with statistical weaknesses, and is no longer used.

Linear Congruential Generator (LCG)

These are the most common methods in widespread use, but are slowly being replaced by newer methods. They are computed with $X_{n+1} = (aX_n + b) \bmod m$, for constants a and b. The modulus m is often chosen as a power of 2 making it efficiently implemented as a

[5] http://en.wikipedia.org/wiki/Diehard_tests

bitmask. Careful choice of a and b is required to guarantee maximal period and avoid other problem cases. LCGs have various pathologies, one of which is that choosing points in 3-tuples and plotting them in space shows the points fall onto planes, as exhibited later in the section on RANDU, and is a result of linear relations between successive points. LCGs with power-of-two modulus m = 2^e are known to be badly behaved, especially in their least significant bits [L'Ecuyer90]. For example *Numerical Recipes in C* [Press06] recommends a = 1664525, b = 1013904223, m = 2^32, and the lowest order bit then merely alternates.

LCGs strengths are they are relatively fast and use a small state, making them useful in many places including embedded applications. If the modulus is not a power of two then the modulus operation is often expensive.

Writing a LCG as as LCG(m,a,b), Table 1 shows some LCGs in use.

LCG	Use
LCG(2^31, 65539, 0)	the infamous RANDU covered below.
LCG(2^24, 16598013, 12820163)	Microsoft VisualBasic 6.0.
LCG(2^48, 25214903917, 11)	drand48 from the Unix standard library; was used in java.util.Random.
LCG(10^12 − 11, 427419669081, 0)	Used in Maple 9.5 and in MuPAD 3. Replaced by MT19937 (below) in Maple 10.

Table 1 - Some LCGs in use

Truncated Linear Congruential Generator (TLCG)

These store an internal state S_i updated using a LCG, which in turn is used to generate the output X_i. Symbolically, $S_{n+1} = (aS_n + b) \bmod m$, $X_{n+1} = Floor\left[\frac{S_{n+1}}{K}\right]$. This allows using the fast m as a power of two but avoids the poor low order bits in the LCGs. If K is a power of 2, then the division is also fast. This algorithm is used extensively throughout Microsoft products (likely as a result of being compiled with VC++), including VC++ `rand()`, with the implementation

```
/* MS algorithm for rand() */
static unsigned long seed;
seed = 214013L * seed + 2531011L;
return (seed>>16)&0x7FFF; // return bits 16-30
```

This is not secure. In fact, for a cryptographic analysis project, this author has determined only three successive outputs from this algorithm are enough to determine the internal state (up to an unneeded most significant bit), and thereby know all future output. A simple way to compute the state is to notice the top bit of the state has no bearing on future output; so only 31 bits are unknown. The first output gives 15 bits of the state, leaving 17 bits unknown. Now, given two more outputs, take the first known 15 bits and test each of the possible 2^17 unknown bit states to see which gives the other two known

outputs. This provably determines the internal state. Two outputs are not enough since they do not uniquely determine the state.

Borland C++ and TurboC also used TLCGs with a =22695477 and b =1. Although the C specification does not force a `rand` implementation, the example one in the "C Programming Language" [Kernighan91] is a TLCG with a =113515245 and b =12345, with a `RAND_MAX` of the minimum allowable 32767.

Linear Feedback Shift Register (LFSR)

A Linear Feedback Shift Register (LFSR, Figure 1) generates bits from an internal state by shifting them out, one at a time. New bits are shifted into the state, and are a linear function of bits already in the state. LFSRs are popular because they are fast, easy to do in hardware, and can generate a wide range of sequences. Tap sequences can be chosen to make an n bit LFSR have period $2^n - 1$. Given $2n$ bits of output the structure and feedback connections can be deduced, so they are definitely not secure.

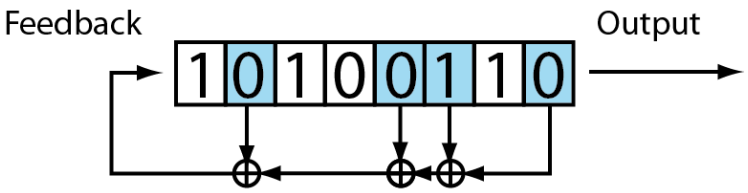

Figure 1 - LFSR

Inversive Congruential Generator

These are similar to LCGs but are nonlinear, using $X_{n+1} = \left(aX_n^{-1} + b\right) \bmod m$, where X_n^{-1} is the multiplicative inverse $\bmod m$, that is, $X_n X_n^{-1} \equiv 1 \left(\bmod m\right)$. These are expensive to compute due to the inverse operation, and are not often used.

Lagged Fibonacci Generator (LFG)

LFGs use k words of state $X_n = \left(X_{n-j} \otimes X_{n-k}\right) \bmod m$, $0 < j < k$ where $\otimes$ is some binary operation (plus, times, xor, others). These are very hard to get to work well and hard to initialize. The period depends on a starting seed and the space of reached values breaks into hard to predict cycles. They are now disfavored due to the Mersenne Twister and later generators. Boost [Boost07] includes variants of LFGs.

Cellular Automata

Mathematica prior to Version 6.0 uses the cellular automata Wolfram rule 30 to generate large integers[6]. Version 6.0 uses a variety of methods.

Linear Recurrence Generators

These are a generalization of the LFSRs, and most fast modern PRNGs are derived from these over binary finite fields. Note that none of these pass linear recurrence testing due

[6] http://mathworld.wolfram.com/Rule30.html

to being linear functions. The next few are special examples of this type of PRNG, and are considered the best general purpose RNGs.

Mersenne Twister

In 1997 Makoto Matsumoto and Takuji Nishimura published the Mersenne Twister algorithm [Matsumoto98], which avoided many of the problems with earlier generators. They presented two versions, MT11213 and MT19937, with periods of 2^11213-1 and 2^19937-1 (approximately 10^6001), which represents far more computation than is likely possible in the lifetime of the entire universe. MT19937 uses an internal state of 624 longs, or 19968 bits, which is about expected for the huge period. It is (perhaps surprisingly) faster than the LCGs, is equidistributed in up to 623 dimensions, and has become the main RNG used in statistical simulations. The speed comes from only updating a small part of the state for each random number generated, and moving through the state over multiple calls.

Mersenne Twister is a Twisted Generalized Feedback Shift register (TGFSR). It is not cryptographically secure: observing 624 sequential outputs allows one to determine the internal state, and then predict the remaining sequence.

Mersenne Twister has some flaws, covered in the WELL algorithm below.

LFSR113, LFSR258

[L'Ecuyer99] introduces combined LFSR Tausworthe generators LFSR113 and LFSR258 designed specially for 32-bit and 64-bit computers, respectively, with periods of approximately 2^113 and 2^258, respectively. They are fast, simple, and have a small memory footprint. For example, here is C/C++ code for LFSR113 that returns a 32-bit value:

```
unsigned long z1, z2, z3, z4; /* the state  */
/* NOTE: the seed MUST satisfy
      z1 > 1, z2 > 7, z3 > 15, and z4 > 127 */
unsigned long lfsr113(void)
   { /* Generates random 32 bit numbers.    */
   unsigned long b;
   b  = (((z1 << 6) ^ z1)   >> 13);
   z1 = (((z1 & 4294967294) << 18) ^ b);
   b  = (((z2 << 2) ^ z2)   >> 27);
   z2 = (((z2 & 4294967288) <<  2) ^ b);
   b  = (((z3 << 13) ^ z3)  >> 21);
   z3 = (((z3 & 4294967280) <<  7) ^ b);
   b  = (((z4 << 3) ^ z4)   >> 12);
   z4 = (((z4 & 4294967168) << 13) ^ b);
   return (z1 ^ z2 ^ z3 ^ z4);
   }
```

Since 2^113 is approximately 10^34, this already represents a huge number of values, and has a much smaller footprint than MT19937. The LFSR generators also are well equidistributed, and avoid LCGs problems.

WELL Algorithm

Matsumoto (co-creator of the Mersenne Twister), L'Ecuyer (a major RNG researcher), and Panneton introduced another class of TGFSR PRNGs in 2006 [Panneton06]. These algorithms produce numbers with better equidistribution than MT19937 and improve upon "bit-mixing" properties. WELL stands for "Well Equidistributed Long-period Linear," and they seem to be better choices for anywhere MT19937 is currently used. They are fast, come in many sizes, and produce higher quality random numbers.

WELL period sizes are presented for period 2^n for n = 512, 521, 607, 800, 1024, 19937, 21701, 23209, and 44497, with corresponding state sizes. This allows a user to trade period length for state size. All run at similar speed. 2^512 is about 10^154, and it is unlikely any video game will ever need that many random numbers, since it is far larger then the number of particles in the universe. The larger periods ones aren't really needed except for computation like weather modeling or earth simulations. A standard PC needs over a googol[7] of years to count to 2^512.

A significant place the WELL PRNGs perform better than MT19937 is in escaping states with a large number of zeros. If MT19937 is seeded with many zeros, or somehow falls into such a state, then the generated numbers have heavy bias towards zeros for a many iterations. The WELL algorithms behave much better, escaping zero bias states quickly.

The only downside is that they are slightly slower than MT19937, but not much. The upside is the numbers are considered to be higher quality, and the code is significantly simpler. Here is WELL512 C/C++ code written by the author and placed in the public domain[8]. It is about 40% faster than the code presented on L'Ecuyer's site, and is about 40% faster than MT19937 presented on Matsumoto's site.

```
/* initialize state to random bits  */
static unsigned long state[16];
/* init should also reset this to 0 */
static unsigned int index = 0;
/* return 32 bit random number      */
unsigned long WELLRNG512(void)
      {
      unsigned long a, b, c, d;
      a  = state[index];
      c  = state[(index+13)&15];
      b  = a^c^(a<<16)^(c<<15);
      c  = state[(index+9)&15];
      c ^= (c>>11);
      a  = state[index] = b^c;
      d  = a^((a<<5)&0xDA442D24UL);
      index = (index + 15)&15;
      a  = state[index];
      state[index] = a^b^d^(a<<2)^(b<<18)^(c<<28);
      return state[index];
      }
```

[7] Googol = 10^100. Google it.

[8] However, if you use it, I'd appreciate a reference or at least an email with thanks!

Cryptographic RNG Methods

Cryptographically Secure PRNGs (CSPRNGs) make it hard for an attacker to deduce the internal state of the generator or to predict future output given large amounts of output. Several CSPRNGs have been standardized and can be found online[9]. Two RFCs[10] dealing with randomness requirements for security are RFC1750 and RFC4086. Any implementation of these methods has to be done very carefully to avoid many pitfalls. Whenever possible use an implementation from a trusted and competent source.

Blum Blum Shub

Published in 1986 by Lenore Blum, Manuel Blum and Michael Shub, Blum Blum Shub [Blum86] is considered a secure PRNG. It is computed via $S_{n+1} = \left(S_n^2\right) \bmod m$ where $m = pq$ for two properly chosen large primes p, q. Then the output X_{n+1} is some function on S_{n+1}, which often is taken as bit parity or some particular bits of S_{n+1}. Its strength relies on the hardness of integer factoring, which is the same problem RSA public key encryption relies on for security[11]. Blum Blum Shub is only useful for cryptography, since is it much slower than the non-cryptographic PRNGs.

ISAAC, ISAAC+

[Jenkins96] introduced ISAAC, a CSPRNG based on a variant of the RC4 cipher. It is relatively fast for a CSPRNG, requiring an amortized 18.75 instructions to produce a 32-bit value. There are no cycles in ISAAC shorter than 2^{40} values, and the expected cycle length is 2^{8295} values. ISAAC-64, a version for 64-bit machines, requires 19 instructions to produce a 64-bit result.

/dev/random

Although not a specific algorithm, Linux and many Unix flavors implement a source of randomness in /dev/random which returns random numbers based on system entropy, so it is considered a true random number generator. /dev/random blocks, that is, does not return until enough entropy has been gathered to satisfy the request. As a result, many programs use the non-blocking /dev/urandom. However these numbers are not as secure, and use of /dev/urandom depletes system entropy, allowing some attacks on bad implementations. The underlying algorithm is not specified; some systems use Yarrow as mentioned below.

[Gutterman06] revealed exploitable weaknesses in the Linux implementation at the time, which should have been fixed by now. Overall /dev/random is the preferred place on Linux to get CSPRNGs.

---

[9] http://en.wikipedia.org/wiki/CSPRNG
[10] www.ietf.org/rfc
[11] Note Shor's quantum factoring algorithm factors integers efficiently, so once quantum computers are in use Blum Blum Shub will become insecure.

Microsoft's CryptGenRandom

Microsoft's CryptoAPI function CryptGenRandom function fills a buffer with cryptographically secure random bytes. Like /dev/random it is considered a true random number generator. Although closed source, it is FIPS validated, and is considered secure. This author is unaware of any weaknesses with recent implementations. On Windows, it is the preferred source of CSPRNGs.

Yarrow

[Kelsey99] introduces Yarrow, which uses system entropy to generate random numbers. It is explicitly unpatented and royalty-free, and no license is required to use it. Yarrow is used in Mac OS X and FreeBSD to implement /dev/random. Yarrow is no longer supported by the designers, who have released an improved design titled Fortuna.

Fortuna

Fortuna is another CSPRNG from the book Practical Cryptography [Ferguson03]. The generator is based on any good block cipher, and encrypts in counter mode, encrypting successive values of a counter. The key is changed periodically to prevent some statistical weaknesses. It uses entropy pools that gather information from random sources available to the system, and is considered a true RNG since it uses external entropy.

Common Mistakes

Knuth Example

The history of RNGs is scattered with examples of bad design. Even algorithm master Donald Knuth tells a story in [Knuth98] about trying his hand at making a random number generator by creating a "Super-random" generator. His first run settled onto a 10-digit number that then repeated forever. His second run began to repeat itself after 7401 values with a cycle of 3178. So creation of good RNGs is not trivial.

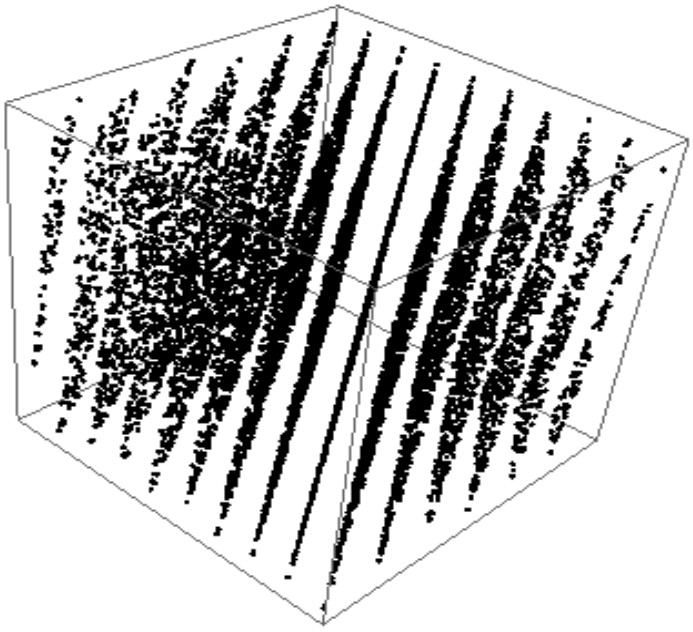

Figure 2. LCG Bias

Here are a few more examples that hopefully will dissuade people from using homemade RNGs in critical applications.

RANDU

RANDU is an infamous LCG used since the 1960s; it is LGC(2^31,65539,0), and requires an odd initial seed. The constants were chosen for easy and fast implementation. As all LCGs, it suffers from linear relations between successive numbers. Figure 2 shows the output of 10,000 triplets (x, y, z) plotted in 3D, which happen to fall into planes.

Netscape

An early version of Netscape needed a CSPRNG, but seeded it with three values that weren't very well spread out (time of day, process ID, and parent process ID) and used the result for cryptography. [Goldberg96] published a successful attack on Netscape's SSL protocol, with the exploitable flaw being a poor choice of seed.

Folklore Algorithms

The author encountered a folklore algorithm from a game programmer around 1992, who explained that he had a fast and simple PRNG for his NES code. The basic idea was to shift bits out of a seed, and whenever the seed had 1 bit about to shift off, exclusive or in a constant. This was fast and looked nice in assembly, but being a skeptic this author thought about the claim of randomness, and said this should produce numbers that tend to decrease, and every so often jump back up, making saw tooth outputs. This led the original programmer to discover a bug in his AI that was using the PRNG which he

hadn't suspected (he had used it the PRNG for years). Although anecdotal, it is wise to test a new RNG and see that it behaves as expected before committing it to your toolbox.

One last particularly funny example is the xkcd webcomic version of a random number generator at http://xkcd.com/c221.html, reproduced for your viewing pleasure:

```
int getRandomNumber()
      {
      return 4; // chosen by fair dice roll.
                // guaranteed to be random.
      }
```

Code

There are many online places to obtain source code for the algorithms covered in this article. Boost [Boost07] contains high quality implementations for many of them, and Wikipedia contains more information and links to most of the presented topics. L'Ecuyer's webpage (www.iro.umontreal.ca/~lecuyer/papers.html) is a good source of papers and many implementations. In addition, Technical Review 1 (TR1) for the C++ language includes many distributions and generators (including MT19337), so it is likely C++ will someday have some of these features built in.

Conclusion

This article has provided basics of RNGs, including many common algorithms. LFSR113, LSFR258, and the WELL generators offer better choices than the Mersenne Twister for many applications, and this presentation brings knowledge of them to a wider audience. Strengths and weaknesses were presented for algorithms where possible. Knowledge about RNG types and when to apply them should be in the toolkit of any serious developer, just as any serious developer should know multiple sorting algorithms, or numerous tree structures. Hopefully this article provides a base and reference for such knowledge.

History

2008 – Original Release.
2011 – Fix to the WELL512 code: corrected 0xDA442D20 to 0xDA442D24.
– Fix to incorrect uniform [a,b] routine on page 3.

References

[Blum86] Blum, Lenore, Manuel Blum, and Michael Shub. "A Simple Unpredictable Pseudo-Random Number Generator", SIAM Journal on Computing, volume 15, pp. 364–383, May 1986.
[Brown06] Brown, Robert G., and Dirk Eddelbuettel , "DieHarder: A Random Number Test Suite Version 2.24.4", www.phy.duke.edu/~rgb/General/rand_rate.php
[Boost07] "The Boost C++ Library," 2007, www.boost.org.

[Ferguson03] Ferguson, Niels, and Bruce Schneier, *Practical Cryptography*, Wiley, 2003. ISBN 0-471-22357-3.
[Goldberg96] Goldberg, Ian, and David Wagner, "Randomness and the Netscape Browser," Dr. Dobb's Journal, January 1996, pp. 66-70. www.cs.berkeley.edu/~daw/papers/ddj-netscape.html
[Gutterman06] Gutterman, Pinkas, and Reinman, "Open to Attack: Vulnerabilities of the Linux Random Number Generator," March 2006, Black Hat 2006, www.pinkas.net/PAPERS/gpr06.pdf.
[Jenkins96] Jenkins, Bob, "ISAAC and RC4," www.burtleburtle.net/bob/rand/isaac.html as of 2007.
[Kelsey99] Kelsey, J., B. Schneier, and N. Ferguson "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator," Sixth Annual Workshop on Selected Areas in Cryptography, Springer Verlag, August 1999, www.schneier.com/paper-yarrow.html.
[Kernighan91] Kernighan, B., and Dennis Ritchie, *The C Programming Language, Second Edition*, Prentice-Hall, 1991.
[Knuth98] Knuth, Donald, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition*, Addison-Wesley, 1998.
[L'Ecuyer90] L'Ecuyer, P., "Random Numbers for Simulation", Communications of the ACM, 33 (1990), pp. 85-98, www.iro.umontreal.ca/~lecuyer/papers.html.
[L'Ecuyer99] L'Ecuyer, P., "Tables of Maximally-Equidistributed Combined LFSR Generators", Mathematics of Computation, 68, 225 (1999), 261-269. www.iro.umontreal.ca/~lecuyer/papers.html.
[L'Ecuyer06] L'Ecuyer, P. and R. Simard, "TestU01: A C Library for Empirical Testing of Random Number Generators", May 2006, Revised November 2006, ACM Transactions on Mathematical Software, 33, 4, Article 1, December 2007, to appear. www.iro.umontreal.ca/~lecuyer/papers.html
[Marsaglia95] Marsaglia, George. DIEHARD, http://www.csis.hku.hk/~diehard/
[Matsumoto98] Matsumoto, M., and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. Model. Comput. Simul. 8, 3 (1998). www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html.
[Panneton06] Panneton, F. P. L'Ecuyer, and M. Matsumoto, "Improved Long-Period Generators Based on Linear Recurrences Modulo 2", ACM Transactions on Mathematical Software, 32, 1 (2006), 1-16. www.iro.umontreal.ca/~lecuyer/papers.html
[Press06] Press, William H. (Editor), Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, *Numerical Recipes in C++: The Art of Scientific Computing*, Cambridge University Press; 2 edition, 2002.