TinyRNG - a simplistic random number generation and transformation library

Documentation

Guido Klingbeil

May 23, 2013

TinyRNG is a very small and simplistic random number generation and transformation library implemented in C. It is built around the two very simple random number generators (RNGs) XORSHIFT and KISS (Keep It Simple Stupid) designed by George Marsaglia. Both pass the BigCrunch battery test of the TestU01 testsuite by L'Ecuyer and Simard [12].
TinyRNG generates random numbers with 32-bit or 64-bit of randomness and provides transformation into samples from the following distributions:

- uniform $[0, 1)$,
- exponential,
- standard normal,
- normal,
- binomial,
- Poisson,
- Gamma,
- Beta.

TinyRNG uses a strict *call by reference* design. Each function returns the integer 0 on success and a non-zero error code defined in TinyRNG.h if it fails. The generated sample is written to an address provided by the user as a C pointer.
The library consists of two distinct parts: (i) the pseudo RNG generating 32-bit or 64-bit random integers, (ii) the transformation algorithms. To make the addition of new RNGs simple, the RNGs are passed as function pointers to the transformation algorithms. The example below demonstrates this. A random integer generated by the 32-bit XORSHIFT RNG xorshift32 is transformed into a uniformly distributed random sample on the interval $[0, 1)]$ by calling unifrnd32 with the generator xorshift32 as a parameter:

```
float sample = 0.0f;
uint32_t err = 0;
uint32_t *seeds;

// allocate the seed memory
seeds = (uint32_t*)malloc(5 * sizeof(uint32_t));

// seed the RNG
err = seed(seeds, 5);

// generate a 32-bit uniform [0,1) random sample using
  the XORSHIFT RNG
err = unifrnd32(xorshift32, seeds, &fsample);

// free the allocated memory
free(seeds);
```

Listing 1: Example code generating a uniform random sample on the interval $[0, 1)$.

A design goal of TinyRNG was to make it thread safe. The state vector seeds of the RNGs are passed as a pointer to the RNGs. Furthermore, non of the transformation algorithms maintains an internal state between function calls.
While many simulation tasks require a reliable random number generator (RNG), implementing such a generator is not the main focus of many researchers. However, a simple and at the code level understandable RNG seems to be desirable. The RNG design is chosen to minimise the error possibilities: (i) the implementation is as minimal as possible, (ii) the dependencies between transformations are kept minimal, all of them are transformations of uniform random numbers or normal random numbers. The uniform RNG is tested rigorously with the *TestU01* test suite by [12]. Our implementations of the transformations are tested by generating 10000 using our implementation and the same number of samples from the same distribution using MATLAB's RNG and testing both sets of random samples using the Kolmogorov-Smirnov-test [8].

Uniform random number generator

The uniform RNG generates a sequence of samples uniformly distributed $U(0, 1)$ on the interval $(0, 1]$ which can easily be re-scaled to the interval $(a, b]$ where $-\infty < a < b < \infty$. The implemented uniform RNG is a member of the XORSHIFT family of generators proposed by [15]. XORSHIFT generators are a special form of the linear feedback shift register (LFSR) generators [3, 19]. They are compact, only requiring a very few lines of source-code, and only bit-shift and integer addition operations which are efficiently implemented in hardware. This section gives a very brief sketch of the XORSHIFT generator's design idea. The starting point is the linear recurrence for fixed values $r > s > 0$:

$$x_k = x_{k-r}\mathbf{A} + x_{k-s}\mathbf{B} \ , \tag{1}$$

where x_k is a binary vector, consisting of $\{0,1\}$, of length w and $\mathbf{A}$ and $\mathbf{B}$ are $w \times w$ binary matrices. w is the word length and is typically chosen to match the computer's word length ($w = 32$ or $w = 64$).

Computing vector matrix products is usually regarded as computationally expensive. The core idea of Marsaglia [15] is to lower the computational burden by assuming that $\mathbf{A}$ and $\mathbf{B}$ are products of small terms $(\mathbf{I}+\mathbf{L}^a)$ and $(\mathbf{I}+\mathbf{R}^b)$ where $\mathbf{L}$ and $\mathbf{R}$ are $w \times w$ shift matrices. The left shift matrix $\mathbf{L}$ is a binary matrix such that $(x_1, \ldots, x_w)\mathbf{L} = (x_2, \ldots, x_w, 0)$, and the right shift matrix $\mathbf{R}$ is such that $(x_1, \ldots, x_w)\mathbf{R} = (0, x_2, \ldots, x_{w-1})$. Since, $\mathbf{R} = \mathbf{L}^T$ it is sufficient to give $\mathbf{L}$:

$$\mathbf{L} = \begin{pmatrix} 0 & 0 & \ldots & 0 \\ 1 & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \ldots & 1 & 0 \end{pmatrix} .$$

Marsaglia [15] assumes:

$$\mathbf{A} = (\mathbf{I}+\mathbf{L}^a)(\mathbf{I}+\mathbf{R}^b) \ ,$$

and

$$\mathbf{B} = (\mathbf{I}+\mathbf{R}^c) \ ,$$

where a, b, and c are small positive integers. The recurrence given in Equation (1) can be rewritten as:

$$x_k = x_{k-r}(\mathbf{I}+\mathbf{L}^a)(\mathbf{I}+\mathbf{R}^b) + x_{k-s}(\mathbf{I}+\mathbf{R}^c) \ .$$

Note that $x(I+\mathbf{L}^a) = x + x\mathbf{L}^a$ is the sum of x with a shifted version of itself. Since x is a binary vector, this translates into the C programming language as x = x ^(x <<a) where x is the unsigned integer computer representation of x. $x(I+\mathbf{R}^b) = x + x\mathbf{R}^b$ is translated into x = x ^(x >>b) in the C programming language. $x(I+L^a)(I+L^b)$ is broken up into two statements in the C programming language x = x ^(x <<a); x = x ^(x >>b).

The recurrence Equation (1) can be rewritten as:

$$(x_{k-r+1}, x_{k-r+2}, \ldots, x_k) = (x_{k-r}, x_{k-r+1}, \ldots, x_{k-1})\mathbf{C} \ ,$$

where $\mathbf{C}$ is the $n \times n$ companion matrix, where $n = rw$. $\mathbf{C}$ is a block matrix consisting of $r \times r$ blocks of size $w \times w$:

$$\mathbf{C} = \begin{pmatrix} 0 & 0 & \dots & \mathbf{A} \\ \mathbf{I} & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & \mathbf{I} & \mathbf{B} \end{pmatrix} .$$

The intuitive computer implementation is a ring buffer or "circular array" of length r [11]. Furthermore, this gives us an intuitive understanding of the parameter r as the number of the RNG's internal state variables.
Marsaglia [15] chooses the parameter $s = 1$. Choosing the parameters a, b, and c appropriately, the XORSHIFT generator has the full period length p of $p = 2^{r \times w} - 1$. The implemented XORSHIFT has three combined XOR / SHIFT operations with parameters $a = 7$, $b = 13$, $c = 6$ and a period length of $2^{r \times w} = 2^{160} - 1 \approx 1.562 \times 10^{48}$ where $w = 32$ is the word length and $r = 5$ is the number of seed values. Even when running 10^4 threads in parallel and considering Knuth's (1997b) suggestion not to use more than $\frac{p}{1000}$, this gives a period length of greater than 10^{41} per thread.
While the XORSHIFT generators passes the *Diehard* battery tests [13] as well as the tests implemented by Marsaglia and Tsang [17], L'Ecuyer and Simard [12] and Panneton and L'Ecuyer [19] report that the XORSHIFT generators fail the TestU01 *Crush* test suite in large parts. However, it has to be noted that the XORSHIFT generator implemented by L'Ecuyer and Simard [12] with $w = 32$ and $r = 1$ has a period length of only $2^{32} - 1$. The *Crush* battery test of *TestU01* uses 2^{35} random numbers in 144 statistical tests. This means approximately 2^{33} per test. The *BigCrush* battery test generates 2^{38} random numbers in 160 statistical tests, or approximately 2^{36} per test. This is beyond the period length of 2^{32} of the XORSHIFT generator implemented and the generator is likely to fail [12].
The implemented XORSHIFT generator with a period length of $2^{160} - 1$ is tested with the same rigorus tests as L'Ecuyer and Simard [12], the *SmallCrush*, *Crush*, and *BigCrush* battery of tests, and passes all the tests.

Inversion method to sample exponential random numbers

Exponential distributed random samples are generated by the inversion method. Let X be a continuous random variable with the distribution function F_X. Now if $Y = F_X(X)$, then Y has a uniform distribution on $(0, 1)$. Thus if we replace X with the uniform distribution U, then X is given by [6]:

$$X = F_X^{-1}(U) \ ,$$

and has the distribution function F_X. This directly translates into the inversion method:

```
Sample $u$ from a uniform random variable;
$x = F^{-1}(U)$;
return $x$;
```

Listing 2: Inversion method to sample exponential random numbers [7].

The cumulative distribution function of an exponential distribution is:

$$F(x) = \begin{cases} 1 - e^{-\lambda x} & x \geq 0 \ , \\ 0 & x < 0 \ , \end{cases}$$

and its inverse is

$$F^{-1}(x) = \frac{-ln(1-x)}{\lambda} \ , x \in (0,1) \ .$$

Observing that if x is from a uniform distribution on the interval $(0,1)$, then so is $1-x$ [7, 11, 20]:

$$F^{-1}(x) = \frac{-\ln x}{\lambda} \ .$$

Normal distributed random numbers

The Box and Muller [2] transformation converts two samples from the standard normal distribution into two samples from the uniform distribution. Let U_1 and U_2 be two independent uniform random variables. The transformation is given by:

$$\begin{aligned} X_1 &= r\cos(\Theta) = \sqrt{-2\ln(U_1)}\cos(2\pi U_2) \ , \\ X_2 &= r\sin(\Theta) = \sqrt{-2\ln(U_1)}\sin(2\pi U_2) \ , \end{aligned}$$

where X_1 and X_2 are independent standard normal random variables. A simple way to grasp this transformation is by reversing it, showing that one can transform two standard normal samples into uniform ones. Assume that X_1 and X_2 are cartesian coordinates. In polar coordinates they can be writen as the length of the radius vector squared $r^2 = X_1^2 + X_2^2$ and the angle $\theta = \text{atan}(\frac{X_2}{X_1})$. The probability density on a circle for any given r is constant. This means that Θ is uniformly distributed in $(0, 2\pi]$ which can be easily rescaled to $(0,1]$. A χ-squared distribution is the sum of the squares of normal distributions and thus r^2 is χ-squared distributed with two degrees of freedom. A χ-squared distribution with two degrees of freedom is identical to an exponential distribution with $\lambda = \frac{1}{2}$ [4].
This means that one may rewrite r^2 and Θ as:

$$\begin{aligned} R^2 &= -2\ln U_1 \ , \\ \Theta &= 2\pi U_2 \ . \end{aligned}$$

```
Let $x = 0$;

for $i = 1$ to $n$ do:
    sampel $U$ from the uniform distribution $[0,1]$;

    if $U \leq p$ do:
        $x = x + 1$;
    end;
end;

return $x$;
```

Listing 3: Bernoulli method to sample from a binomial distribution.

Binomial distributed random numbers

To limit the complexity of the random number generation, a transformation is chosen that transforms uniform random numbers into binomial ones. For small values of n, the sum of n Bernoulli trials is computed. This method scales linearly with $\mathcal{O}(n)$ [11, 7].
For larger values of n, the geometric method by Devroye [6] is used. The run-time of the geometric method scales with $\mathcal{O}(np)$. While the inversion method is more efficient, it seems not to be suitable for the use on GPUs. The generator used exploits the waiting time property of the binomial distribution. Let G_i, $i = 1, \ldots, m$, be m independent geometric random variables with probability p. Furthermore let x be the smallest integer such that:

$$\sum_{i=0}^{x+1} G_i > n \ ,$$

Then x is a binomial random variable $\mathcal{B}(n,p)$. Each geometric random variable G_i gives the number of trials to the first success. This means that the sum $\sum_{i=0}^{x+1} G_i$ gives the total number of trials to the $(x+1)$-th success. The total number of trials exceeds n if and only if there are at most x successes in the first n Bernoulli trials, which is the definition of the binomial distribution [7].
This method requires samples from the geometric distribution. A sample from the geometric distribution with probability p can be generated in constant time from an exponential sample with $\lambda = \ln(1-p)$ by truncating it [7]:

$$F^{-1}(x) = \left\lceil \frac{\ln(x)}{\ln(1-p)} \right\rceil \ .$$

```
Let $y = 0$, $x = 0$, $c = \ln(1-p)$;

if $c == 0$ return 0;

```

```
do {
    sample u from a uniform distribution
    $y = y + \lfloor \ln(u)/c \rfloor + 1$; // integer addition
    $x = x + 1$;
} while($y \geq n$);

return $x - 1$;
```

Listing 4: Geometric method to sample from a binomial distribution by Devroye [6].

For certain parameter regimes the binomial distribution may be approximated by the Poisson and normal distributions. The approximation by the Poisson $\mathcal{P}(np)$ distribution works well if $n > 20$ and $np < 0.05$ (this implies $p < 0.0025$). However in this parameter regime the multiplication method sampling from the Poisson distribution has the same complexity as the Bernoulli method to sample from the Poisson distribution.
For large values of $n \geq 200$ and $np \geq 70$, the approximation by the normal distribution $\mathcal{N}(np, \sqrt{np(1-p)}$ seem to work well.

Rejection sampling

Let f be the probability density function of the distribution to be sampled from. Rejection sampling does not sample f but a utility distribution g with:

$$f(t) \leq c \cdot g(t) \ ,$$

where c is a constant. Compute a sample X from g. Let U be an independent uniform variate in the interval $(0, 1]$. Then if $U \geq \frac{f(X)}{c \cdot g(X)}$ reject X. If $U < \frac{f(X)}{c \cdot g(X)}$ accept X as having the desired density f [7, 9, 11].
As an example, assume x and y to be the axes of a two-dimensional cartesian coordinate system. The x-ordinate is a sample from the utility distribution X and the y-ordinate is a sample from the uniform distribution on $(0, 1]$.
Figure 1 shows this for sampling from a half-normal distribution $f(x) = \sqrt{\frac{2}{\pi}} \exp(-\frac{x^2}{2}),\ x \geq 0$ using and exponential utility distribution $g(x) = \exp(-x),\ x \geq 0$ and a constant $c = 0.7$.
On average, rejection sampling takes c iterations to accept a sample. This means that the smaller c is, the more efficient rejection sampling is [11].
Rejection sampling is used to sample from a Poisson and Gamma distribution.

Poisson distributed random numbers

The Poisson random number generation is divided into three domains depending on λ. For small values, $\lambda < 30$, the multiplicative method as described for example by Knuth [11] or Devroye [7] is used. The multiplicative method scales linearly with μ is thus only acceptable for small values of μ. The generator exploits straightforward that the inter-arrival times are from the exponential distribution with mean λ [11, 7]:

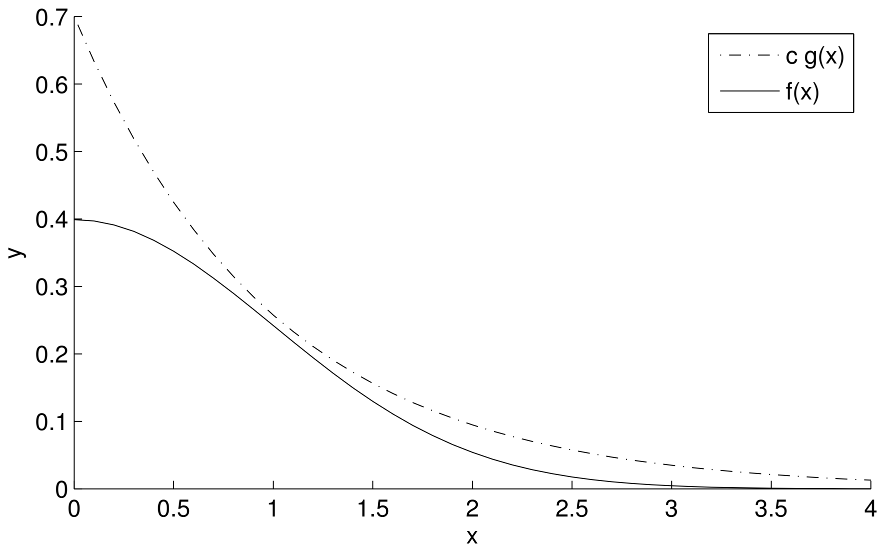

Figure 1: Rejection sampling of a half-normal distribution using an exponential distribution and a scaling constant $c = 0.7$.

```
Let $x = 0$, $sum = 0$;

while $sum < \lambda$ do:
    sample $e$ from the exponential random number $E$;
    $sum = sum + e$;
    $x = x + 1$;
end;

return $x$;
```

Listing 5: Multiplicative method to sample from a Poisson distribution [7].

If $30 < \lambda \leq 100$ then rejection sampling is applied [1]. The rejection method uses a linear approximation of μ^{-1}. Due to the chosen constants, this approximation deteriorates if $\mu < 30$. Numerical studies conducted by Atkinson [1] indicate that the run-time of the rejection method is independent of μ and for values $\mu > 30$ it indicates the rejection method is faster than the multiplication method.

Stirling's approximation

The rejection scheme proposed by Atkinson [1] requires the computation of $\ln(n!)$. Computing first the factorial and then the logarithm poses the problem that the factorial for even moderate values of n is out of the range of 32-bit or 64-bit value stored in a computer. However, one can compute $\ln(n!)$ directly based on Stirling's approximation [10]:

$$n! \simeq \sqrt{2\pi n}\ \left(\frac{n}{\mathrm{e}}\right)^n , \qquad n \to \infty \ .$$

Following Knuth [10], the Stirling sequence to compute $\ln(n!)$ can be written as:

$$\ln(n!) = (n+\frac{1}{2})\ln(n) - n + \sigma + \sum_{1<k\leq m} \frac{B_k(-1)^k}{k(k-1)n^{k-1}} + O(\frac{1}{n^m}) \ ,$$

where $B_k(n) = n^k - \sum_{m=0}^{k-1} \binom{m}{k} \frac{B_m(n)}{k-m+1}$ is the k-Bernoulli number, and σ a canstant with $e^\sigma = \sqrt{2\pi}$. This expression can be written as:

$$\ln(n!) = n\ln(n) + n + \frac{1}{2}\ln(2\pi n) + \sum_{1<k\leq m} \frac{B_k(-1)^k}{k(k-1)n^{k-1}} + O(\frac{1}{n^m}) \ .$$

Stirling's approximation is accurate for large n. This can be compensated for by pre-computing values for small n and stringing them in a look-up table or binary search tree. This gives the opportunity to balance pre-computation and estimation of $\ln(n!)$. Assuming a binary search tree of depth 5 storing $2^5 = 32$ pre-computed values, approximating $\ln(n!)$ by:

$$\ln(n!) \approx n\ln(n) + \tfrac{1}{2}\ln(2\pi n) - n + \frac{1}{12n} \ ,$$

gives an error smaller than $\frac{1}{360\cdot 33^3} \approx 7.73^{-8}$ which is on the order of single precision floating point machine precision and thus sufficient for single precision computations. Using:

$$\ln(n!) \approx n\ln(n) + \tfrac{1}{2}\ln(2\pi n) - n + \frac{1}{12n} - \frac{1}{360n^3} \ ,$$

the error is smaller than $\frac{1}{1260\cdot 33^5} \approx 2.02^{-11}$.
Another method to compute the $n!$ for large values of n is to use the gamma function Γ since $n! = \Gamma(n+1)$ [4]. Furthermore, many mathematical libraries including the CUDA mathematical libraries include functions to compute the logarithm of the gamma function $\ln(\Gamma(n))$. This function is called lgamma.
For large values of $\lambda > 100$ the Poisson distribution is approximated by a normal distribution. The approximation by the Central Limit Theorem (CLT) is $\Phi(\frac{k-\lambda}{\sqrt{\lambda}})$ where $\Phi(k)$ denotes the cumulative distribution function of the standard normal distribution. Often, the continuity correction $\Phi(\frac{k+\frac{1}{2}-\lambda}{\sqrt{\lambda}})$ is used [22]. However, while reducing the absolute error Molenaar [18] points out that the relative error in the tails is increased.

Gamma distributed random numbers

The Gamma distribution has a scale and shape parameter. For $a > 0$ the Gamma distribution is given by:

$$F(x) = \frac{a}{\Gamma(a)} \int_0^x t^{a-1} e^{-t} dt, \quad x \geq 0 \ .$$

The Γ distribution has an infinite peak at 0 for $a < 0$ and special care has to be taken. However, using a special case of Stuart's Theorem circumvents this problem and samples from $\Gamma_{a+1}$are sufficient to generate $\gamma_a = \gamma_{a+1} U^{\frac{1}{a}}$ where U is a uniform distributed random variable on $(0, 1]$. For values of $a > 1$ the *RNOR* generator by Marsaglia and Tsang [16] is implemented. This is based on the rejection method augmented by the squeeze method. The latter is a refinement of rejection sampling introduced by Marsaglia [14] to reduce the number of iterations until a sample is accepted. While the rejection function uses only one envelope above the distribution to be sampled, the squeeze method adds a second envelope below:

$$h_1(x) \leq f(x) \leq h_2(x) \ .$$

The generic rejection sampling algorithm with squeeze is given in Listing 6:

```
Let $accept = 0$;

do {
    sample u from a uniform distribution;
    sample $x$ from the distribution with density $g$;

    $w = u \cdot c \cdot g(x)$;

    $accept = (w \leq h_1(x))$;
    if $\neg accept$ then:
        if $w \leq h_2(x)$ then:
            $accept = (x \leq f(x))$;
        end
    end;
} while($\neg accept$);

return $x - 1$;
```

Listing 6: Rejection-sampling with squeeze [6].

Let Z and Y be independent random variables Γ_a distributed and $\beta_{b,a-b}$ distributed, respectively. Then $Z(1 - Y)$ is from the Γ_{a-b} distribution. The distribution $\beta_{1,a}$ is linked to the uniform distribution $1 - Y^{\frac{1}{a}}$ is uniformly distributed. Furthermore, if U is a uniform random number on $(0, 1]$, then $1 - U$ is also uniform on $(0, 1]$. Let $b = 1$ and substitute a with $a + 1$, then $ZU^{\frac{1}{a}}$ is also from the Γ_a distribution [21, 7].

Beta distributed random numbers

The Beta distribution has two shape parameters $\alpha > 0$ and $\beta > 0$. The Beta distribution is given by:

$$F(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B_{\alpha,\beta}} \ , \tag{2}$$

where

$$B_{\alpha,\beta} = \int_0^1 x^{\alpha-1}(1-x)^{\beta-1}dx = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)} \ . \tag{3}$$

Let $\Gamma_{\alpha,1}$ and $\Gamma_{\beta,1}$ be independent Gamma distributed random variables. Then the Beta distribution is linked to the gamma distribution such that $\frac{\Gamma_{\alpha,1}}{\Gamma_{\alpha,1}+\Gamma_{\beta,1}}$ is from the Beta distribution $B_{\alpha,\beta}$ [7].
So one may simply generate a beta random number from three Gamma distributed random numbers. However, to increase efficiency, Beta distributed random numbers are generated using the rejection sampling scheme by Cheng [5].

References

[1] A. C. Atkinson. The computer generation of poisson random variables. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):pp. 29–35, 1979.

[2] G. E. P. Box and Mervin E. Muller. A note on the generation of random number deviates. *Annals of Mathematical Statistics*, 29(2):610–611, 1958.

[3] Richard P. Brent. Note on Marsaglia's Xorshift Random Number Generators. *Journal of Statistical Software*, 11(4):1–5, 8 2004.

[4] I. N. Bronstein, K. A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik*. Deutsch (Harri), 3. überarb. und erw. aufl. der neubearb. edition, 1997.

[5] R. C. H. Cheng. Generating beta variates with nonintegral shape parameters. *Commun. ACM*, 21(4):317–322, April 1978.

[6] Luc Devroye. Generating the maximum of independent identically distributed random variables. *Computers & Mathematics with Applications*, 6(3):305 – 315, 1980.

[7] Luc Devroye. *Non-uniform Random Variate Generation*. Springer-Verlag New York, Inc., 1986.

[8] E. J. Dudewicz and Mishra S. N. *Modern Mathematical Statistics*. Wiley series in probability and mathematical statistics. John Wiley & Sons, 1988.

[9] Bernard D. Flury. Acceptance–rejection sampling made easy. *SIAM Review*, 32(3): 474–476, 1990.

[10] Donald E. Knuth. *The Art of Computer Programming - Fundamental Algorithms*, volume 1. Addison-Wesley, 3rd edition, 1997.

[11] Donald E. Knuth. *Art of Computer Programming - Seminumerical Algorithms*, volume 2. Addison-Wesley, 3rd edition, 1997.

[12] Pierre L'Ecuyer and Richard Simard. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33, 2007.

[13] G. Marsaglia. The Marsaglia Random Number CDROM, the The Diehard Battery of Tests of Randomness. produced at Florida State University under a grant from The National Science Foundation. Access available at www.stats.fsu.edu/pub/diehard, and a revised version of the Diehard tests at www.csis.hku.hk/ diehard, 1995.

[14] George Marsaglia. The squeeze method for generating gamma variates. *Computers & Mathematics with Applications*, 3(4):321 – 325, 1977.

[15] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 7 2003.

[16] George Marsaglia and Wai Wan Tsang. The ziggurat method for generating random variables. *Journal of Statistical Software*, 5(8):1–7, 10 2000.

[17] George Marsaglia and Wai Wan Tsang. Some difficult-to-pass tests of randomness. *Journal of Statistical Software*, 7(3):1–9, 1 2002.

[18] W. Molenaar. Approximations to the poisson, binomial and hypergeometric distribution functions. Technical report, Mathematisch Centrum Amsterdam, 1970. Mathematical Centre Tracts 31.

[19] François Panneton and Pierre L'Ecuyer. On the xorshift random number generators. *ACM Trans. Model. Comput. Simul.*, 15:346–361, October 2005.

[20] W. H. Press, A. T. Saul, William T. V., and Brian P. F. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 2007. ISBN 9780521880688.

[21] Alan Stuart. Gamma-distributed products of independent random variables. *Biometrika*, 49(3-4):564–565, 1962.

[22] F. Yates. Contingency tables involving small numbers and the χ^2 test. *Supplement to the Journal of the Royal Statistical Society*, 1(2):pp. 217–235, 1934.