Ziggurat Revisited

Dirk Eddelbuettel[a]

[a] http://dirk.eddelbuettel.com

This version was compiled on September 27, 2017

Random numbers following a Standard Normal distribution are of great importance when using simulations as a means for investigation. The Ziggurat method (Marsaglia and Tsang, 2000; Leong *et al.*, 2005) is one of the fastest methods to generate normally distributed random numbers while also providing excellent statistical properties. This note provides an updated implementations of the Ziggurat generator suitable for 32- and 64-bit operating system. It compares the original implementations to several popular Open Source implementations. A new implementation embeds the generator into an appropriate C++ class structure. The performance of the different generator is investigated both via extended timing and through a series of statistical tests, including a suggested new test for testing Normal deviates directly. Integration into other systems such as R is discussed as well.

Introduction

Generating random number for use in simulation is a classic topic in scientific computing and about as old as the field itself. Most algorithms concentrate on the uniform distribution. Its values can be used to generate randomly distributed values from other distributions simply by using the relevant inverse function.

Regarding terminology, all computational algorithms for generation of *random* numbers are by definition deterministic. Here, we follow standard convention and refer to such numbers as *pseudo-random* as they can always be recreated given the seed value for a given sequence. We are not concerned in this note with *quasi-random* numbers (also called low-discrepnancy sequences). We consider this topic to be subset of pseudo-random numbers subject to distributional constraints. Without lack of generality, we will henceforth drop the qualifier *pseudo* when refering to random numbers.

Due to its importance for many modeling tasks, the Normal distribution has also been studied extensively in order to find suitable direct algorithms which generate stream of normally distributed (pseudo) random numbers. A well-known examples for such algorithms includes the method by Box and Muller (Box and Muller, 1958). A useful recent survey of the field is provided by Thomas *et al.* (2007).

In an important paper, Marsaglia and Tsang (2000) introduced the Ziggurat method. Since its initial publication, this algorithm has become reasonably popular[1] as it provides a very useful combination of both excellent statistical properties and execution speed. Thomas *et al.* (2007) conclude their survey by saying that

> the Ziggurat method, the second in speed, is about 33% slower than the [fastest] method but does not suffer from correlation problems. Thus, when maintaining extremely high statistical quality is the first priority, and subject to that constraint, speed is also desired, the Ziggurat method will often be the most appropriate choice.

[1] Usage of a code search engine such as code.ohloh.net or codesearch.debian.net provides a good approximation to the popularity of the Ziggurat algorithm as its name is also a reasonably unique search term within the field of computing.

This paper reexamines the Ziggurat method, provides a new and portable C++ implementation, and compares it to several other Open Source implementations of the underlying algorithm by applying three different statistical tests.

Ziggurat

This sections briefly discusses the key papers related to Ziggurat.

Marsaglia and Tsang. Marsaglia and Tsang (2000) introduced the Ziggurat method. It provides a fast algorithm for generating both normally and exponentially distributed random numbers. The original paper also contains a corresponding implementation in the C language.

The listing in Figure 1 shows this initial implementation. We have removed the code for generating exponentially distributed random numbers as well a comment header from the listing to keep the display more compact. The full listing is also included in the **RcppZiggurat** package for reference.

As can be seen from Figure 1, the Ziggurat algorithm is implemented in bare-bones C code using a number of macros, and two helper functions. The helper functions allow setting a seed, ane d deal with parameter updates needed in about 2.5% of cases. The actual core component—the function to draw a random number distributed according to thstandard normal distribution—is provided by the macro `RNOR`. Needless to say, using C macros is no longer cosidered *de rigeur*. Possible side effects include inadvertent changes in globally visible variables, as well as possible bugs from the macro evaluation.

A more important concern is that this implementation uses `unsigned long` types, and explicit bit mapping operations. The code was originally developed for 32-bit operating systems where `int` and `long` are typically four bytes (or 32 bits) wide. Hence the code does not produce correct results on a 64-bit operating system as (signed and unsigned) `long` types are typically eight bytes (or 64 bits) wide (whereas `int` is still 32 bits).

Our modified version introduced below overcomes both issues.

Leong, Zhang et al. Leong *et al.* (2005) show in a comment that the Ziggurat method as introduced by Marsaglia and Tsang (2000) suffers from another weakness due to the `SHR3` generator (by Marsaglia). The authors show via a χ^2-test that the generator has a short period of about $2^{32} - 1$, or the four byte limit. Replacing it with the `KISS` generator (also by Marsaglia) improves the performance beyond this limit.

```
#define MWC  ((znew<<16)+wmew )
#define SHR3 (jz=jsr, jsr^=(jsr<<13), \
        jsr^=(jsr>>17), jsr^=(jsr<<5),jz+jsr)
#define CONG (jcong=69069*jcong+1234567)
#define KISS ((MWC^CONG)+SHR3)

#define RNOR (hz=KISS, iz=hz&127, \
        (fabs(hz)<kn[iz]) ? hz*wn[iz] : nfix())
```

```
#include <math.h>
static unsigned long jz,jsr=123456789;

#define SHR3 (jz=jsr, jsr^=(jsr<<13), jsr^=(jsr>>17), jsr^=(jsr<<5),jz+jsr)
#define UNI (.5 + (signed) SHR3*.2328306e-9)
#define IUNI SHR3

static long hz;
static unsigned long iz, kn[128], ke[256];
static float wn[128],fn[128], we[256],fe[256];

#define RNOR (hz=SHR3, iz=hz&127, (fabs(hz)<kn[iz])? hz*wn[iz] : nfix())

/* nfix() generates variates from the residue when rejection in RNOR occurs. */
float nfix(void)
{
const float r = 3.442620f;     /* The start of the right tail */
static float x, y;
 for(;;)
  {  x=hz*wn[iz];      /* iz==0, handles the base strip */
     if(iz==0)
       { do{ x=-log(UNI)*0.2904764; y=-log(UNI);}   /* .2904764 is 1/r */
         while(y+y<x*x);
         return (hz>0)? r+x : -r-x;
       }
                        /* iz>0, handle the wedges of other strips */
      if( fn[iz]+UNI*(fn[iz-1]-fn[iz]) < exp(-.5*x*x) ) return x;

     /* initiate, try to exit for(;;) for loop*/
      hz=SHR3;
      iz=hz&127;
      if(fabs(hz)<kn[iz]) return (hz*wn[iz]);
  }
}

/*--------This procedure sets the seed and creates the tables------*/
void zigset(unsigned long jsrseed)
{  const double m1 = 2147483648.0, m2 = 4294967296.;
   double dn=3.442619855899,tn=dn,vn=9.91256303526217e-3, q;
   double de=7.697117470131487, te=de, ve=3.949659822581572e-3;
   int i;
   jsr^=jsrseed;

/* Set up tables for RNOR */
   q=vn/exp(-.5*dn*dn);
   kn[0]=(dn/q)*m1;
   kn[1]=0;

   wn[0]=q/m1;
   wn[127]=dn/m1;

   fn[0]=1.;
   fn[127]=exp(-.5*dn*dn);

    for(i=126;i>=1;i--)
    {dn=sqrt(-2.*log(vn/dn+exp(-.5*dn*dn)));
     kn[i+1]=(dn/tn)*m1;
     tn=dn;
     fn[i]=exp(-.5*dn*dn);
     wn[i]=dn/m1;
    }
}
```

Fig. 1. Ziggurat code by Marsaglia and Tsang (2000).

Following Leong *et al.* (2005), Ziggurat code should use an improved uniform generator. Choices are either KISS as suggested initially, or another trusted (and fast) uniform generator such as the Mersenne Twister (Matsumoto and Nishimura, 1998). Other Open Source implementations (such as the ones discussed below) frequently use the Mersenne Twister as the souce of uniformly distributed random numbers.

Voss's implementation in GNU GSL. Voss (2011) provided another Ziggurat implementation for use in the GNU Scientific Library or GSL (Galassi *et al.*, 2013). It uses the Mersenne Twister generator by Matsumoto and Nishimura (1998) which also avoids the issue identified by Leong *et al.* (2005) in which the originally-used uniform generator had too short a cycle.

Voss (2011) notes two differences between his implementation and the original work by Marsaglia and Tsang (2000). First, he uses only 128 instead of 256 steps which reduces the memory requirements and computational cost at a possible (though presumably minor) loss of precision. Second, he uses an exponential distribution with tail rejection for the base strip, which is motivated by a simpler implementations. Both of these aspects could have implications for the statistical properties of the generator. Voss also appears to be unaware of the work by Leong *et al.* (2005), yet sidesteps the issue raised by these author by relying on the Mersenne Twister generator.

Here, the implementation from the current GSL sources and file `randist/gausszig.c` is used, and adapted to the class strucuture detailed in section .

Gretl. The Gretl (Cottrell and Lucchetti, 2015) econometrics program contains another Open Source implementation of the Ziggurat algorithm. The code credits the implementation by Voss (2011) described above. Yalta and Schreiber (2012) review the Gretl implementation and performance of Ziggurat and find it to be satisfactory.

We use the implementation from the file `src/lib/random.c` from the current Gretl sources and adapt to the class strucuture detailed in section .

QuantLib. The QuantLib library for quantitative finance (Ametrano *et al.*, 2015) contains another open source implementation of Ziggurat. It is provided in the files `ql/experimental/math/zigguratrng.hpp` and `ql/experimental/math/zigguratrng.cpp`. As part of the experimental section, it is made available for further study and use, but not yet part of the default build. As before, we integrate this source into the class structure used here.

Speed

R Generators. Before comparing the speed of the different Ziggurat implementations, it is also illustrative to compare the different R generators. Figure 2 provides a comparison.

We see that the Box-Muller generator is the slowest by some margin. However, both Kinderman-Ramage and Ahrens-Dieter are faster than the Inversion method chosen as the default in R. So even before considering Ziggurat generators, R users could reap a speed benefit simply by calling `RNGkind(,"Ahrens-Dieter")` or `RNGkind(,"Kinderman-Ramage")`.

Ziggurat Generators. All Ziggurat generators are significantly faster the the default generator in R which uses a inversion method. Among the Ziggurat generators, we notice that approaches using an external uniform number generator (GNU GSL, GNU Gretl, QuantLib) are all slower than our compact and self-contained implementation which is seen as the fastest method.

Accuracy

Standard Test for Uniform RNG draws. Test for random number generators are often focussed on the case of uniform generators which are the most common type of generators. As detailed for example in Brown *et al.* (2013), a test proceeds as follow:

1. Take n draws from a $U(0,1)$ distribution (as any given $U(a,b)$ can always be scaled to $U(0,1)$), and then compute the sum of the n values.

2. Repeat this m times to create a set of m sums of uniform RNG draws.

3. With n large enough, the collection of m results will converge towards normally distributed random variable with a mean of $n/2$ and a standard deviation of $\sqrt{n/12}$ (which is the Irwin-Hall distribution of the sum of uniformly distributed values).

4. Given this asymptotic result, one can construct a probability value p_i for each of the m values using the inverse of the Normal distribution using the known mean and standard deviation from the Irwin-Hall distribution.

5. We now have m uniformly distributed values. A standard test such as Kolmogorov-Smirnow or Wilcoxon can be used to test against departures from the uniform distribution.

Here, we can apply this test for first converting the $N(0,1)$ distributed values produced by the given Ziggurat implementation to $U(0,1)$ distributed values by using the inverse of the normal distribution. We are then ready to simulate and test. Figure 4 below displays Q-Q plots for the empirical distribution against the uniform, and displays the p-values of a Kolmogorov-Smirnow as well as a Wilcoxon test.

We see that five of the six generators pass the test. In the case of the original Marsaglia and Tsang (2000) generator, we can see the departure from the expected diagonal clearly once we draw more than 4.2×10^9 numbers (which is the limit of the representation of an unsigned four-byte nunber). However, only the Kolmogorov-Smirnow test can formally reject; the Wilcoxon test appear to lack sufficient power in this setting. The QuantLib generator is seen as suspicious which p-value just below a conventional rejection level.

Normal Test for Normal RNG draws. We can propose a simpler variant of the test outlined in the previous section. As the random numbers we are drawing are following a $N(0,1)$ distribution, the sum of their values follows a $N(0, \sqrt{(N)}$ distribution. This allows us to skip one inversion step:

1. Take n draws from a $N(0,1)$ distribution and then compute the sum of the n values.

2. Repeats this m times to create a set of m sums of (standard) normals RNG draws.

3. The collection of the m sums of n normals converges towards a mean of 0 and a standard deviation of $\sqrt{n}$.

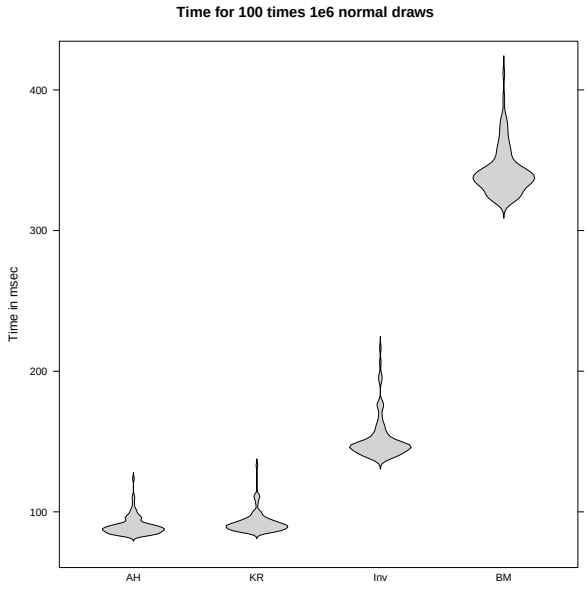

Fig. 2. R Normal RNG Generator Performance

Note: Figure shows timings from the **microbenchmark** package using 100 replications of 1,000,000 draws per generator. Code for the figure is included in the **RcppZiggurat** package, and the source code for this document.

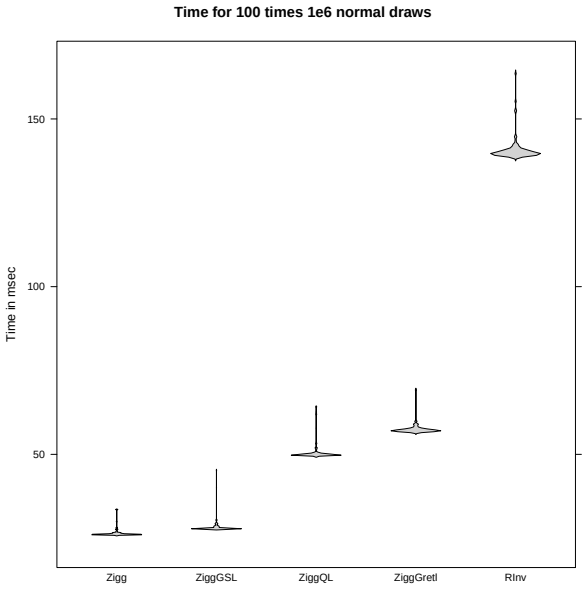

Fig. 3. Ziggurant and R Normal RNG Generator Performance

Note: Figure shows timings from the **microbenchmark** package using 100 replications of 1,000,000 draws per generator. Code for the figure is included in the **RcppZiggurat** package, and the source code for this document.

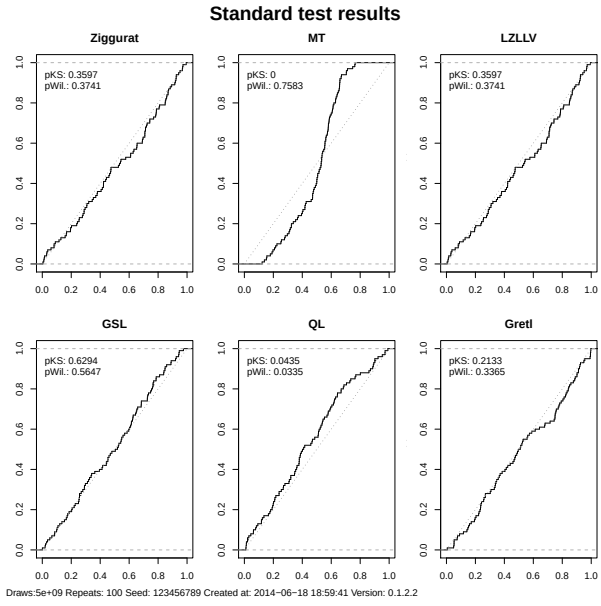

Fig. 4. Standard Test applied to Ziggurat generators

Note: Code for the figure is included in the **RcppZiggurat** package, and the source code for this document.

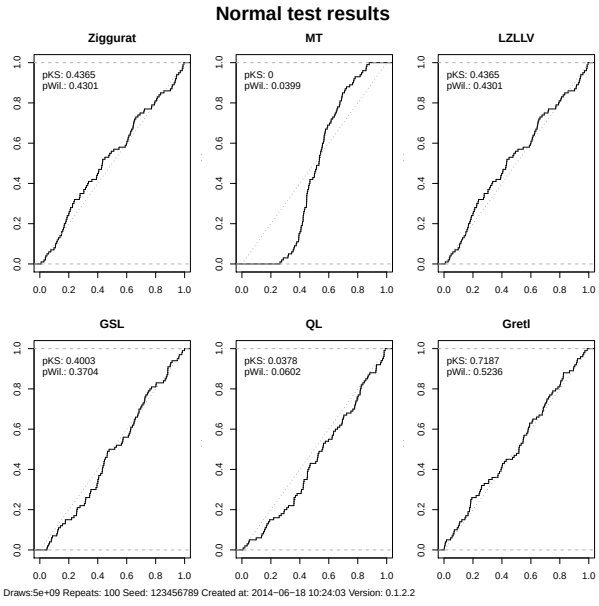

Fig. 5. Normal Test applied to Ziggurat generators

Note: Code for the figure is included in the **RcppZiggurat** package, and the source code for this document.

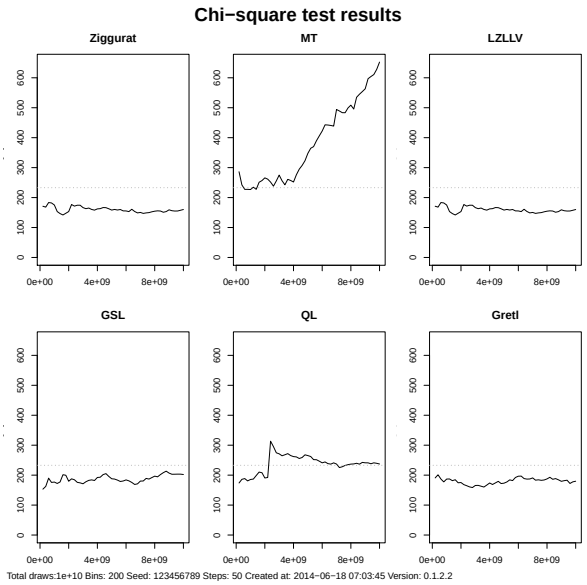

Fig. 6. χ^2 Test applied to Ziggurat generators

Note: Code for the figure is included in the **RcppZiggurat** package, and the source code for this document.

4. Given this known result, one can construct a probability value p_i for each of the m values using the inverse of the Normal distribution using the known mean and standard deviation.

5. We again have m uniformly distributed values. A standard test such as Kolmogorov-Smirnow or Wilcoxon can be used to test against departures from the uniform distribution.

Results, shown in Figure 5, are qualitatively similar to the result discussed above. Kolmogorov-Smirnow rejects for the Marsaglia and Tsang (2000) generator. However, we note that the Wilcoxon test now has a lower p-value—we would now reject at conventional test levels. The QuantLib implementation is now rejected by the Wilcoxon test but not the Kolmogorov-Smirnow.

χ^2 test. Another test variant is the χ^2 test which was also used by Leong *et al.* (2005). The basic idea is as follow:

1. The real line is divided into B bins, equally spaced between (symmetric) values distant enough from zero so that no $N(0,1)$ draw should exceed them.

2. Here, we follow Leong *et al.* (2005) and use a range from -7 to 7 with a total of 200 bins.

3. A large number of $N(0,1)$ random variates is drawn, and for each of these numbers a counter in the bin corresponding to the draw is increased.

4. After the N draws, the empirical distribution is compared to the theoretical (provided by the corresponding value of the Normal density function) using a standard chi^2 test.

As can be seen in Figure 6, the original proposal by Marsaglia and Tsang (2000) fails as was shown by Leong *et al.* (2005). All other tests pass again.

C++ Implementation. Preceding work by Marsaglia and Tsang (2000) and Leong *et al.* (2005) also contained implementations in the C language. These versions were implemented in just a few lines, and used idioms common to C programmers such as macros and global variables.

C++ programming style permits encapsulation in order to avoid possible collisions and side-effects. Moreover, by using a modest amount of object-oriented programming we can use a class structure with a common base class to express commonalities between the implementations. The following code segment shows the virtual base class used here.

```
#ifndef RcppZiggurat__Zigg_h
#define RcppZiggurat__Zigg_h

#include <cmath>
#include <stdint.h>         // or cstdint (C++11)

namespace Ziggurat {

   class Zigg {
   public:
      virtual ~Zigg() {};
      virtual void setSeed(const uint32_t s) = 0;
      // no getSeed() as GSL has none
      virtual double norm() = 0;
   };
}

#endif
```

As shown in the preceding code segment, we provide two user-accessible functions to obtain a normal random deviate, and to set the seed, respectively. The actual implementation uses portable types such as `uint32_t`, an unsigned 32-bit integer provided by the C header file `stdint.h`, which provides correct and identical results on both 32-bit and 64-bit operating systems.

Each actual implementation can then encapsulate its state variable as a private variable inaccessible to other functions. Such a small core to each class also makes it feasible to provide a Ziggurat generator in each thread in a parallel execution framework.

Our Ziggurat implementation has no external dependencies and can therefore be used in other projects. The testing framework used for this note has a single dependency on the GNU GSL as the generator by Voss (2011) is used via its GSL implementations. The generator and testing framework in the corresponding R package have a build-dependency on R, and are of course accessed by R. But the generator discussed here could equally well be used in standalone programs or with other scripting languages.

R Integration

In the **RcppZiggurat** package, the Rcpp Attributes (Allaire *et al.*, 2015) feature of the **Rcpp** C++/R integration package (Eddelbuettel and François, 2015; Eddelbuettel, 2013) are used to access instances of the corresponding generator class.

```
#include <Rcpp.h>

#include <Ziggurat.h>
```

```
static Ziggurat::Ziggurat::Ziggurat zigg;

// [[Rcpp::export]]
Rcpp::NumericVector zrnorm(int n) {
    Rcpp::NumericVector x(n);
    for (int i=0; i<n; i++) {
        x[i] = zigg.norm();
    }
    return x;
}

// [[Rcpp::export]]
void zsetseed(unsigned long int s) {
    zigg.setSeed(s);
    return;
}
```

In this particular reference implementation, we have chosen a namespace `Ziggurat` for the entire project. Within this namespace, we opted to provided an extra namespace layer for each generator as some of these generators still use global variables—which are therefore shielded in their own namespace. For example, for the Marsaglia and Tsang (2000) generator, we use `Ziggurat::ZigurratMT`. Next is the name of the actual class—which in the case of the reference implementation shown above is once again `Ziggurat` leading to the triple use of the term. Actual deployment of the Ziggurat generator (without comparison to other implementations and concerns about interaction between variables, particularly for the older implementations having global variables) can of course be used with a single namespace.

The remainder of code segment shows how two functions `znorm()` and `zsetseed()` are provided via the attribute `[[Rcpp::Export]]` as described in the Rcpp Attributes vignette (Allaire *et al.*, 2015) of the **Rcpp** package (Eddelbuettel and François, 2015; Eddelbuettel, 2013).

Conclusion

This note describes the **RcppZiggurat** package and its new implementation of the Ziggurat generator for normally distributed random numbers. The package is implemented in a way which is portable so that it can be used on 32-bit and 64-bit operating systems, filling a gap left by the original implementations (Marsaglia and Tsang, 2000; Leong *et al.*, 2005).

By embedding the code in a simple C++ class structure, we can ease testing and comparison of several variants of the algorithm. Our note reconfirmed the findings by Leong *et al.* (2005) of a short cycle due to to the use of an inferior uniform generator. Replacing the generator leads to better performance.

We suggest a new test for random number generators producing deviates distributed according to the standard normal distribution by adapting and simplifying an existing test framework for uniform deviates. Both tests confirm the (previously documented) failure of the original Ziggurat proposal but do not find a problem with any of the new implementations (apart from the still-experimental QuantLib generator).

A key motivation for this work has been a desire to improve the speed of creating standard-normally distributed random numbers in R. We find Ziggurat to be faster than the existing implementations, and hope that this generator will be of use to those generating large numbers of draws.

Acknowledgments. The initial implementations by John Burkardt were very helpful at the beginning of this project. Our suggestions of using portable types from the `stdint.h` header file for 32- and 64-bit use, as well as the need to reflect the insight of Leong *et al.* (2005) improved his version and our versions in a very fruitful email exchange which is gratefully acknowledged. Comments by Jochen Voss are also gratefully acknowledged.

References

Allaire JJ, Eddelbuettel D, François R (2015). *Rcpp Attributes*. Vignette included in R package Rcpp, URL http://CRAN.R-Project.org/package=Rcpp.

Ametrano F, Ballabio L, Bianchetti M, Césaré ND, Eddelbuettel D, Firth N, Jean N, Kenyon C, Lichters R, Marchioro M, Spanderen K, Wang J (2015). "QuantLib: A free/open-source library for quantitative finance." Version 1.6., URL http://quantlib.org/.

Box GEP, Muller ME (1958). "A Note on the Generation of Random Normal Deviates." *Annals of Mathematical Statistics*, **29**(2), 610–611.

Brown RG, Eddelbuettel D, Bauer D (2013). "Dieharder: A Random Number Test Suite." Open Source software library, under development, URL http://www.phy.duke.edu/~rgb/General/dieharder.php.

Cottrell A, Lucchetti R (2015). *Gretl User's Guide*. Version 1.10.1, URL http://gretl.sourceforge.net/.

Eddelbuettel D (2013). *Seamless R and C++ Integration with Rcpp*. Use R! Springer, New York. ISBN 978-1-4614-6867-7.

Eddelbuettel D, François R (2015). *Rcpp: Seamless R and C++ Integration*. R package version 0.12.0, URL http://CRAN.R-Project.org/package=Rcpp.

Galassi M, Davies J, Theiler J, Gough B, Jungman G, Alken P, Booth M, Rossi F (2013). *GNU Scientific Library Reference Manual*, 3rd edition. Version 1.16. ISBN 0954612078, URL http://www.gnu.org/software/gsl.

Leong PHW, Zhang G, Lee DU, Luk W, Villasenor J (2005). "A Comment on the Implementation of the Ziggurat Method." *Journal of Statistical Software*, **12**(7), 1–4. ISSN 1548-7660. URL http://www.jstatsoft.org/v12/i07.

Marsaglia G, Tsang WW (2000). "The Ziggurat Method for Generating Random Variables." *Journal of Statistical Software*, **5**(8), 1–7. ISSN 1548-7660. URL http://www.jstatsoft.org/v05/i08.

Matsumoto M, Nishimura T (1998). "Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator." *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, **8**(1), 3–30. ISSN 1049-3301. . URL http://doi.acm.org/10.1145/272991.272995.

Thomas DB, Luk W, Leong PHW, Villasenor JD (2007). "Gaussian random number generators." *ACM Computing Surveys*, **39**(4), 11:1–11:38.

Voss J (2011). "The Ziggurat Method for Generating Gaussian Random Numbers." Webpage at http://www.seehuhn.de/pages/ziggurat.

Yalta AT, Schreiber S (2012). "Random Number Generation in gretl." *Journal of Statistical Software, Code Snippets*, **50**(1), 1–13. ISSN 1548-7660. URL http://www.jstatsoft.org/v50/c01.