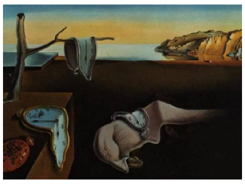

Journal of Statistical Software
August 2012, Volume 50, Code Snippet 1. *http://www.jstatsoft.org/*

Random Number Generation in gretl

A. Talha Yalta
TOBB University of Economics
and Technology

Sven Schreiber
Macroeconomic Policy Institute
(IMK)

Abstract

The increasing popularity and complexity of random number intensive methods such as simulation and bootstrapping in econometrics requires researchers to have a good grasp of random number generation in general, and the specific generators that they employ in particular. Here, we discuss the random number generation options, their specifications, and their implementations in gretl. We also assess the performance and the reliability of gretl in this department by conducting extensive empirical testing using the **TestU01** library. Our results show that the available alternatives are soundly implemented and should be sufficient for most econometric applications.

Keywords: gretl, **TestU01**, RNG, random numbers.

---

1. Introduction

Mathematicians have long known that random number generation is too important to be left to chance (Coveyou 1969). Many econometric discussions, however, just assume that one will always get reliable random numbers by using in one way or another the random number generator (RNG) found in all econometric software packages. This, of course, is far from the truth since the scientific literature has many examples of invalid simulation results caused by bad RNGs (Coddington 1994, 1996). Moreover, studies such as McCullough (1999a,b, 2008), Vinod (2000), McCullough and Wilson (1999, 2002, 2005), and Yalta (2007) have performed tests exposing problems in the RNGs of various popular programs offering statistical and econometric functionality. Furthermore, studies such as Tirler, Dalgaard, Hormann, and Leydold (2004), Doornik (2005), Panneton, L'Ecuyer, and Matsumoto (2006), and Saito and Matsumoto (2008) have offered corrections and improvements over well-established methods for generating random values. These efforts, in many cases, have resulted in the adoption of better algorithms or implementation improvements in the newer versions of various econometric and statistical programs.

Given the importance of the issue, researchers should always be able to verify that the RNG they use in a given simulation is of a high quality. This requires evaluation of the available RNG options, assessment of their limitations, and documentation of details regarding their delivery in a given software package. This information is crucial not only for the reliability of computational results, but also for their reproducibility, which is becoming an increasingly important concern in the field of economics.[1]

Our objective in this study is to provide a discussion and a detailed empirical testing of the RNGs in gretl (Cottrell and Lucchetti). Written "by econometricians for econometricians," gretl is the flagship free/libre and open source software (FLOSS) in the domain of econometrics. It provides a comprehensive set of tools accessible through an intuitive graphical user interface as well as an integrated scripting language. It also links to R, Octave, and Ox for further data analysis. Thanks to being FLOSS, the program is by definition more inspectable and therefore can be more trustworthy in comparison to proprietary black-box alternatives (Yalta 2010). Its accuracy has been tested, verified, and documented on various fronts and occasions by Baiocchi and Distaso (2003), Yalta and Jenal (2009) as well as Yalta and Yalta (2007, 2009). In addition, reviews by Mixon and Smith (2006), and Rosenblad (2008) discuss the merits of the program in doing research as well as teaching econometrics; while Adkins (2011) and Lucchetti (2011) respectively focus on performing simulations and state space estimations using gretl.

The rest of the paper is organized as follows. In the next section, we discuss from the perspective of applied researchers some of the key concepts in random number generation in econometric programs. In Section 3, we examine the RNGs and the other methods employed by gretl to produce random uniform and random normal values. This is followed by the presentation of detailed empirical results assessing the RNGs implemented in gretl. Section 5 concludes.

2. Random number generators

The fundamental fact that a researcher needs to know about computer generated random numbers is, of course, that they are not truly random.[2] For most research purposes, we rely on *pseudorandom* numbers generated deterministically by various mathematical algorithms.[3] The independently identically distributed $U(0,1)$ values that these algorithms are designed to produce is the source of virtually all random numbers used in increasingly large quantities for a variety of tasks such as simulation, bootstrapping, and Bayesian econometrics.

The first and the most basic RNG is probably the middle-square method described by von Neumann (1951). His approach allows creating a series of random numbers by first picking a 4 digit starting value, taking its square, and using the middle 4 digits of the result as the input for the next iteration. This technique has a number of desirable features expected from a computer based RNG such as being fast, being portable, and having a small memory

---

[1] Regarding the difficulty of reproducing results in economics and how to overcome this problem, see Anderson, Greene, McCullough, and Vinod (2008) and Koenker and Zeileis (2009).

[2] The need for true randomness is rather limited to specific tasks such as lotteries and cryptography, which first and foremost require unpredictability. Real random numbers required for such purposes are usually obtained based on physical randomness created by various specialized hardware devices.

[3] Because the scope of the paper is limited to pseudorandom numbers, here we will simply refer to them as random numbers.

footprint. Among its important limitations, however, are repeating short series as well as series declining to zero, which have later resulted in the development of more sophisticated RNGs thanks to increased computational capacity.

Ripley (1990) describes the four quality criteria for a good RNG as

1. repeatability based on a simply specified starting point (seed),
2. a very long period,
3. output approximates the uniform distribution very well,
4. output is independent in a moderate number of dimensions.

These characteristics are perhaps best discussed by using as an example a well-known class of simple RNGs known as linear congruential generators (LCGs)

$$x_{i+1} = (ax_i + b) \mod m, \tag{1}$$

which has the integer constants $0 < m$ as the "modulus", $0 < a < m$ as the "multiplier", and $0 < b < m$ as the "increment".

Obviously, an LCG satisfies the first requirement by allowing series that are easily reproduced based on x_0, the seed. For deterministic RNGs, this is the most vital feature required for the verification and repeatability of a given simulation experiment without having to store a huge set of data.

The period of an RNG, mentioned as a second important criterion, measures the maximum number of random variates that can be generated before the series starts repeating itself. Having a huge period length is crucial due to the ever increasing size and complexity of random number intensive applications. Depending on the choice of a and b, an LCG is known to have a maximum period of $m - 1$. For many modern applications, this will be too short even for the typical case where $m = 2^{31} - 1$, the largest signed integer representable in a 32-bit integer type.[4]

The third condition regarding whether a finite set of values produced by an RNG satisfies uniformity in $(0, 1)$ is, in general, assessed empirically by using various statistical tests for randomness. Because what represents a random series is a relative concept, a high entropy output, one that shows a significant level of disorder and chaos, is an important concern in these tests. The LCG class of generators also fail in this department because their successive output values can show strong correlation, especially when the choice of the parameters a, b, and m is poor (L'Ecuyer and Simard 2007).

The fourth element requires that, starting with a random seed s_0, the first n output values are uniformly distributed over the n-dimensional unit hypercube $[0, 1]^n$. In other words, drawing, say, three random numbers at a time should give individual sets that are independent. LCGs disappoint here as well since performing the aforementioned test with an LCG actually results in points lying orderly on a relatively small number of parallel planes inside the unit cube. This phenomenon is known as the Marsaglia (1968) effect.

---

[4] 2^{31} roughly equals 10^9, which is rather limited given that today simulations can require billions of random numbers. In addition, as the calls to a LCG approaches the period length, the generated series diverge from the uniform distribution, which in turn requires using only a fraction of the RNG's output (Knuth 1997).

Over the years, better understanding of the structural weaknesses of LCGs, along with the need for bigger and better random series, has resulted in an explosion of RNGs proposed by different authors. Today, we have classes of RNGs such as multiple recursive generators, lagged Fibonacci generators, subtract with borrow generators, feedback shift register generators, inversive generators, and so on. Within each group, there are also individual RNGs with different specifications and behavior based on the choice of parameters. As RNGs become more and more complex, their structures become increasingly difficult to analyze theoretically. This in turn requires that their performance and quality be appraised empirically.

One simple way to assess randomness is a chi-square test comparing the observed frequencies of a series of numbers to those expected from the $U(0, 1)$ distribution. Other tests, in addition to the various tests measuring global uniformity, include clustering tests, run and gap tests, serial tests, power divergence tests, linear complexity tests, and so on. A large number of tests are needed and available because there are many many ways in which a set of numbers may be related systematically, especially when these numbers are in fact produced systematically. Regarding this fact, von Neuman is famously quoted by Knuth (1997) “Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.” On the other hand, because such systematically generated numbers are important for practical purposes, Marsaglia, a prominent guru of random number generation, replies “Who among us has not sinned?”[5] Indeed, in many cases, all a researcher needs is a good RNG whose structure is so well hidden that the null hypothesis of orderly pattern is rejected by a sufficiently large set of statistical tests.

The first standard battery of randomness tests was proposed in the first edition of Knuth (1997) in 1969. This test suite included 11 relatively basic tests, some of which had been proposed much earlier during the days of using tables of random digits. Over time, these needed to be supplanted by 17 “stringent” tests compiled by Marsaglia (1996) in the test suite “**Diehard**: A Battery of Tests of Randomness.” The **Diehard** program, which is based on analyzing a user provided file containing by default 3 million random digits, has a few drawbacks particularly in the departments of extensibility, customizability, and documentation. Inevitably, with the need for increasingly bigger random series, **Diehard** became not so stringent, which resulted in a successor entitled **TestU01** proposed by L’Ecuyer and Simard (2007). Today, when it comes to testing of RNGs, **TestU01** can be said to be the method of choice, at least until the ever increasing scale of computations requires an even more stringent suite of tests.[6]

Without doubt, one can always design new tests and test suites that will reject randomness for any finite series considered random previously. This is due to the fact that suites such as **TestU01** are not really tests of “randomness” in the modern mathematical definition of Kolmogorov/Chaitin algorithmic information theory.[7] Rather, they are tests of equidistribution and independence in specific dimensions, which are open ended criteria. Consequently, for

---

[5] Quoted in Dembski (1991).

[6] There exists other software for testing RNGs such as the **Dieharder** program by Brown (2006) as well as the Scalable Parallel Pseudo Random Number Generators Library (**SPRNG**) by Mascagni and Srinivasan (2000), which includes a suite of the parallel versions of some of the classical tests. Also the National Institute of Standards and Technology (2010) offers the Statistical Test Suite (**STS**) comprised of 15 tests intended for testing of RNGs for use in cryptographic applications.

[7] The algorithmic information theory postulates that a sequence is more and more random as the shortest program that can produce it becomes longer and longer (Kolmogorov 1965). In these terms, software based RNGs are not random because they produce vast sequences from a small computer program.

assessing the quality of an RNG, the generally accepted approach is to have it undergo a large number of tests and see if it passes most of the complicated trials and does not fail any of the basic ones. Such attempts have revealed, and continue to reveal, serious deficiencies in random number generation in many widely-used programs. As L'Ecuyer (2010) notes

> The bad news is that a majority of the RNGs available in popular commercial software fail (randomness) tests unequivocally, with p values smaller than 10^{-15}. These generators should be discarded, unless we have very good reasons to believe that for our specific simulation models, the problems detected by these failed tests will not affect the results. The good news is that some freely-available high-quality generators pass all the tests in these batteries. Of course, passing all these tests is not a proof that the RNG is reliable for all possible simulations; but it certainly improves our confidence in the generator.

For a detailed account of RNGs and their testing, the reader is referred to L'Ecuyer (2006, 2010) and L'Ecuyer and Simard (2009).

3. Random number generation in gretl

Up until version 1.0.5, the uniform RNG in gretl was the relatively basic `rand()` function in the system C library. With version 1.0.6 (March 2003), the existing generator was replaced by the much superior Mersenne Twister (MT) algorithm by Matsumoto and Nishimura (1998). MT has a number of desirable properties including a super astronomical period of $2^{19937} - 1$ as well as a 623-dimensional equidistribution up to 32-bit accuracy. It is particularly well suited for Monte Carlo analysis thanks to being about as fast as a simple LCG, and also because the quality of its output can be studied in detail analytically. The version of the algorithm used by gretl is MT19937, with 32-bit word length, as implemented with some modifications in the seeding algorithm by the **GLib** software utility library.[8] The RNG can be scrutinized from the `glib/grand.c` file under the `glib/2.26.1` source directory at `http://ftp.gnome.org/pub/gnome/sources/glib/2.26/`.

Various custom implementations of the popular MT generator are used in many software packages including R, Maple, SPSS, EViews, and MATLAB among others. However, the MT is also known to have a subtle equidistribution problem. When the state space of the RNG contains decidedly more 0's than 1's, which can be caused by bad initialization, then it can have a negative bias for the generation of up to around 700000 random numbers (Panneton *et al.* 2006). Because it is possible that such "0-excess" states lead to wrong simulation results, various authors have recently proposed MT-based alternatives which are superior from a theoretical perspective.

In order to offer an improved alternative to MT19937, gretl 1.9.4 (February 2011) included a new RNG as the default method for generating random uniforms. The SIMD-oriented Fast Mersenne Twister (SFMT, Saito and Matsumoto 2008) is a new variant of MT based on using 128-bit single instruction multiple data (SIMD) operations, which represent an advanced technique for achieving data level parallel computing. In comparison to MT, SFMT is not

---

[8] gretl 1.0.6 also included the original MT code (`mt19937ar.c`) as an alternative to the **GLib** implementation because, previously, use of **GLib** was optional at compile time. This code was later removed in 2007, by which time **GLib** had become a required dependency for libgretl.

only significantly faster, but also has improved dimensions of equidistribution as well as better recovery from a 0-excess state space. The RNG supports various periods from $2^{607} - 1$ to $2^{216091} - 1$. gretl's SFMT19937 implementation, which has a period of $2^{19937} - 1$, is based on version 1.3.3 of the original C code by Saito and Matsumoto (2007). It can be examined using SourceForge `viewvc` interface located at `http://gretl.cvs.sourceforge.net/viewvc/gretl/gretl/rng/`.

In addition to having a good uniform RNG, econometric programs also need a reliable and efficient method for generating a random variable from a given decreasing density. In particular, the generation of random normals is of interest due to the crucial importance of the normal distribution in statistical applications. From the first introduction of a normal RNG until late 2009, the generation of random normals in gretl was based on the Box and Muller (1958) method. This approach, which uses two independent uniform $(0, 1)$ variates to produce a pair of standard normal variates, is commonly expressed in two forms namely the basic form and the polar form. For a single random value, gretl employs the polar form, however, the basic form is also used when filling an array.[9]

Although Box-Muller is faster than its predecessors (Golder and Settle 1976), it is also slow in comparison to some newer alternatives. In addition, the method is known to cause poor sampling distributions leading to wrong simulation results when it is used in conjunction with the multiplicative congruential class of RNGs (Neave 1973). Since version 1.8.7 (January 2010), gretl uses "ziggurat" as the default method for generating normal variates on the basis of uniform input.[10] Introduced by Marsaglia and Tsang (1984) and refined by Marsaglia and Tsang (2000), the ziggurat approach is based on partitioning a unimodal density into horizontal blocks of equal area and then using a single uniform random number to choose a point by the method of rejection. It is an attractive choice thanks to its speed and efficiency. However, as noted by Doornik (2005), ziggurat also has a potential dependency problem, which was recognized during its adoption by gretl. The innovation in the gretl RNG code was to solve the problem by sacrificing speed to the least possible extent. The programming code for ziggurat as well as Box-Muller, which is still available as an option, is accessible from `http://gretl.cvs.sourceforge.net/viewvc/gretl/gretl/lib/src/random.c`.

Although designed to be extensible in terms of RNGs, currently, gretl only offers the SFMT and MT RNGs as well as the ziggurat and the Box-Muller methods discussed above. By comparison, R provides by default 6 uniform RNGs and 4 different methods for the generation of decreasing densities from uniform random values.[11] Consequently, in order to provide better support for replicating simulation results, it can be useful to include more RNG options, especially those relatively more popular ones such as the lagged-Fibonacci generator proposed in the 2002 reprint of Knuth (1997). Two other worthwhile additions would be the well-equidistributed long-period linear (WELL) proposed by Panneton *et al.* (2006) as well as George Marsaglia's SUPER KISS generator discussed by Jones (2010). Despite being noticeably slower in comparison to SFMT, the WELL currently has the highest quality output theoretically (Saito and Matsumoto 2008). SUPER KISS, on the other hand, provides a

---

[9] The respective functions are `ran_normal_box_muller()` and `gretl_two_snormals()`, which depend on gretl's custom implementation of the method.

[10] The implementation is based on the `gauss.c` code by Voss (2005), which was written for use with the GNU Scientific Library.

[11] R also has a number of contributed packages which provide further RNG options. For more information, the reader is referred to the Comprehensive R Archive Network task view on "Probability Distributions" (Dutang 2012).

	SmallCrush		Crush		BigCrush	
RNG	32-bit	64-bit	32-bit	64-bit	32-bit	64-bit
SFMT19937	0	0	0	0	2	2
MT19937	0	0	2	2 (2)	2	2

Table 1: **TestU01** failures for random uniform generation in gretl.

period of 54767×2^{1337279}, which is over 10^{396564} times as long as Mersenne Twister.

A second feature currently lacking in gretl in the department of random number generation is the support for RNG "streams". This important functionality enables defining and saving the state of user-named generators, which are useful for complicated simulation experiments. At present, this can only be achieved by using in conjunction with **libgretl** the **GLib** software library which offers named RNG streams. Doing this, however, requires programming in C.

4. Randomness tests results

Written as a C library, **TestU01** by L'Ecuyer and Simard (2009) is widely used for empirical testing of RNGs (see McCullough 2006, for a review). It constitutes four modules containing respectively a large selection of RNG algorithms, dozens of randomness tests, a number of predefined batteries of tests, and various tools for testing entire families of generators. The program allows assessing variants as well as combinations of different RNGs without requiring a limited file based input. Among the included test suites are SmallCrush, Crush, and BigCrush in order of increasing difficulty. SmallCrush and Crush are less time consuming and are useful for detecting gross defects or wrong implementations. Very few RNGs can pass all of the 106 very stringent tests included in BigCrush, which takes hours on commodity hardware and uses about 2^{38} random numbers.

For our empirical testing of random number generation in gretl 1.9.3cvs, we employed the three Crush suites included in **TestU01** version 1.2.3. The online supplements accompanying the paper include the computational details and the **TestU01** output for all tests as separate text files as well as the C programs necessary to run these tests.

Table 1 presents the test results for uniform random number generation in gretl. Both RNGs pass the SmallCrush tests without failures. MT19937 fails the two *linear complexity* tests included in Crush and BigCrush with p values larger than $1 - 10^{-10}$. This is expected since all linear feedback shift register (LFSR) class generators, including MT and SFMT, fail these tests (L'Ecuyer and Simard 2007), which are designed to measure the linear complexity of binary sequences. The SFMT19937 performs significantly better by failing only at the BigCrush level. Also, as shown in parentheses, MT19937 has 2 additional "suspect" failures in the 64-bit Crush tests, with p values inside the interval $[10^{-10}, 1 - 10^{-10}]$ but outside of $[10^{-3}, 1 - 10^{-3}]$.[12] The failed tests are *collision over* and *weightdistrib*. The former is an overlapping pairs sparse occupancy (OPSO) test discussed by Marsaglia and Zaman (1993). The latter is one of the 8 uniformity tests included in the **TestU01** `svaria` module.

---

[12] When RNGs fail randomness tests, they usually do so with an extreme p value. L'Ecuyer and Simard (2007) consider test results with p values outside the interval $[10^{-10}, 1 - 10^{-10}]$ as "clear" failures, while they also discuss "suspect" failures where the p values are outside a smaller interval such as $[10^{-3}, 1 - 10^{-3}]$.

	SmallCrush		Crush		BigCrush	
RNG	32-bit	64-bit	32-bit	64-bit	32-bit	64-bit
Ziggurat (SFMT)	0	0	0	0	0	0
Ziggurat (MT)	0	0	0	0	(1)	0
Box-Muller (SFMT)	0	0	1	1	1	1 (1)
Box-Muller (MT)	0	0	1 (1)	1 (1)	1	1 (1)

Table 2: **TestU01** failures for random normal generation in gretl.

Because random normals are essential for most statistical simulations, testing the generation of normal variates is at least as important as testing the underlying uniform RNG. For this purpose, Doornik (2005) suggests transforming the normal random numbers back to uniformity using the normal cdf and then testing the resulting values using **TestU01**. Consequently, in order to test gretl's random normal implementation, we reconverted the normal output to $U(0,1)$ via the **libgretl** function `normal_cdf()`, which is accessible at `http://gretl.cvs.sourceforge.net/viewvc/gretl/gretl/lib/src/pvalues.c`.

As can be seen in Table 2, the ziggurat method passes all of the three crush tests in both 32-bit and 64-bit computations using either SFMT19937 or MT19937 as the underlying uniform generator, except only one suspect failure in a *matrix rank* test. Box-Muller, however, has one clear failure in the Crush and BigCrush tests, which is sometimes accompanied by a suspect failure depending on the test suite or the uniform source. The reason that the F_2-linear RNGs SFMT19937 and MT19937 pass the linear complexity tests after a transformation via either Box-Muller or ziggurat is straightforward: Both methods are nonlinear transformations which destroy the linearity, so the **TestU01** Crush tests no longer detect linearity. Also, the Box-Muller method transforms lines into spirals while the ziggurat performs a more complicated nonlinear transformation. This could be the explanation for the additional failures when Box-Muller is used, however, it should not be taken to mean that ziggurat is a better method. In all of the cases, the clear failures are once again in the *linear complexity* tests. The suspect failures, on the other hand, are seen in a *birthday spacings* test and a *maximum-of-t* test for the 32-bit and 64-bit Crush suite, and a *maximum-of-t* test and a *collision over* test for the 64-bit BigCrush suite with uniforms from SFMT and MT respectively.[13]

Table 3 shows our speed comparisons for the generation of random uniform and random normal values in gretl. For each RNG option, we generated 10^8 random 32-bit integers using two different methods, namely block generation and sequential generation. The first method is based on running 10^3 iterations of 10^5 variates in a single call. In the second method, the 10^8 random values are generated one per a call. The tests were performed using a 2.1 GHz Core 2 Duo computer set up for dual booting with the Windows Vista and Ubuntu Linux 10.10 operating systems. The results, which are based on reporting the best performance out of three separete runs, show that SFMT19937 is indeed significantly faster in comparison to MT19937. Also, Windows performance seems to be slightly better except for block generation using the Box-Muller method. It is interesting to see that, when the Ziggurat method is used, uniform generation can take longer than the normal RNG in some cases. The explanation is that, Ziggurat may sometimes require less calculation than for a random uniform when

---

[13] For detailed information regarding these tests, see L'Ecuyer and Simard (2009).

	Windows		Linux	
RNG	Block	Sequential	Block	Sequantial
SFMT	2.152	234.875	2.410	308.270
MT	6.489	241.841	7.120	312.710
Ziggurat (SFMT)	5.023	238.572	5.430	305.070
Ziggurat (MT)	5.881	242.448	6.410	309.410
Box-Muller (SFMT)	12.465	264.303	10.610	329.540
Box-Muller (MT)	18.813	276.054	16.350	344.200

Table 3: CPU times for generating 10^8 random numbers in gretl.

some of the work is done by simple look-up driven by a random integer instead of converting the integer to a double in $(0, 1)$. This can be verified from the programming code for gretl's `ran_normal_ziggurat()` command.

5. Conclusion

The ever increasing computational capacity, along with the introduction of newer and more advanced simulation techniques in econometrics, requires the use of RNGs with excellent statistical properties. There are many good and bad alternatives to choose from and the output of an RNG can vary depending on its implementation in a given program as well as the set of parameters chosen. Consequently, it is important that researchers have a level of proficiency in computer based RNGs, the specific RNG algorithm that they use, and its delivery in a given software package.

We provided a discussion of random number generation from the perspective of applied researchers. In addition, we offered detailed information regarding the RNG options in gretl, and also performed comprehensive empirical testing of the uniform generators as well as the methods used to obtain random normals. Our results show that the RNG related procedures in gretl are implemented soundly and perform well in the three crush test suites of the **TestU01** library. In particular, normal random values generated by the default ziggurat method pass all crush tests even though the underlying uniform values generated by the SFMT and MT RNGs fail only 2 tests in the BigCrush suite. For the reliability and replicability of simulation based research in economics and related fields, it is important that similar studies be performed and the results are published for other widely used programs.

Acknowledgments

We are grateful to Pierre L'Ecuyer and the three anonymous reviewers for extremely useful comments. We also wish to thank Allin Cottrell for his comments, and for providing prompt and useful answers to our reports and inquiries regarding gretl's inner workings.

References

Adkins LC (2011). "Using gretl for Monte Carlo Experiments." *Journal of Applied Econometrics*, **26**, 880–885.

Anderson RG, Greene WH, McCullough BD, Vinod HD (2008). "The Role of Data-Code Archives in the Future of Economic Research." *Journal of Economic Methodology*, **15**, 99–119.

Baiocchi G, Distaso W (2003). "gretl: Econometric Software for the GNU Generation." *Journal of Applied Econometrics*, **18**, 105–110.

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

Brown RG (2006). ***Dieharder**: A GNU Public Licensed Random Number Tester.* Included as file `manual/dieharder.tex` in the **Dieharder** sources, URL `http://www.phy.duke.edu/~rgb/General/dieharder.php`.

Coddington PD (1994). "Analysis of Random Number Generators Using Monte Carlo Simulation." *International Journal of Modern Physics C*, **3**, 547–560.

Coddington PD (1996). "Tests of Random Number Generators Using Ising Model Simulations." *International Journal of Modern Physics*, **7**, 295–303.

Cottrell A, Lucchetti R (2010). *gretl User's Guide – GNU Regression, Econometrics and Time-Series Library.* Version 1.9.3, URL `http://gretl.sourceforge.net/`.

Coveyou RR (1969). "Random Number Generation Is too Important to Be Left to Chance." *Studies in Applied Mathematics*, **3**, 70–111.

Dembski WA (1991). "Randomness by Design." *Noûs*, **25**, 75–106.

Doornik JA (2005). "An Improved Ziggurat Method to Generate Normal Random Samples." URL `http://www.doornik.com/research/ziggurat.pdf`.

Dutang C (2012). "CRAN Task View: Probability Distributions." Version 2012-07-29, URL `http://CRAN.R-project.org/view=Distributions`.

Golder ER, Settle JG (1976). "The Box-Müller Method for Generating Pseudo-Random Normal Deviates." *Journal of the Royal Statistical Society C*, **25**, 12–20.

Jones D (2010). "Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications." URL `http://www.cs.ucl.ac.uk/staff/d.jones/GoodPracticeRNG.pdf`.

Knuth DE (1997). *Art of Computer Programming, Volume 2: Seminumerical Algorithms.* 3rd edition. Addison-Wesley, Reading, MA.

Koenker R, Zeileis A (2009). "On Reproducible Econometric Research." *Journal of Applied Econometrics*, **24**, 833–847.

Kolmogorov AN (1965). "Three Approaches to the Quantitative Definition of Information." *Problems of Information Transmission*, **1**, 1–7.

L'Ecuyer P (2006). "Uniform Random Number Generation." In SG Henderson, BL Nelson (eds.), *Handbooks in Operations Research and Management Science: Simulations*, pp. 55–81. Elsevier, Amsterdam.

L'Ecuyer P (2010). "Pseudorandom Number Generators." In E Platen, P Jaeckel (eds.), *Simulation Methods in Financial Engineering*, pp. 1431–1437. John Wiley & Sons, Chichester, UK.

L'Ecuyer P, Simard R (2007). "**TestU01**: A C Library for Empirical Testing of Random Number Generators." *ACM Transactions on Mathematical Software*, **33**(22).

L'Ecuyer P, Simard R (2009). ***TestU01** A Software Library in ANSI C for Empirical Testing of Random Number Generators User's Guide, Compact Version.* URL `http://www.iro.umontreal.ca/~simardr/testu01/guideshorttestu01.pdf`.

Lucchetti R (2011). "State Space Methods in gretl." *Journal of Statistical Software*, **41**(11), 1–22.

Marsaglia G (1968). "Random Numbers Fall Mainly in the Planes." *Proceedings of the National Academy of Sciences of the United States of America*, **61**(1), 25–28.

Marsaglia G (1996). "**Diehard**: A Battery Test of Randomness." URL `http://stat.fsu.edu/pub/diehard`.

Marsaglia G, Tsang WW (1984). "A Fast, Easily Implemented Method for Sampling from Decreasing or Symmetric Unimodal Density Functions." *SIAM Journal on Scientific and Statistical Computing*, **5**, 349–359.

Marsaglia G, Tsang WW (2000). "The Ziggurat Method for Generating Random Variables." *Journal of Statistical Software*, **5**(8), 1–7.

Marsaglia G, Zaman A (1993). "Monkey Tests for Random Number Generators." *Computers and Mathematics with Applications*, **26**, 1–10.

Mascagni M, Srinivasan A (2000). "Algorithm 806: **SPRNG**: A Scalable Library for Pseudorandom Number Generation." *ACM Transactions on Mathematical Software*, **26**, 436–461.

Matsumoto M, Nishimura T (1998). "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator." *ACM Transactions on Modeling and Computer Simulation*, **8**, 3–30.

McCullough BD (1999a). "Assessing the Reliability of Statistical Software: Part II." *The American Statistician*, **53**, 149–159.

McCullough BD (1999b). "Econometric Software Reliability: **EViews**, **LIMDEP**, **SHAZAM** and **TSP**." *Journal of Applied Econometrics*, **14**, 191–202.

McCullough BD (2006). "A Review of **TESTU01**." *Journal of Applied Econometrics*, **21**, 677–682.

McCullough BD (2008). “Microsoft **Excel**’s ‘Not the Wichmann-Hill’ Random Number Generators.” *Computational Statistics & Data Analysis*, **52**, 4587–4593.

McCullough BD, Wilson B (1999). “On the Accuracy of Statistical Procedures in Microsoft **Excel** 97.” *Computational Statistics & Data Analysis*, **31**, 27–37.

McCullough BD, Wilson B (2002). “On the Accuracy of Statistical Procedures in Microsoft **Excel** 2000 and **Excel** XP.” *Computational Statistics & Data Analysis*, **40**, 713–721.

McCullough BD, Wilson B (2005). “On the Accuracy of Statistical Procedures in Microsoft **Excel** 2003.” *Computational Statistics & Data Analysis*, **49**, 1244–1252.

Mixon JW, Smith RJ (2006). “Teaching Undergraduate Econometrics with gretl.” *Journal of Applied Econometrics*, **21**, 1103–1107.

National Institute of Standards and Technology (2010). “A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.” NIST Special Publication 800-22 Revision 1a, URL `http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf`.

Neave HR (1973). “On Using the Box-Müller Transformation with Multiplicative Congruential Pseudo-Random Number Generators.” *Journal of the Royal Statistical Society C*, **22**, 92–97.

Panneton F, L’Ecuyer P, Matsumoto M (2006). “Improved Long-Period Generators Based on Linear Recurrences Modulo 2.” *ACM Transactions on Mathematical Software*, **32**, 1–16.

Ripley BD (1990). “Thoughts on Pseudorandom Number Generators.” *Journal of Computational and Applied Mathematics*, **31**, 153–163.

Rosenblad A (2008). “gretl 1.7.3.” *Journal of Statistical Software, Software Reviews*, **25**(1), 1–14. URL `http://www.jstatsoft.org/v25/s01`.

Saito M, Matsumoto M (2007). “SIMD-Oriented Fast Mersenne Twister (SFMT): Twice Faster than Mersenne Twister.” URL `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/`.

Saito M, Matsumoto M (2008). “SIMD-Oriented Fast Mersenne Twister: A 128-bit Pseudorandom Number Generator.” In A Keller, S Heinrich, H Niederreiter (eds.), *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pp. 607–622. Springer-Verlag, Berlin.

Tirler G, Dalgaard P, Hormann W, Leydold J (2004). “An Error in the Kinderman-Ramage Method and How to Fix It.” *Computational Statistics & Data Analysis*, **47**, 433–440.

Vinod HD (2000). “Review of GAUSS for Windows, Including Its Numerical Accuracy.” *Journal of Applied Econometrics*, **15**, 211–220.

von Neumann J (1951). “Various Techniques Used in Connection with Random Digits.” In AS Householder, GE Forsythe, HH Germond (eds.), *Monte Carlo Method*, pp. 36–38. US Government Printing Office, Washington DC.

Voss J (2005). “The Ziggurat Method for Generating Gaussian Random Numbers.” URL `http://seehuhn.de/pages/ziggurat`.

Yalta AT (2007). "The Numerical Reliability of GAUSS 8.0." *The American Statistician*, **61**, 262–268.

Yalta AT (2010). "Should Economists Use Open Source Software for Doing Research?" *Computational Econometrics*, **35**, 371–394.

Yalta AT, Jenal O (2009). "On the Importance of Verifying Forecasting Results." *International Journal of Forecasting*, **25**, 62–73.

Yalta AT, Yalta AY (2007). "gretl 1.6.0 and Its Numerical Accuracy." *Journal of Applied Econometrics*, **22**, 849–854.

Yalta AT, Yalta AY (2009). "Wilkinson Tests and gretl." In I Díaz-Emparanza, P Mariel, MV Esteban (eds.), *Econometrics with gretl – Proceedings of the gretl Conference 2009*, EHUCHAPS, chapter 16, pp. 243–251. Universidad del Pais Vasco – Departamento de Economia Aplicada III. URL http://www.gretlconference.org/proceedings/book2009-16.pdf.

Affiliation:

A. Talha Yalta
TOBB University of Economics and Technology
Department of Economics
Sogutozu Caddesi No:43
06560, Ankara, Turkey
E-mail: yalta@etu.edu.tr
URL: http://yalta.etu.edu.tr/index-en.html

Sven Schreiber
Macroeconomic Policy Institute (IMK)
Hans Böckler Foundation
40476 Düsseldorf, Germany
and
University of Hamburg
Institute for Growth and Business Cycle Analysis
Von-Melle-Park 5
20146 Hamburg, Germany
E-mail: svetosch@gmx.net
URL: http://sven.schreiber.name/, http://www.imk-boeckler.de/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/

Volume 50, Code Snippet 1 *Submitted:* 2011-01-31
August 2012 *Accepted:* 2012-08-06