Proceedings of the 2019 Winter Simulation Conference
N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, eds.

A MULTI-DIMENSIONAL, COUNTER-BASED PSEUDO RANDOM NUMBER GENERATOR AS A STANDARD FOR MONTE CARLO SIMULATIONS

Douglas W. Hubbard

Hubbard Decision Research
2S410 Canterbury Ct.
Glen Ellyn, IL 60137, USA

ABSTRACT

As decisions models involving Monte Carlo simulations become more widely used and more complex, the need to organize and share components of models increases. Standards have already been proposed which would facilitate the adoption and quality control of simulations. I propose a new pseudo-random number generator (PRNG) as part of those standards. This PRNG is a simple, multi-dimensional, counter-based equation which compares favorably to other widely used PRNGs in statistical tests of randomness. Wide adoption of this standard PRNG may be helped by the fact that the entire algorithm fits in a single cell in an Excel spreadsheet. Also, quality control and auditability will be helped because it will produce the same results in any common programming language regardless of differences in precision.

1 INTRODUCTION

Monte Carlo simulations are a powerful way to model uncertainty in risky and complex decisions. Even relatively simple cost-benefit analysis problems with a few uncertain variables will benefit from the use of a simulation to determine the probability distribution of net benefits given the probability distributions of project cost and durations, product demand, interest rates and growth rates (Hubbard 2009; Hubbard 2014; Savage 2012; Savage 2016).

These require the generation of samples of values which are distributed in a way which appear to be consistent with random values. The popular Microsoft Excel spreadsheet, for example, uses a "rand()" function to generate such values. Yet, like all programing languages, the generated values are not truly random, only *pseudo-random* (Knuth 1997; L'Ecuyer 2017). With a good pseudo-random number generator (PRNG), only the most advanced statistical randomness tests can detect any non-random pattern. A widely used PRNG, which is considered to be one of the better methods, is the Mersenne Twister (MT), a relatively complicated method based on large matrix operations of bits (Matsumoto and Nishimura 1998).

The use of Monte Carlo simulations do show benefits in making many decisions under uncertainty (Hubbard 2014). In fact, in many cases reliance on intuition or deterministic methods can lead to significantly suboptimal outcomes. Still, Monte Carlo simulations are not adopted nearly as widely as they should be. Adoption could be greatly facilitated with the development and promotion of certain standards. The establishment of standards in other areas such as units of measure, accounting methods, manufacturing, communication protocols and more, have resulted in revolutions of efficiency and quality. Likewise, the standardization of components of simulations can be the basis of creating sharable modules which can be exactly reproducible and can improve quality control and adoption of methods.

Standards are being proposed to this end. One such proposed standard is the Stochastic Information Packet (SIP) (Savage 2012; Kirmse and Savage 2014). A SIP is a vector of stored random values for a particular variable. Individual trials in a simulation call indexed values from this vector. A potential addition to these standards would be a PRNG simple enough to write in a single cell in a spreadsheet which could uniquely identify results for a given trial for uniquely identified variables. This would allow for the generation of a SIP without actually storing the values. These values could further be uniquely identified for time increments and other dimensions relevant to a model.

Using this "multi-dimensional" approach to construct a PRNG, parts of an organization making models supporting various decisions should be consistent in the use of distributions for the same variables – such as forecasts of demand for its products or the costs of raw materials. Government agencies may want to share SIPs representing distributions for weather-related disasters, epidemics, and so on.

This can be accomplished with a variation on a "counter-based" PRNG (Salmon et al. 2011). Most PRNGs, like MT, are "serial," meaning that the output of one iteration becomes the seed for the next. If I wanted to go to the millionth scenario, I would need to first compute the previous 999,999 or store a series of "milestones" to skip ahead. A counter-based PRNG only uses the number of the scenario as input. A multi-dimensional, counter-based PRNG would use separate counters for trial ID, variable ID, and other dimensions.

2 PRNG REQUIREMENTS

My objective was to make a simple and widely-usable pseudo-random number generator (PRNG) which compares well to other common PRNG methods on statistical tests of randomness. Any such attempt to make a widely used standard requires that I consider how to make it usable on a platform like Microsoft Excel, which has an estimated number of users of about 1 billion, worldwide. To meet this objective, I propose developing a method with the following guiding principles.

- It has to produce the same output regardless of the platform. Differences in precision and limits to various operations could lead to rounding errors and other problems that would mean that the series of values would be partly dependent on whether it ran on, for example, Python or Excel. Excel is a more limited environment than popular programming languages but as long as allowances are made for the following three considerations, it will produce the same answers in Excel as R, Python, C and other languages:
    - *Precision:* Excel has a precision limit of 10^15. Any operation that produces a value larger than that may introduce a rounding error at that step which can produce a significantly different final result.
    - *Modulus function:* A common and useful operation in a PRNG is the modulus function, X mod Y. It produces the remainder when dividing X by Y. In Excel, where this function is written MOD(X,Y), the "dividend" X cannot be greater than 2^27 (134,217,728) times the divisor Y. There is an obvious work around for this but it would also increase the length of the function especially if there are several modulus operations.
    - *Integer-based:* Integers should be both the input and the output of each operation in the algorithm. This precludes the use of operations like SIN() or EXP() or even just division other than a modulus function.

- It should fit in one cell in Excel. This would make it difficult to efficiently use loops and recursive functions that would be easy to implement in other languages. The MT would not be practical under this constraint. The upper limit for the size of a function in an Excel cell would still allow for a very large expression but, for the purpose of speed, the number of operations should be as small as possible while still meeting the other stated objectives in this list.

- It has to have multiple dimensions. I want to have a convenient way to identify a specific random number not only for a scenario, but also for each variable. An organization could have its own identifier and it would assign unique IDs to each variable and each of those organization and variable combinations would have its own sequence of numbers. Furthermore, each of those scenarios could itself have an entire time series.

- It should be a counter-based. As explained in the introduction, this provides a way to implement multiple dimensions. This could also facilitate massive parallel computing in simulations by simply assigning different processors different ranges of scenarios (one process runs the ten thousand, another the second, and so on). One could call this a "random access random number generator."

- It should be able to generate large numbers of scenarios for many combinations of variables, entities and other dimensions. At a minimum, the PRNG must be able to generate millions or tens of millions of scenarios each for combinations of millions of variables and millions of entities. This also implies limits on the length and complexity of operations in the PRNG.

- It should be fast enough to be practical as an interactive modeling tool. This means that running, say, 10,000 scenarios on 100 variables should take time on the order of only a few seconds in Excel given the best current PC processors. While Excel would allow for a very large expression in a single cell, a large number of operations would slow down modeling. Speed considerations mean that certain operations should be avoided. In Excel, converting decimal to text, binary and hexadecimal and back would slow down the algorithm. Additionally, Excel severely limits the size of numbers converted to binary or hexadecimal. Also, in PC-based Excel, operations like squaring could slow down the performance since this any operation involving an exponent moves to a specialized area of the processor.

- It must perform well on randomness tests compared to other methods. Even with all of these constraints, it must perform about as well or better on tests of statistical randomness than other widely used PRNG methods.

3 THE TESTS FOR RANDOMNESS

Random number tests work by looking for distributions one would expect to see for truly random numbers in a variety of situations. There are several suites of such tests, one of which is the "Dieharder" tests. These were developed as an evolutionary improvement on the earlier named "Diehard" tests developed by George Marsaglia. The Dieharder tests are a set of 114 statistical tests for Pseudo-Random Number Generators. Multiple online sources list the tests; one such source is Ubuntu (2019).

Each test uses the generated values to produce various distributions which are then compared to what an ideal random distribution should produce. For example, one test in Dieharder uses the PRNG to simulate the dice game of craps over and over and comparing the distribution of outcomes to what I would expect if I were playing with good random dice. Another example is the "parking lot" test, where the PRNG is used to plot circles of a certain size are placed on a 2-dimensional "parking lot." Parking is considered successful if the circle does not crash (overlap) with any other circle. For a random number source, I would expect that the number of successes also follows a defined distribution. Many of the tests require millions of values and one test (dab_monobit2) requires 65 million.

Each test produces a p-value, representing the probability that a truly random set of values would have produced those results or something more extreme. The p-value is used to determine whether the PRNG passes a given test, fails the test, or gives it a "weak" pass. Two of the tests were particularly difficult to pass and were good predictors of whether other tests would be passed. The "Marsaglia-Tsang" test requires

10 million generated values and the tests are based on finding the greatest common divisor in pairs of these generated numbers (Marsaglia and Tsang 2003). The greatest common divisors should follow a known distribution and the number of steps involved to find the greatest common divisor should also follow a known distribution. The Dieharder tests contain two versions of Marsaglia-Tsang tests. For many of the algorithms I tested, I used Marsaglia-Tsang tests as an initial screening method.

4 OVERVIEW OF THE SETS OF PRNGS CREATED FOR THIS STUDY

Over 50 fundamentally different classes of algorithms were generated. Within each of the broad classes of algorithms, dozens to hundreds of variations were produced where constants were changed, but not the operations. In total, with the assistance of several interns and contractors, I created and tested 12,779 specific algorithms. Of these, there were 8,238 algorithms where I only ran the two Marsaglia-Tsang tests. If an algorithm passed the two Marsaglia-Tsang tests on the first run, it would then be selected for one or more runs on all 114 tests. There were 4,541 algorithms where I ran all 114 tests on at least two different sets of 65 million numbers. Since this is a random sampling method, there would be some expected variation in test results. So, for the best performing algorithms I ran even more separate sets of 65 million to confirm performance. For just six algorithms I ran all 114 tests on 10 separate sets of 65 million numbers.

In every algorithm I tested there was some form of "nested" modulus functions (e.g., mod(mod(A,B)+C*mod(D,E),F)) and other operations. To meet the previously listed guidelines, the other operations were typically limited to multiplication, addition, and in some cases subtraction. Early PRNGs, like Linear Congruential Generators (Bolte 2011), vary the dividend in a modulus function (Knuth 1997) but I quickly found that algorithms varying the divisor, instead of the dividend, tended to perform well in the Dieharder tests. Then, all of algorithms using the varying divisor method would use additional nested modulus functions which varied the dividend.

Based on the results of the first few thousand algorithms, I settled on testing only variations of the following form (the right side is written in the syntax of Excel) in Equation (1). Algorithms of this form accounted for the majority of the test runs and they made up nearly all of the 8,238 quick tests (consisting of only the Marsaglia-Tsang tests).

$$\begin{aligned} R(Trial)=&\mathrm{mod}(\mathrm{mod}(\mathrm{mod}(999999999999989,\mathrm{mod}(Trial*T1,A1)*B1+C1)*D1,E1)*F1 \\ &+\mathrm{mod}(\mathrm{mod}(999999999999989,\ \mathrm{mod}(Trial*T2,A2)*B2+C2)*D2,E2)*F2,2\text{^}32) \end{aligned} \quad (1)$$

Where:

- R(Trial) is the random number produced for a given Trial. Trial is a counter representing a unique identifier for a trial in a simulation. For simplicity, only the Trial ID is shown.
- T1 and T2 are primes. If other dimensions besides trial were shown, there would be more of these primes used as coefficients in a simple linear formula to produce an integer as a dividend in this modulus function.
- A1 through F1 and A2 through F2 are other various constants, some of which are prime. They are all chosen to ensure that none produces a value beyond Excel's precision limit (10^15) and no dividend of a modulus function is more than 2^27 times the divisor.

In short, this class of algorithms is simply two terms with varying divisors, which are each sent through additional modulus functions of varying dividends, added together and then sent though a final modulus function with the divisor of 2^32 to produce the values required by Dieharder.

The constants would also have to be chosen in a way that increases eventual "coverage" of the range up to 2^32. In principle, if I generated far more values than what is required by Dieharder, I should see a large number of the possible unique values being generated. If I generated, say, 10 billion values then I would expect that less than 10% of the possible values the range up to 2^32 would not have been generated

at least once. Ideal coverage from a perfectly random source should generate every value in the range of 2^32 after 100 billion trials.

I made some other observations when comparing the performance of these algorithms. First, simplifying the algorithm much further by removing any operation greatly reduced the number of Dieharder tests that were passed. On the other hand, adding a third coefficient group or adding more levels of nesting the modulus functions did not improve performance without greatly increasing the length of the algorithm and reducing speed. About this level of complexity appears to be just the right tradeoff between complexity, speed and randomness test performance. Also, while the Excel constraints on precision (10^15) and modulus functions (the dividend<2^27*divisor) must be maintained, I found that pushing these constraints to their limits was a good practice in improving randomness test performance.

5 RESULTS OF THE DIEHARDER TESTS AND COMPARISON TO OTHER METHODS

After thousands of algorithms, I found that many of the algorithms in the form described above did well for a variety of constants. Among these, I chose one (algorithm identified as 703756) as that was among the better-performing algorithms but the data shown are an entirely new set of tests. If I would have chosen the best performer out of thousands of tests, I risk simply showing the "champion at coin-flipping." That is, I might just be selecting the luckiest algorithm so far. So I ran additional tests on this algorithm that were not run before. This algorithm did about as well as five other similar algorithms and the difference was well within sampling error on all five (for simplicity and clarity, the details of those aren't shown here). Also, the chosen algorithm had the mathematically highest coverage of the random number space, as described earlier.

For comparison, I also ran the Dieharder tests on a total of multiple sets of 65 million numbers generated using the random number functions in following methods: Excel (i.e., the "=Rand()" function), R, C and Python. I also ran Dieharder on two sets of 65 million numbers purchased from Amazon Web Services (AWS). AWS provides a method of generating large sets of random numbers primarily for use in cryptographic security (Amazon 2018). This would be an interesting comparison because one component of the AWS method is an undisclosed "natural" random source. Some examples of highly non-linear natural phenomenon include the random error from meteorological instruments (Haahr 2006; Haahr 2019) or photos of lava lamps (Liebow-Feeser 2017).

Table 1 shows the results of the tests. Documentation for Excel, R, and Python indicates they all use some form of MT (Microsoft 2019; Python 2019; Dutang and Kiener 2019) as their default generator. Surprisingly, however, they did not produce the same results. Some variation would be expected in a random sampling method but these results varied by much more than could be explained by chance. Python was the best performing of these algorithms, and R was significantly worse than expected even accounting for some random variation of results. As expected, C was the worst performing using an outdated linear congruential generator (LCG) in the srand() function. It should be noted that implementations of the MT exist for C, and the documentation is aware the older and default LCG implementations were insufficient for serious random-number generation needs.

Table 1: Results of Dieharder tests.

Number of WEAK/FAIL test results out of 114 (Remaining tests produced a "PASS")						
Test Set (Each set contains 65 million numbers)	Algorithm 703756	Python	Excel	R	C	AWS
1	3/1	2/2	10/7	4/73	0/87	4/1
2	5/2	4/1	5/8	5/74	1/87	3/1
3	2/0	1/1	5/7	2/74		
4	3/1	3/0	14/4	5/74		
5	6/0	5/0	12/5	4/73		
6	5/1	3/0	5/6	6/73		
7	3/1	3/0	10/8	5/74		
8	2/0	5/0	11/3	4/74		
9	6/0	2/0	7/9	2/75		
10	1/1	5/0	12/14	2/74		

Since this was a random sample of possible number sets, it can be seen that the test results varied from set to set. I used these results to compute the error for the estimate of what the mean result would be if I ran this for infinite trials. This is using a beta distribution as a method for estimating population proportion of WEAK and FAIL rates for the applied a simple method. Using this method, Figure 1 shows the 90% confidence intervals for the estimate of the WEAK and FAIL rates based on the observed samples. The results show that the performance of Python, AWS and Algorithm 703756 were indistinguishable from each other. Excel rand(), on the other hand, is clearly much worse than what could be attributed to chance. Note that one particularly surprising result is that AWS, which apparently uses a natural random number source, did not get a perfect score. Even though I only used two sets of 65 million numbers for AWS, instead of 10 like the others, the beta distribution indicated it might not do much better than Python or the 703756 algorithm.

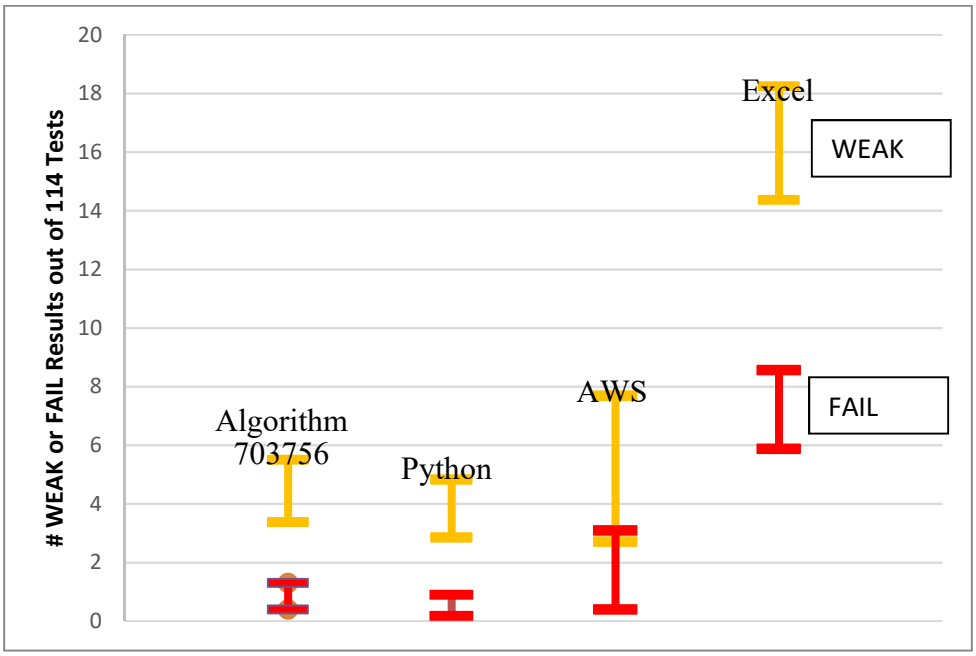

Figure 1: Sampling error around Dieharder tests results. This uses a Beta Distribution to with a prior of alpha=1 and beta=1 to compute a 90% confidence interval for estimate of mean weak/fails out Dieharder test suite. The top row of brackets are estimates of WEAK result rates and the bottom row are estimates of FAIL rates.

6 THE PROPOSED PRNG

The thousands of tests have led to the selection of the algorithm shown in Equation (2) for the proposed multi-dimensional, counter-based PRNG (for simplicity, it shows only the Trial ID dimension)**.**

$$R(Trial)=(\mathrm{mod}((\mathrm{mod}(\mathrm{mod}(999999999999989,\mathrm{mod}(Trial*2499997,7450589)*4658+7450581)*383,99991)*7440893+\mathrm{mod}(\mathrm{mod}(999999999999989,\mathrm{mod}(Trial*2246527,7450987)*7580+7560584)*17669,7440893))*1343,4294967296)+.5)/4294967296 \quad (2)$$

Note that a final operation, (R(Trial)+.5)/4294967296, is added to convert the 32-bit integer result to a 0 to 1 interval. Otherwise, it is algebraically equivalent to the form defined previously but expressed a bit differently.

Other dimensions are added by multiplying each counter (Trial ID, Variable ID, Entity ID, Time ID and Agent ID) times the following prime coefficients and adding them. See Table 2 for the prime coefficients needed for the other dimensions.

Table 2: Prime coefficients for five dimensions.

	1st Term	2nd Term
Trial	2499997	2246527
Var	1800451	2399993
Ent	2000371	2100869
Time	1796777	1918303
Agent	2299603	1624729

For any dimension excluded, the counter is defaulted to zero. For example if I wanted to use only the Trial ID and Variable ID the equation would default to Equation (3). This would be the minimum number of dimensions for simulations with multiple random variables.

$$R(Trial,Var)=(mod((mod(mod(999999999999989,mod(Trial*2499997+Var*1800451,7450589)*4658+7450581)*383,99991)*7440893+mod(mod(999999999999989,mod(Trial*2246527+Var*2399993,7450987)*7580+7560584)*17669,7440893))*1343,4294967296)+.5)/4294967296 \quad (3)$$

Even with the constraints on Excel, this formula has the capacity to uniquely identify 100 million variables, trials and entities as well as millions of time increments for each one and millions of agent ID's for each of those. See Table 3 for further details on each of the proposed dimensions.

Table 3: Further details on each of the dimensions.

Dimension	Description
Trial ID	This represents a unique identifier for a given scenario in a simulation. This 8 decimal digit identifier allows for up to 100 million unique trials for each variable in a model.
Variable ID	This is a unique identifier for a variable. It would be an 8-digit variable ID allowing for up to 100 million unique variables. For example, if "Monthly Demand for Product X" and "average time spent in activity Y" were variables in a model, they would each be given unique variable IDs. Organizations may structure their Variables IDs so that related variables are in groups. For example, perhaps all marketing and sales related variables have "11" for the first two digits and all cybersecurity related variables have "73" for the first two digits, and so on. Variable IDs could be assigned on an ad hoc basis but a large organization making many models with a lot of shared variables would want to develop an internal library of assigned variable IDs similar to an accountant's "chart of accounts."
Entity ID	This identifies an organization or some other category of users. A corporation or government agency may be assigned a unique 8 decimal digit Entity ID. Since this provides for 100 million potential entities, that should be enough for every business, not for profit and government agency that wants one on the planet. This is useful if there are models using random variables from many organizations do not have variables that produce the same random sequences. For example, many banks may use variables defined by the FDIC for "stress testing" to ensure banks are financially stable even during times of economic stress. The bank would want to ensure that internally defined variables with the same Variable ID are not correlated to the shared variables. The FDIC would supply the variable ID along with the Entity ID of the FDIC so that every bank using those variables produces the same sequence while avoiding duplicating the sequence of internally defined variables. A default Entity ID of 0 can be used by anyone as long as sharing variables would not be an issue.
Time ID	This identifies a particular time unit for a given variable/trial/entity combination. This allows one scenario for a given variable to contain an entire unique time series. A 7-digit time series ID would allow for time series containing 115 days of seconds, 19 years of minutes, or 27,397 years of days. This is an optional dimension. Variables that do not represent a time series use the default Time ID of 0.
Agent ID	This provides a fifth optional dimension for the counter-based PRNG. One possible use is as an identify for agents in agent-based modeling. If this ID is not used, the default value is 0.

7 CONCLUSION

Through millions of simulations, a five-dimensional PRNG was defined which can be implemented in any software, including Excel, while still producing the same results regardless of whether it is used in Excel or any other current common software language. The structure of the PRNG also gives the user a way to construct independent random numbers in a simple and organized fashion for many variables. Though the MT algorithm in Python was able to achieve similar pass rates on the Dieharder tests, the same performance was not seen in either Excel or R. Though the MT algorithm represents a substantial improvement over previous methods, the flexibility and ease of implementation of the five dimensional PRNG allows the construction of random numbers at least as good as the industry standard while allowing for trackability and uniformity across several software platforms.

REFERENCES

Amazon Web Services. 2018. AWS Key Management Service Cryptographic Details. https://d1.awsstatic.com/whitepapers/KMS-Cryptographic-Details.pdf, accessed 15th May.

Bolte, J. 2011. Linear Congruential Generators. https://demonstrations.wolfram.com/LinearCongruentialGenerators/, accessed 6th May.

Dutang, C. and P. Kiener. 2019. CRAN Task View: Probability Distributions. https://cran.r-project.org/web/views/Distributions.html, accessed 15th May.

Haahr, M. 2006. "Random Numbers." In *Encyclopedia of Measurements and Statistics,* edited by N. J. Salkind. Thousand Oaks, California: Sage Publications.

Haahr, M. 2019. Random.org: True Random Number Service. https://www.random.org, accessed 15th May.

Hubbard, D. 2009. *The Failure of Risk Management: Why It's Broken and How to Fix It*. Hoboken, New Jersey: Wiley.

Hubbard, D. 2014. *How to Measure Anything: Finding the Value of Intangibles in Business*. [3rd ed.]Hoboken, New Jersey: Wiley.

Kirmse, M. and S. Savage. 2014. "Probability Management 2.0." *OR/MS Today* 41(5):30-33.

Knuth, D. 1997. *Seminumerical Algorithms. The Art of Computer Programming*. [3rd ed.]Reading, MA: Addison-Wesley Professional.

L'Ecuyer, P. 2017. "History of Uniform Random Number Generation." In *Proceedings of the 2017 Winter Simulation Conference*, edited by W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Liebow-Feeser, J. 2017. LavaRand in Production: The Nitty-Gritty Technical Details. https://blog.cloudflare.com/lavarand-in-production-the-nitty-gritty-technical-details/, accessed 14th May.

Marsaglia, G., and W. W. Tsang. 2002. "Some Difficult-to-Pass Tests of Randomness." *Journal of Statistical Software*, 7(3):1–9.

Matsumoto, M., and T. Nishimura. 1998. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator." *ACM Transactions on Modeling and Computer Simulation*, 8(1): 3–30.

Microsoft. 2019. RAND Function. https://support.office.com/en-us/article/rand-function-4cbfa695-8869-4788-8d90-021ea9f5be73, accessed 15th May.

Python. 2011. 8.6. Random - Generate Pseudo-random Numbers https://docs.python.org/release/3.2/library/random.html, accessed 15th May.

Salmon, J. K., M. A. Moraes, R. O. Dror, and D. E. Shaw. 2011. "Parallel Random Numbers: As Easy as 1, 2, 3". In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, New York: Association for Computer Machinery.

Savage, S. 2016. "Monte Carlo for the Masses." *Analytics Magazine* September/October 2016:64-68.

Savage, S. L. 2012. "Distribution Processing and the Arithmetic of Uncertainty." *Analytics Magazine* November/December 2012:28-32.

Ubuntu. 2019. Dieharder - A testing and benchmarking tool for random number generators. http://manpages.ubuntu.com/manpages/bionic/man1/dieharder.1.html, accessed 15th May.

AUTHOR BIOGRAPHY

DOUGLAS W. HUBBARD is the inventor of the Applied Information Economics (AIE) method and founder of Hubbard Decision Research (HDR). He is the author of one of the best-selling business statistics books of all time, *How to Measure Anything: Finding the Value of Intangibles in Business*. His other books include *The Failure of Risk Management: Why It's Broken and How to Fix It, Pulse: The New Science of Harnessing Internet Buzz to Track Threats and Opportunities* and *How to Measure Anything in Cybersecurity Risk*. He has sold over 130,000 copies of his books in eight different languages and his books are used in courses in over a dozen major universities. His first two books are now required reading for the Society of Actuaries exam preparation. Mr. Hubbard's career has focused on the application of AIE to solve current business issues facing today's corporations. Mr. Hubbard has completed over 100 risk/return analyses of large, critical projects, investments and other management decisions in the last 20 years. AIE is the practical application of several fields of quantitative analysis including Bayesian analysis, Monte Carlo simulations, and many others. Mr. Hubbard's consulting experience and financial analysis totals over 30 years and spans many industries including pharmaceuticals, insurance, banking, utilities, cybersecurity, interventions in developing economies, mining, federal and state government, entertainment media, military logistics, and manufacturing. His email address is dwhubbard@hubbardresearch.com and his website is https://www.hubbardresearch.com/.