Fast Gaussian Distributed Pseudorandom Number Generation in Java via the Ziggurat Algorithm

Vincent A. Cicirello

Computer Science
Stockton University
Galloway, NJ 08205 USA
https://www.cicirello.org/

Technical Report ALG-24-009

May 2024

Fast Gaussian Distributed Pseudorandom Number Generation in Java via the Ziggurat Algorithm

Vincent A. Cicirello

Computer Science
Stockton University
Galloway, NJ 08205 USA
https://www.cicirello.org/

Technical Report ALG-24-009

Abstract

We report on experiments with the ziggurat algorithm for generating Gaussian distributed random numbers. The study utilizes our open source Java implementation that was introduced originally for Java 11 at a time when the Java API only provided the much slower polar method. Our Java implementation of the ziggurat algorithm is a port of the GNU Scientific Library's C implementation. Java 17 introduced a significant overhaul of pseudorandom number generation, including several modern pseudorandom number generators (PRNGs) as well as additional functionality, among which includes switching from the polar method to a modified ziggurat algorithm. In the experiments of this paper, we explore whether there is still a need for our implementation for Java 17+ applications. Our results show that Java 17's modified ziggurat is faster than our implementation for the PRNGs that support it. However, Java 17+ continues to use the polar method for the legacy PRNGs Random, SecureRandom, and ThreadLocalRandom. The linear congruential method of Java's Random class lacks the statistical properties required by Java's modified ziggurat implementation; and SecureRandom and ThreadLocalRandom unfortunately use the polar method as a side-effect of extending Random. Our implementation of the original ziggurat algorithm does not require the same statistical properties of the underlying PRNG as Java 17's optimized version, and can be used with any of these PRNGs, and is especially relevant where pre-Java 17 support is required.

Keywords: algorithm; Gaussian; Java; normal distribution; open source; pseudorandom number generator; random numbers; ziggurat

ACM Classes: F.2.1; I.1.2; D.2.13; G.3; G.4

MSC Classes: 68Q87; 68W20; 60-04; 60G15; 65C10

1 Introduction

Efficiently generating Gaussian distributed random numbers is often important in modeling and simulation contexts. The most commonly encountered algorithm for generating Gaussian random variates is the polar method [7], which has been around for decades. However, there are now modern alternatives that are

 https://orcid.org/0000-0003-1072-8559

Table 1: Important URLs for the open source ziggurat library

Source	`https://github.com/cicirello/ZigguratGaussian`
Maven	`https://central.sonatype.com/artifact/org.cicirello/ziggurat`
DOI	`https://doi.org/10.5281/zenodo.4106912`

significantly faster and provide higher quality results, such as the ziggurat method and its variations [10, 9, 16, 11].

Our original motivation for efficiently generating Gaussian random variates was in the context of implementing Gaussian mutation [6] for an evolutionary algorithm (EA) [3]. The EA in question was implemented in Java, which at the time (e.g., Java 8) provided only the polar method for Gaussian random number generation. Runtime profiling of the EA revealed this as a bottleneck. We ported the GNU Scientific Library's C implementation [16] of the ziggurat algorithm [10, 9] to Java to optimize the EA runtime. We also carefully considered other random number related elements of the EA, such as the choice of pseudorandom random number generator (PRNG) itself. We demonstrated that an EA relies so heavily on random number generation that switching to the ziggurat algorithm and making a few other PRNG related optimizations resulted in an EA that is 20% to 25% faster [3]. We since released our Java implementation of the ziggurat algorithm as a small open source library, which we utilize in the experiments of this report. See Table 1 for URLs to the source code as well as to the artifacts in the Maven Central Repository.

Among other things, Java 17 overhauled random number generation [13], including introducing several modern PRNGs [1, 15], a hierarchy of interfaces for different categories of PRNG, as well as other random number improvements. One of those improvements was providing an implementation of McFarland's modified ziggurat algorithm [11] for Gaussian random variates that is faster than the original. It did not totally replace the polar method, however, as the Random class's linear congruential generator (LCG) [7] does not meet the quality of randomness required by the modified ziggurat method. Thus, Random retains the slow polar method. The other legacy PRNG classes that extend Random have unfortunately also retained the polar method, as a consequence of subclassing Random, despite providing the required quality of randomness.

In this paper, we report on experiments comparing our implementation of the original ziggurat algorithm with the Gaussian random variate implementations provided in Java 17, which is the polar method for some PRNG classes, and the modified ziggurat for others. Unlike McFarland's modified ziggurat, our implementation of the original ziggurat is applicable for all of Java's PRNG classes, as it does not rely on the low-order bits of random 64-bit longs. We will see that Java's new modified ziggurat is faster than the original ziggurat in cases where it is applicable. But in cases where the modified ziggurat is not applicable (e.g., when used with a LCG) or where the modified ziggurat is not otherwise provided that the original ziggurat as implemented in our library is significantly faster than the alternative, and continues to be relevant for applications that must support pre-Java 17 runtime environments.

We proceed as follows. We explain our methodology in Section 2. We discuss the experimental results in Section 3. We conclude in Section 4 with a discussion and observations of when our ziggurat library is still applicable, and the better alternatives that are available in other cases.

2 Methodology

We use OpenJDK 64-Bit Server VM version 17.0.2 in the experiments on a Windows 10 PC with an AMD A10-5700, 3.4 GHz processor and 8GB memory. We experiment with three of Java's legacy PRNGs:

- Random: implementation of a LCG [7];

Listing 1: Coercing ThreadLocalRandom to use Java 17's modified ziggurat algorithm

```
RandomGenerator wrappedThreadLocalRandom =
  new RandomGenerator() {
    @Override
    public long nextLong() {
      return ThreadLocalRandom.current().nextLong();
    }
  };

// You can then generate Gaussian random numbers using Java 17's
// modified ziggurat with ThreadLocalRandom, with calls like:
double r1 = wrappedThreadLocalRandom.nextGaussian();

// Whereas calls like the following would use the polar method:
double r2 = ThreadLocalRandom.current().nextGaussian();
```

- SplittableRandom: implementation of SplitMix [14] algorithm, which is a faster optimized version of the DotMix [8] algorithm, and which passes the DieHarder [2] tests; and
- ThreadLocalRandom: implementation of the same PRNG algorithm as SplittableRandom, but manages thread local seed data in multi-threaded scenarios.

For Gaussian random number generation, and for each of the above PRNGs, we consider the following:

- Ziggurat: This is our implementation (see Table 1) of the ziggurat algorithm [10, 9], which is a Java port of the GNU Scientific Library's C implementation [16].
- Java 17: We compare our ziggurat implementation to Java 17's builtin support for Gaussian random numbers, which is the polar method [7] for the Random and ThreadLocalRandom classes, and is McFarland's modified ziggurat algorithm [11] for SplittableRandom.
- Wrapped: For ThreadLocalRandom, we consider a third option where we coerce the use of Java 17's modified ziggurat implementation instead of the polar method by implementing the RandomGenerator interface in a way that wraps calls to ThreadLocalRandom.current().nextLong() in the only required method of the RandomGenerator interface, the RandomGenerator.nextLong() method. This forces the use of the default RandomGenerator.nextGaussian() method, which is McFarland's modified ziggurat algorithm [11], but utilizing ThreadLocalRandom as the PRNG. See Listing 1.

Listing 2 illustrates usage of our ziggurat implementation versus the Java API's builtin Gaussian support.

We use the Java Microbenchmark Harness (JMH) [12] to implement our experiments. For each experiment condition (combination of PRNG and Gaussian implementation), we use five 10-second warmup iterations to ensure that the Java JVM is properly warmed up, and we likewise use five 10-second iterations for measurement. We measure and report average time per operation in nanoseconds, along with 99.9% confidence intervals.

The code to reproduce our experiments is available on GitHub, as is the data from our runs of the experiments. See Table 2 for the relevant URLs.

Listing 2: Using our ziggurat implementation vs Java's builtin functionality

```
Random random = new Random();
SecureRandom secure = new SecureRandom();
SplittableRandom splittable = new SplittableRandom();

// Calls to nextGaussian() for Random, SecureRandom, and ThreadLocalRandom
// classes use the polar method:
double g1 = random.nextGaussian();
double g2 = secure.nextGaussian();
double g3 = ThreadLocalRandom.current().nextGaussian();

// The equivalent call for SplittableRandom uses Java's modified ziggurat:
double g4 = splittable.nextGaussian();

// To use our implementation of the original ziggurat algorithm with Random, SecureRandom,
// ThreadLocalRandom, and SplittableRandom, respectively, pass the PRNG instance to
// ZigguratGaussian.nextGaussian as a parameter, or no parameter for ThreadLocalRandom:
double z1 = ZigguratGaussian.nextGaussian(random);
double z2 = ZigguratGaussian.nextGaussian(secure);
double z3 = ZigguratGaussian.nextGaussian();
double z4 = ZigguratGaussian.nextGaussian(splittable);
```

Table 2: Reproducible results: URLs to experiment code and data

Code	`https://github.com/cicirello/ZigguratGaussian/experiment/timing17`
Data	The file /results17.txt in above directory

3 Results

Table 3 summarizes the results. Our implementation of the ziggurat algorithm for generating Gaussian distributed random numbers is 83.12% faster than Java 17's polar method implementation when the legacy Random class is used; and our ziggurat implementation is 87.72% faster than Java 17 in the case of the ThreadLocalRandom class. We expect that we would find approximately the same significant advantage in the case of Java's SecureRandom class, although we did not test this, since that class likewise uses the slow polar method.

The case of SplittableRandom, however, is different. In Java 17, the SplittableRandom class uses McFarland's modified ziggurat algorithm [11], which in our experiments is 11.52% faster than our implementation of the older version of the ziggurat algorithm. When we use our trick from Listing 1 to coerce the use of Java 17's modified ziggurat algorithm for the ThreadLocalRandom class, we find that it is 4.67% faster than our implementation of the original ziggurat algorithm. Java 17 introduced several modern PRNGs [13], including Xor-Based Generators (XBG) [1] and several variations of LXM [15], which combine an LCG and XBG with a mixing function. All of these newly introduced PRNGs use Java 17's modified ziggurat algorithm when generating Gaussian random numbers. So although we did not include these in our experiments, we should find the same pattern as we found with SplittableRandom, namely that Java's modified

Table 3: Results: Average time per operation in nanoseconds

PRNG class	Ziggurat	Java 17	Wrapped
Random	17.393 ± 0.300 ns/op	103.037 ± 0.951 ns/op	n/a
SplittableRandom	10.089 ± 0.139 ns/op	8.927 ± 0.026 ns/op	n/a
ThreadLocalRandom	10.901 ± 0.032 ns/op	88.774 ± 2.096 ns/op	10.392 ± 0.121 ns/op

ziggurat should be a bit faster than our implementation of the original ziggurat algorithm.

Java 17's implementation of McFarland's modified ziggurat depends on the quality of the low-order bits from calls to the nextLong() method. The ThreadLocalRandom class meets those expectations, as it implements the same PRNG algorithm as the SplittableRandom class. Its use of the slow polar method is due to subclassing the Random class, whose use of an LCG leads to low-order bits that do not meet the requirements of Java's modified ziggurat implementation. Thus, it is safe to use our trick from Listing 1 with the ThreadLocalRandom class, but not with the Random class. Our implementation of the original ziggurat algorithm, as ported to Java from the C implementation of the GNU Scientific Library, does not depend on the quality of low-order bits of random longs and should be safe to use with any of the PRNG classes.

4 Conclusions

From our experimental results, we make the following observations:

- Our open source implementation of the original ziggurat algorithm (see library details in Table 1) provides a very significant performance advantage for the Random, ThreadLocalRandom, and SecureRandom classes as compared to the Java API's polar method implementation, with our results (Table 3) showing that our ziggurat implementation is over 83% and 87% faster in the cases of Random and ThreadLocalRandom, respectively. Unlike Java 17's modified ziggurat, our ziggurat implementation is applicable even in the case of weaker PRNGs like the LCG implemented by Random.

- For pre-Java 17 applications, our ziggurat library provides a Gaussian implementation for SplittableRandom, where the Java API itself lacks Gaussian support entirely.

- For Java 17+ applications, Java's modified ziggurat algorithm, implemented by the default interface method RandomGenerator.nextGaussian() is more than 11% faster than our implementation of the original ziggurat in the case of the SplittableRandom class (Table 3), and presumably for all of the modern PRNGs [13, 15, 1] introduced in Java 17. Note that we did not verify this for the new Java 17+ PRNGs as our ziggurat library is designed to support Java 11+, which does not have the RandomGenerator interface. Thus, modifying our ziggurat library to support the RandomGenerator interface would break Java 11 compatibility.

- For Java 17+ applications that use ThreadLocalRandom, another faster option is to utilize our trick from Listing 1 to gain access to Java's implementation of the modified ziggurat algorithm. Although in such a case, utilizing our ziggurat library may be a cleaner approach while almost as fast.

A more comprehensive library of randomization utlities $\rho\mu$ [5] has used our implementation of the ziggurat algorithm for $\rho\mu$ versions 1.0.0 through 4.0.0. However, beginning with version 4.1.0, the $\rho\mu$ library

Table 4: Important URLs for the open source $\rho\mu$ library

Source	`https://github.com/cicirello/rho-mu`
Maven	`https://central.sonatype.com/artifact/org.cicirello/rho-mu`
Website	`https://rho-mu.cicirello.org/`

has been updated based on the insights of these experiments. It now relies on the Java API's modified ziggurat by default for all PRNGs that support it, and for ThreadLocalRandom it uses the Listing 1 trick as a replacement for our ziggurat implementation. It does, however, maintain a utility class with our implementation of the original version of the ziggurat algorithm to continue to support fast Gaussian random number generation for the legacy Random and SecureRandom classes. The $\rho\mu$ library requires a minimum version of Java 17, however, so applications supporting pre-Java 17 may still benefit from our ziggurat library, which only requires Java 11+. See Table 4 for relevant URLs to the $\rho\mu$ library. Among other purposes, $\rho\mu$ supports our original motivation of accelerating the runtime of EAs by optimizing random number generation, serving as a dependency of the open source evolutionary computation library Chips-n-Salsa [4].

References

[1] David Blackman and Sebastiano Vigna. Scrambled linear pseudorandom number generators. *ACM Transactions on Mathematical Software*, 47(4), September 2021. doi:10.1145/3460772.

[2] Robert G Brown, Dirk Eddelbuettel, and David Bauer. *Dieharder: A random number test suite*, 2013. `https://www.phy.duke.edu/~rgb/General/dieharder.php`.

[3] Vincent A. Cicirello. Impact of random number generation on parallel genetic algorithms. In *Proceedings of the Thirty-First International Florida Artificial Intelligence Research Society Conference*, pages 2–7. AAAI Press, May 2018.

[4] Vincent A. Cicirello. Chips-n-salsa: A java library of customizable, hybridizable, iterative, parallel, stochastic, and self-adaptive local search algorithms. *Journal of Open Source Software*, 5(52):2448, August 2020. doi:10.21105/joss.02448.

[5] Vincent A. Cicirello. $\rho\mu$: A java library of randomization enhancements and other math utilities. *Journal of Open Source Software*, 7(76):4663, August 2022. doi:10.21105/joss.04663.

[6] R. Hinterding. Gaussian mutation and self-adaption for numeric genetic algorithms. In *Proceedings of 1995 IEEE International Conference on Evolutionary Computation*, pages 384–389, 1995. doi:10.1109/ICEC.1995.489178.

[7] Donald E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1998.

[8] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 193–204. ACM, 2012. doi:10.1145/2145816.2145841.

[9] Philip H. W. Leong, Ganglie Zhang, Dong-U Lee, Wayne Luk, and John Villasenor. A comment on the implementation of the ziggurat method. *Journal of Statistical Software*, 12(7):1–4, 2005. doi:10.18637/jss.v012.i07.

[10] George Marsaglia and Wai Wan Tsang. The ziggurat method for generating random variables. *Journal of Statistical Software*, 5(8):1–7, 2000. doi:10.18637/jss.v005.i08.

[11] Christopher D. McFarland. A modified ziggurat algorithm for generating exponentially and normally distributed pseudorandom numbers. *Journal of Statistical Computation and Simulation*, 86(7):1281–1294, 2016. doi:10.1080/00949655.2015.1060234.

[12] OpenJDK.org. *Java Microbenchmark Harness (JMH) Version 1.37*, 2023. `https://github.com/openjdk/jmh`.

[13] Guy Steele. Jep 356: Enhanced pseudo-random number generators. JEP 356, OpenJDK, 2017. URL `https://openjdk.org/jeps/356`.

[14] Guy L. Steele, Jr., Doug Lea, and Christine H. Flood. Fast splittable pseudorandom number generators. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 453–472. ACM, 2014. doi:10.1145/2660193.2660195.

[15] Guy L. Steele Jr. and Sebastiano Vigna. Lxm: better splittable pseudorandom number generators (and almost as fast). *Proceedings of the ACM on Programming Languages*, 5, October 2021. doi:10.1145/3485525.

[16] Jochen Voss. The ziggurat method for generating gaussian random numbers, 2014. URL `https://www.seehuhn.de/pages/ziggurat`.