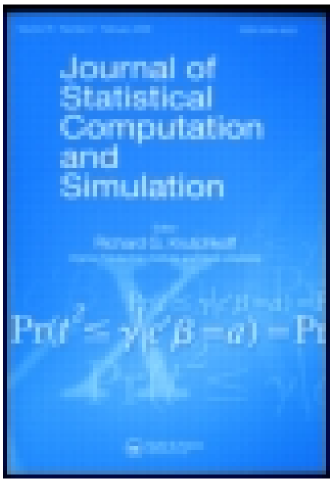

Journal of Statistical Computation and Simulation
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/gscs20

A modified ziggurat algorithm for generating exponentially and normally distributed pseudorandom numbers

Christopher D. McFarland[a]
[a] Department of Biology, Stanford University, Stanford, CA, USA
Published online: 24 Jun 2015.

To cite this article: Christopher D. McFarland (2015): A modified ziggurat algorithm for generating exponentially and normally distributed pseudorandom numbers, Journal of Statistical Computation and Simulation, DOI: 10.1080/00949655.2015.1060234

To link to this article: http://dx.doi.org/10.1080/00949655.2015.1060234

Journal of Statistical Computation and Simulation, 2015
http://dx.doi.org/10.1080/00949655.2015.1060234

A modified ziggurat algorithm for generating exponentially and normally distributed pseudorandom numbers

Christopher D. McFarland*

Department of Biology, Stanford University, Stanford, CA, USA

(*Received 10 April 2014; accepted 5 June 2015*)

The ziggurat algorithm is a very fast rejection sampling method for generating pseudorandom numbers (PRNs) from statistical distributions. In the algorithm, rectangular sampling domains are layered on top of each other (resembling a ziggurat) to encapsulate the desired probability density function. Random values within these layers are sampled and then returned if they lie beneath the graph of the probability density function. Here, we present an implementation where ziggurat layers reside completely beneath the probability density function, thereby eliminating the need for any rejection test within the ziggurat layers. In the new algorithm, small overhanging segments of probability density remain to the right of each ziggurat layer, which can be efficiently sampled with triangularly shaped sampling domains. Median runtimes of the new algorithm for exponential and normal variates is reduced to 58% and 53%, respectively (collective range: 41–93%). An accessible C library, along with extensions into Python and MATLAB/Octave are provided.

1. Introduction

Random numbers are used in a variety of applications: modelling, optimization, and cryptography, to name a few. However, computers are designed to behave deterministically, thereby making truly random number generation from a computer difficult. Pseudorandom numbers (PRNs), or deterministically generated random numbers, are generally used instead of truly random numbers. Because of their wide-range of applications, PRNs have a long history of study. PRN generators (PRNGs) most often work by transforming an initial single random number, or 'seed', into a new PRN, which is then transformed into the next PRN ad infinitum. Although PRNGs are deterministic, they nevertheless exhibit the important properties of truly random numbers: large periodicity, equidistribution, and discontinuity.[1] Initial PRNGs generally return uniformly distributed variates, as they are easily constructed from random bit sequences. Other sampling distributions are then generated by transforming uniform PRNs by downstream algorithms. These downstream transformations often consume more computational time than the initial uniform PRNG, so for some stochastic algorithms, they constitute the primary bottleneck.

The ziggurat algorithm (ZA) is the most common method to obtain non-uniformly distributed PRNs. It was first proposed in the early 1960s [2] and has since been modified many times.[3–5] It is currently the fastest method available on modern CPUs,[3,6] although other methods of comparable speed exist.[7] The ZA outperforms other conceptually simpler algorithms like the

*Email: cmcfarl2@stanford.edu

Box–Muller transformation or inverse transform sampling because these methods typically rely on transcendental functions that are evaluated using lengthy numerical routines.[8]

The ZA is a three-step process for generating random numbers using rejection sampling:

(1) The desired probability density function $f(x)$ (excluding its tail) is inscribed by a set of rectangular boxes, resembling a ziggurat.
(2) A random uniformly distributed point (x, y) within a randomly chosen box (or the tail) is sampled.
(3) If this point (x, y) resides beneath the desired probability density function, i.e. if $y < f(x)$, then x is returned; otherwise the point is 'rejected' and a new point is generated and tested.

To faithfully sample $f(x)$, the ziggurat layers must fully inscribe $f(x)$. Excess sampling area, however, reduces computational speed, as more points will be rejected. Thus, layers are carefully designed to maximize the efficiency of rejection sampling. Additionally, the algorithm can be further accelerated by heuristically identifying values of x that can be preemptively returned without any rejection test (described below).

Here, we present a modified ZA that creates rectangular layers that lie completely beneath the graph of $f(x)$, rather than completely containing $f(x)$. This eliminates the need for any rejection test in these layers, but also leaves short gaps of probability mass that are sampled with special overhang layers in a small minority of iterations. By eliminating the need to rejection sample most PRNs, and by sampling these small gaps of probability mass efficiently, PRN generation is greatly accelerated. In the next section, the modified algorithm is described in detail alongside the traditional ZA. We then discuss the implementation and performance of this new algorithm relative to the best alternatives. The appendix presents code, affirms the random properties of the generators, and discusses additional minor optimizations that improved performance.

2. Description of the algorithm

The Traditional ZA. In a ZA, the hypograph of a sampled probability distribution function $f(x)$ is contained by a stack of rectangular ziggurat layers. All ziggurat layers contain the exact same area by design. Thus, evenly distributed points encapsulating $f(x)$ can be generated each iteration by selecting a random rectangular layer, using a uniform random integer $i \in [0, i_{\max})$, and then uniformly sampling a point $\{x = U_1 X_i,\ y = F_{i-1} + U_2(F_i - F_{i-1})\}$ within the layer. Here, $U_1, U_2 \in [0, 1)$ denote uniformly distributed PRNs, while X_i and F_i denote the ziggurat layer's length and height. These length and height values are pre-calculated and stored in lookup tables, thereby making ZA algorithms comparatively faster on systems with rapid memory access (e.g. modern CPUs with low-latency caches, but not GPUs [7]).

ZAs accelerate computation because the vast majority of points within the ziggurat layers can be a priori guaranteed to lie beneath $f(x)$ [3,5] (Figure 1). For monotonically decreasing and symmetric distributions, these preemptive acceptance regions constitute all points for which $f(x) > F_i$, i.e. points to the left of the leftmost value of the probability density function. Hence, if $U_1 < k_i = X_{i+1}/X_i$, then $y < f(x)\ \forall\ y \in [F_{i-1}, F_i)$ and x can be immediately returned without generating y nor calculating $f(x)$. If $U_1 < k_i$ fails, however, then the algorithm must perform the rejection test in Step 3. Shortcutting Step 3 greatly accelerates computation when $f(x)$ is expensive to compute, e.g. the exponential and normal distributions implemented here. Unfortunately, the traditional exponentially distributed ZA still rejection tests 2.2% of all points when the number of ziggurat layers is optimal ($i_{\max} = 256$).[3]

$f(x)$ often contains a tail that cannot be inscribed by ziggurat layers. Instead, it must be sampled via alternate methods: either inverse transform sampling [9] or, in most cases, ad hoc. We

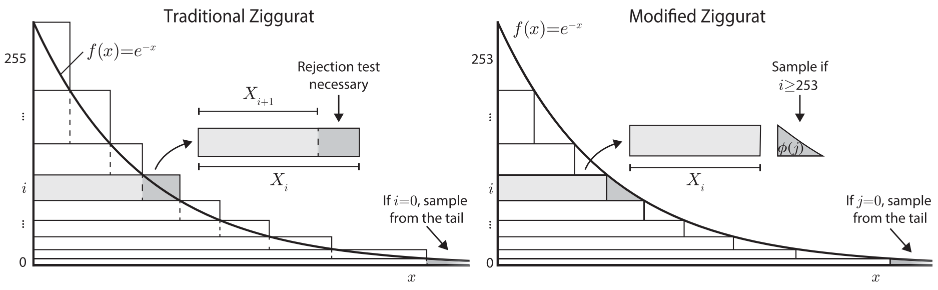

Figure 1. Ziggurat layers in the modified algorithm lie completely beneath the probability density function. In the *Traditional ZA*, equal-area layers encapsulate the desired distribution $f(x)$ (excluding the tail). Encapsulating $f(x)$ with rectangular layers requires extending the layers beyond $f(x)$. For faithful sampling of the distribution defined by $f(x)$, uniformly distributed points $\{x, y\}$ within the layer must be rejected if they reside above $f(x)$. Points can be implicitly accepted, avoiding this time-consuming rejection test, if they lie beneath $f(x)$ for any value of y ($x < X_{i+1}$, light-grey areas). 2.2% of all points are rejection-tested when 256 layers are used; dark-grey area. When $i = 0$, the tail is sampled via alternate methods. In the *Modified (new) ZA*, layers reside completely beneath $f(x)$. Thus, all points within the ziggurat layers are accepted without ever considering a rejection test. However, small gaps of probability mass [1.6% of the total probability mass when $f(x) = \mathrm{e}^{-x}$ and $i \in [0, 256)$] overhang to the right of each layer. These overhangs, with area $\phi(j)$, are sampled using an efficient rejection-sampling method.

do not improve upon these approaches here and, instead, reuse previous techniques to sample the exponential [6] and normal [3] tails. Overall, the ZA is ideal for distributions with infrequent sampling from the tail, i.e. not heavy-tailed distributions. For the exponential distribution with $i_{\max} = 256$, ziggurat layers sample all but 0.04% of total probability mass.

The Modified (New) ZA. The modified ZA presented here differs from the traditional algorithm in one key manner: layers lie completely beneath $f(x)$ (Figure 1), whereas in the traditional algorithm, layers completely contained $f(x)$. A rejection test becomes only necessary within small gaps of probability mass that emerge to the right of each layer. These small gaps, along with the tail, are sampled in proportion to their area: $\phi(j) \propto \int_{X_i}^{X_{i-1}} f(x) - f(X_{i-1})\,\mathrm{d}x$, using a simple new, rarely called procedure (described below). The new approach reduces runtime for two reasons: (i) The comparison $U_1 < k_i$ is avoided within the ziggurat layers, and (ii) a greater fraction of points can now be preemptively accepted.

Table 1 describes execution in the new approach, relative to the old approach, in detail.

To lie completely beneath $f(x)$, ziggurat layers must extend until their *upper-right* corner coincides with $f(x)$ (traditionally, their *lower-right* corner coincides with $f(x)$). The position of this upper-right corner becomes $\{X_i,\ F_i = f(X_i)\}$ in the new approach, where X_i, F_i remain the length and height of each layer. Like the traditional ZA, the lower-left corner sits immediately above the previous layer and begins at $x = 0$, i.e. $\{0, F_{i-1}\}$. Also like the traditional algorithm, layers are equal in area; however, this area is slightly smaller in the new algorithm because layers are now squeezed beneath $f(x)$. To evenly sample $f(x)$ using the uniform random integer $i \in [0, i_{\max})$, each layer's area in the new algorithm must be exactly $1/i_{\max}$.

The lengths of the new ziggurat layers X_i are then defined using this volume constraint from above:

$$\frac{1}{i_{\max}} = X_i[f(X_i) - f(X_{i-1})].$$

This iterative equation is solvable numerically using the Bisection or Secant Method. The first layer begins with its lower-left corner at the origin $\{0, 0\}$, such that $1/i_{\max} = X_0 f(X_0)$, and subsequent layer lengths are continually calculated until no more layers can be created.

Fewer than $i_{\max}$ rectangular layers will fit beneath $f(x)$, as each layer is exactly $1/i_{\max}$ in area, yet additional probability overhangs remain. Indeed, the total number of layers in the modified algorithm $L_{\max}$ cannot be determined until the last layer is calculated, which for an exponential

Table 1. Pseudocode comparison of modified and traditional ZAs.

Traditional algorithm	Modified algorithm
1: $i \sim I(0, i_{\max})$	1: $i \sim I(0, i_{\max})$
2: $U_1 \sim U(0,1)$	2: **if** $i < L_{\max}$ **then return** $X_i U(0,1)$
3: **if** $U_1 < k_i$ **then return** $X_i U_1$	3: $j \sim A$
4: **if** $i == 0$ **then return** a value from the tail	4: **if** $j == 0$ **then return** a value from the tail
5: $x \leftarrow X_i U_1$	5: $x \sim U(X_j, X_{j+1})$
6: $y \sim U(F_{i-1}, F_i)$	6: $y \sim U(F_{j-1}, F_j)$
7: **if** $y < f(x)$ **then return** x	7: **if** $y < f(x)$ **then return** x
8: **goto** 1.	8: **goto** 5.
Operations executed in the common case	
Traditional algorithm	Modified algorithm
97.8% probability of exit at Step 3.*	98.4% probability of exit at Step 2.*
1. Sample $I(0, i_{\max})$	1. Sample $I(0, i_{\max})$
2. Assign i	2. Assign i
3. Sample $U(0,1)$	3. Compare $i < L_{\max}$
4. Assign U_1	4. Sample $U(0,1)$
5. Lookup k_i	5. Lookup X_i
6. Compare $U_1 < k_i$	6. Multiply $U(0,1)X_i$
7. Lookup X_i	
8. Multiply $U_1 X_i$	

* For $i_{\max} = 256$ $I(a,b) \rightarrow$ Uniform PRNG over $[a,b) \cap \mathbb{Z}$ $U(a,b) \rightarrow$ Uniform PRNG over $[a,b)$ In practice, $U(a,b) = a + U(0,1)(b-a)$ A is a discrete probability distribution with probability mass function: $\phi(j) = \frac{i_{\max}}{i_{\max} - L_{\max}} \int_{X_i}^{X_{i-1}} f(x) - f(X_{i-1})\,\mathrm{d}x$ for $j \in [0, L_{\max}]$.$F_i = f(X_i)$

distribution with $i_{\max} = 256$ (the fastest value of $i_{\max}$ tested, see appendix) is 252. Thus, the probability mass overhangs in the exponential case consume only $\frac{4}{256} = 1.6\%$ of all samples.

In the modified algorithm, if the rectangle chosen is less than $L_{\max}$, then x is immediately drawn and returned – eliminating two operations in the common case: the lookup of k_i and the assignment of $U(0,1)$ to the temporary variable U_1 (Table 1). The new algorithm also replaces a 64-bit integer comparison with an 8-bit integer comparison that may accelerate runtime on some architectures. Because execution in the common case dominates runtime and requires only a few operations, these small changes nevertheless reduce runtime by 30% on average, although speedups vary considerably (Section 4).

Like the traditional ZA, the new algorithm uses three pre-calculated tables: the lengths X_i, and heights $F_i = f(X_i)$ of each ziggurat layer, and (in lieu of k_i) the probability mass within each overhang conditional upon the common-case failing – specified by $\phi(j)$ (Table 1).

Sampling an overhang j, from the distribution A defined by $\phi(j)$, occurs when sampling in the common case fails ($i \geq L_{\max}$, Step 3). Sampling j is performed very quickly, in $\mathcal{O}(1)$ operations, using the Alias method.[10]

Unlike the traditional algorithm, the probability $\phi(j)$ represents the actual quantity of probability mass beneath $f(x)$ for overhang j (not the area of the sampling box that subsumes the overhang). As such, for proper sampling, the overhang must be continually sampled until an acceptable point is found (i.e. if the rejection test fails, then return to Step 5 – *not* Step 1). This choice, of re-sampling from within the layer, is not essential to the new approach; however, it does simplify $\phi(j)$.

The new algorithm also increases the likelihood that a point is selected without a rejection test. This difference in likelihood will diminish as $i_{\max}$ increases. Moreover, for the optimum value of $i_{\max} = 256$, the likelihood of early exit increases only by 0.8% for the exponential distribution (from 97.8% to 98.4%). Hence, this change is relatively small compared to the other improvements.

Sampling from the tail, Step 3, is not improved upon in the new algorithm and unique to each probability distribution. The exponential distribution's tail can be quickly sampled by noting that the distribution is memoryless, i.e. the tail is, itself, an exponential distribution.[6] Hence, recursive calls to the algorithm can sample the tail. The tail of the normal distribution can be quickly sampled using a previously described transformation of exponentially distributed PRNs.[3]

Sampling of the ziggurat overhangs (Steps 4–8 in Table 1) can be further accelerated. In general, sampling domains should closely approximate the shape of the function that they sample in order to maximize efficiency. Because the geometric properties of the overhangs and tail vary, these accelerations must be tailored for each particular $f(x)$. Nonetheless, a systematic approach is described below.

Overhangs are best sampled using right-triangular sampling domains, instead of rectangular boxes. Since each ziggurat layer is small, changes in $f'(x)$ between layers will often be small and the overhang will be nearly triangular (Figure 2). Using right-triangular sampling domains not only increases the likelihood of a successful rejection test, but also allows for more points to be preemptively accepted.

Figure 2 illustrates how triangular sampling from the overhangs of an exponential distribution minimizes the number of rejected points and enables preemptive acceptance of most of the remainder. Initially rectangular boxes in these overhangs are subdivided into three regions: (i) a triangular area exclusively above $f(x)$; (ii) a triangular area exclusively below $f(x)$, and (iii) a narrow band of area, proximal to the $f(x)$ curve that must still be sampled by rejection. This first triangular region of points above $f(x)$ does not need to be sampled, so any point drawn in this

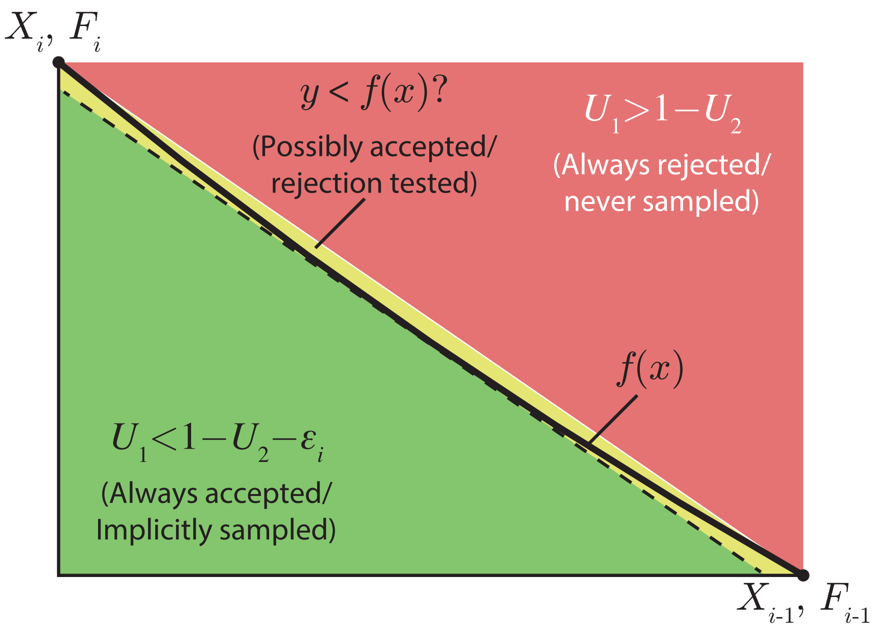
X_i, F_i
$y < f(x)$?
(Possibly accepted/ rejection tested)
$U_1 > 1 - U_2$
(Always rejected/ never sampled)
$f(x)$
$U_1 < 1 - U_2 - \varepsilon_i$
(Always accepted/ Implicitly sampled)
$X_{i\text{-}1}, F_{i\text{-}1}$

Figure 2. Accelerated rejection sampling of ziggurat overhangs. Consider the overhanging probability masses from Figure 1. Faithful sampling requires that random points within these overhanging boxes $\{x = X_i + U_1(X_{i-1} - X_i), y = F_{i-1} + U_2(F_i - F_{i-1})\}$, where $U_1, U_2 \sim U(0, 1)$, be sampled by rejection: x is returned if $y < f(x)$ and rejected if not. Knowledge of the shape of $f(x)$ – that it is convex over its whole support and has a slowly changing derivative – allow us to determine the outcome of this test without ever calculating $f(x)$ in many cases: sampling never succeeds when $U_1 > 1 - U_2$ and can be avoided, while sampling always succeeds when $U_1 > 1 - U_2 - \epsilon_i$ and can be implicitly accepted. Here, ϵ_i denotes the maximum deviation of $f(x)$ from a line segment connecting the vertices of the sampling box. Only a small narrow band proximal to $f(x)$, where $U_2 - U_1 < \epsilon_i$, requires a rejection test.

region can be reflected about the hypotenuse and used to uniformly sample regions (ii) and (iii). Points in region (ii) can then be accepted without calculating $f(x) = \mathrm{e}^{-x}$.

The curvature of $f(x)$ dictates how triangular boxes should be sampled. Line segments beginning and ending on convex functions will always lie above the function, while line segments on a concave functions will always lie below the function (by the definition of concavity). The exponential probability density function is convex over its whole support. Therefore, line segments $Y_i(x) = ((F_{i-1} - F_i)/(X_{i-1} - X_i))(x - X_i) + F_i$, defining the hypotenuse of each triangular sampling domain, lie completely above $f(x)$. Sampling above this line segment, i.e. the upper-right triangle, is unnecessary (Figure 2).

Overhangs in the normal distribution can be either concave or convex and must be sampled accordingly. Convex overhangs are sampled as described for the exponential distribution. Concave overhangs, near $x = 0$, must be sampling in both their upper-right and lower-left corners. It is nevertheless still useful to distinguish between these two right-triangular regions because points drawn within the lower-left corner of a concave overhang are guaranteed to lie below $f(x)$ and can be implicitly accepted. Thus, triangular sampling is still useful. By splitting the overhangs of the normal distribution into three distinctly sampled classes: concave overhangs, convex overhangs, and an overhang containing the inflection point, the algorithm can be accelerated somewhat; however, the additional complexity attenuates speedup (Section 4).

Additional points can be implicitly accepted and rejected in the overhangs by noting that the derivative $f'(x)$ generally changes very little within each overhang. The maximal deviation ϵ_i of $f(x)$ from $Y_i(x)$ is

$$\epsilon_i = \max_x |Y_i(x) - f(x)|,$$

$$\epsilon_i = \max_x \left| \frac{F_{i-1} - F_i}{X_{i-1} - X_i}(x - X_i) + F_i - f(x) \right|.$$

For $f(x) = \mathrm{e}^{-x}$, $\epsilon_i = F_{i-1} + (\mathrm{Log}(F_i - F_{i-1}) + X_i)(F_i - F_{i-1})$. Points lying below the line segment $Y_i(x) - \epsilon_i$ are guaranteed to lie beneath $f(x)$ and can be implicitly accepted (Figure 2). When $i_{\max} = 256$, the overhang that deviates most from $f(x)$, $\epsilon^{\max} = \max_i[\epsilon_i]$, is only 9% of the overhang's height. Thus, simply using $\epsilon^{\max}$ as a preemptive acceptance criterion for all overhangs works most efficiently (it is unnecessary to create a new lookup table ϵ_i). By partitioning the overhanging boxes into three regions, $> 91\%$ of all rejection tests are eliminated in the overhangs for the exponential distribution. Overall, our final implementation evaluates $f(x) = \mathrm{e}^{-x}$ in only 0.27% of samples.

This strategy, of exploiting the small curvature of $f(x)$, can be repeated for the normal distribution. Convex overhangs are, once again, treated as described in the exponential distribution, while concave overhangs must be treated differently. For concave overhangs, the maximum deviation of $f(x)$ from $Y_i(x)$ denotes a *minimal* criterion for acceptance of x. Points (x, y) residing above $Y_i(x) + \epsilon_i$ are implicitly *rejected*.

Sampling overhanging regions of $f(x)$ using triangular boxes is a generic approach that should accelerate sampling for most distributions with slowly changing derivatives. The approach adds to the complexity of the code and replaces calculation of $f(x)$ with other mathematical steps that might reduce speed for some PRNGs. Worse, convex and concave boxes must be handled separately, thereby diminishing speedup. Excessively searching for points that can avoid a rejection test offers diminishing returns because the dissected regions become increasing smaller. The ZA itself, a preemptive selection strategy, already accepts $> 97\%$ of all sampled points. Nevertheless, sampling overhangs with triangular boxes improved performance in the PRNGs developed here. Implementation of this technique reduced median runtimes for exponential and normal variate generation by an additional 25% and 7%, respectively (detailed below). Overall, triangular overhang sampling eliminates $\geq$ 50% of $f(x)$ function calls in the overhangs, while costing an additional subtraction and comparison operation.

3. Implementation

This new ZA was implemented accessibly and in three programing languages. It is implemented in C, seamlessly integrated with a modern uniform PRNG, and also embedded into Python and MATLAB/Octave in a manner that mimics native functions (see appendix for source code and details).

The modern uniform Mersenne Twister PRNG (mentioned above [11]) generates an array of uniform PRNs, which consume $\sim$ 4 KB of cache, but accelerate uniform PRN generation via SIMD instructions. Its code was modified slightly for seamless integration into the ziggurat PRNGs, such that (i) bounds checking and function overhead was minimized via macros, (ii) big endian support was deprecated (which is already incompatible with this and other ZAs [4]), and (iii) a seed is automatically generated from the system time, Process ID, and Parent Process ID. This uniform PRNG can be easily substituted in the provided source code.

Lookup tables were calculated to double precision without rounding errors. This was accomplished using a separate Python script (implemented to long double precision to avoid rounding errors) that calculates and saves tables to a header file. This approach simplified code and avoids pre-calculation overhead, which were already a minuscule 40 µms in the traditional algorithms. The pre-calculation script uses Brent's method to calculate X_i and the Alias method to setup rapid sampling of A, which requires a one-time cost of $\mathcal{O}(i_{\max} \cdot \text{Log}[i_{\max}])$ operations to construct.[10]

4. Performance

All PRNGs discussed here run in linear time and require very few ($\sim 10-100$) CPU clock cycles per PRN. Only programs that generate many PRNs ($> 10^4$) will observe noticeable differences in runtime. All C implementations of PRNGs tested here consume very little cache ($\sim$ 8KB) and no RAM; the Python and MATLAB functions allocate arrays of PRNs of user-defined size. Computation time per iteration varies for all Monte Carlo sampling techniques (such as ZAs) because of their stochastic nature. Nevertheless, average runtimes can be measured precisely.

The modified ZA executes faster than all other exponentially and normally distributed PRNGs for all architectures and compilers tested here. Relative Runtimes, i.e. the timing of the new algorithm divided by the timing of the fastest competing algorithm, were a median of 58% (Range 50–93%) for exponential variates and 53% (Range 41–63%) for normal variates (Figure 3). These fastest competing ziggurat implementations have been cited in the literature as the fastest exponential [3] and normal PRNGs available [4,7] and were further optimized in our comparison (by eliminating unnecessary assignment operations, avoiding integer overflows, etc). Thus, our new ZAs were compared against the very best alternatives available today (Section A.3).

The algorithm was profiled on five architectures (built circa 2013) and three compilers that yielded highly variable speedups (Table A1). The timing of each PRNG (on various architectures/compilers) denotes the time required to generate and aggregate 10^9 PRNs (median of five identical trails; code provided in the appendix). The clang compiler (Version 6) exhibited a wider range of performance times and was *often* slower than gcc (Table A1). In short, the modified

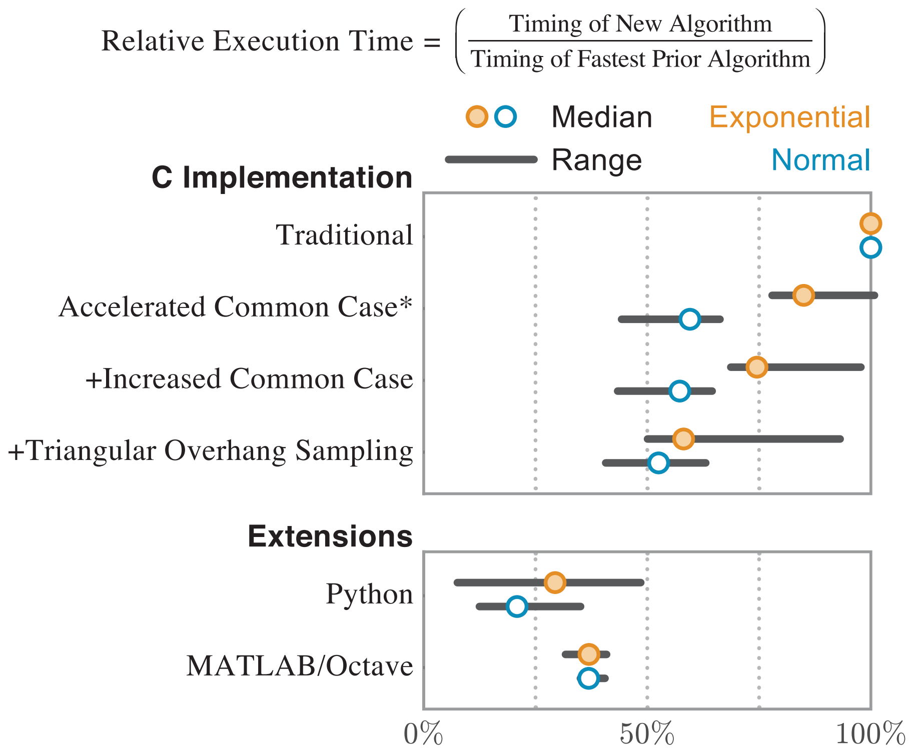

Figure 3. The modified ZA executes faster than all existing algorithms. The median relative runtime (circles) and range (black lines) of a benchmarking script that accumulates 10^9 PRNs was measured on seven different architecture/compiler combinations for the exponential and normal PRNG algorithms. 'O2' optimizations were enabled. *C Implementation* Each PRNG's median runtime was divided by the runtime of the fastest traditional ZA (see Figure A2 for benchmarking of various traditional ZAs). Hence, traditional algorithms have a 100% median runtime. The first improvement denotes the time saved by moving layers beneath $f(x)$, thereby eliminating two computational steps in the common case. This time is inferred (see appendix) from changes in runtime when the probability of early exit is intentionally altered. The second timing incorporates the increased probability of early exit that accompanies movement of the ziggurat layers beneath $f(x)$. In the final timing, the triangular overhang sampling domains are incorporated. *Extensions*. The new routine executes much faster than native functions in Python and MATLAB/Octave. This is, in part, because native functions use a different, slower uniform PRNG (see Figure A3 for uniform PRNG-independent profiling).

algorithm generates a uniform PRN, transforms this value into an exponential or normal variate, and accumulates the variate in $<$ 10 CPU cycles on average per iteration.

The new ZA differs from the old algorithm in three notable ways, which we profiled separately (Figure 3). The new algorithm (i) samples points faster in the common case, (ii) more often in the common case (Table 1), and (iii) more efficiently in the exceptional case when overhangs are sampled. Faster sampling in the common case accounts for the greatest share of speedup, but each improvement benefits runtime (see Table A1 for profiling details). While this first improvement eliminates only two operations (a variable assignment and table lookup), it still strongly reduces runtime because sampling now requires only six total steps. Improving overhang sampling particularly accelerated exponential generation, while improving common-case sampling particularly accelerated normal generation. This is because overhang sampling is more common in the exponential PRNG (for both the traditional and modified algorithms).

Native Python/Matlab functions utilize different uniform PRNGs than our new implementations and the competing C algorithms, which all used.[11] This increased variance in Relative Runtime across programming languages. Thus, subtracting the runtime spent generating uniform variates reduced the variance in execution time across programming environments (Figure A3).

5. Discussion

Here, we present a modified ZA that moves ziggurat layers beneath the probability density function, accelerating execution. This modification simplifies the calculation of exponentially and normally distributed PRNs in the common case. Sampling from the remaining probability mass overhangs using triangular sampling domains further accelerated computation. Speedups were obtained for all architectures and compilers tested.

The modified algorithm was implemented for the two most common non-uniform distributions and in common programming languages used by the scientific computing community. In principle, the algorithm could be extended to other variates, although these alternative distributions are often generated via transformations of exponential and normal variates (e.g. Gamma-distributed PRNs can be efficiently generated from normal PRNs [12]). The modified algorithm should accelerate sampling for any distribution where ziggurat layers cover the vast majority of the distribution. This is because the modified algorithm removes computational steps from the common case (without adding steps) and sampling from the common case dominates runtime.

Some distributions would be difficult to fill with ziggurat layers. For example, the ZA has generally only been applied to uni-modal distributions that are either monotonically decreasing or symmetric. Sampling from multi-modal distributions would require either additional creativity or the creation of ziggurat layers that do not begin at $x = 0$. In principle, however, these distributions could be creatively carved-up into rectangular and right-triangular domains that permit rapid Monte Carlo sampling. Finally, ZAs cannot sample the tail of distributions and rely upon alternative algorithms for these regions.

Triangular sampling domains should be useful for ziggurat overhangs of most probability distributions, although acceleration depends upon the shape of $f(x)$ – slowly changing derivatives and few inflection points are ideal. A slowly changing derivative of $f(x)$ implies that overhangs will be nearly triangular. Overall, the strategy trades fewer calls to $f(x)$ in exchange for a linear transformation of uniform variates. Because the transformation is not necessarily faster than computation of $f(x)$, our approach will not guarantee faster executing on all architectures or distributions. Nonetheless, all architectures/distributions profiled here were accelerated by this technique.

ZAs are the most efficient PRNGs today [7] in large part because they exploit advantages of modern architectures. ZAs use cached lookup tables and control flow operations that execute faster today than they would on simpler architectures. Computers lacking these strengths may benefit from alternate algorithms. ZAs in general should become more competitive as greater accuracy is desired. Implementing this algorithm to greater precision does not require augmenting the code in any way; only higher precision mathematical operations and lookup tables are required. In contrast, inverse transform sampling algorithms must calculate a transformation function that often requires a polynomial expansion.[8] For example, the natural log function, which inverts $f(x) = e^{-x}$, is calculated using a Taylor series expansion on IA-64 processors.[13] Higher precision calculations for this routine would require both additional Taylor-series terms and higher precision mathematics for each term. Hence, a ZA's speed should be most competitive for high-precision tasks. In general, the needs and computational resources of a program should be considered before choosing any PRNG.

Acknowledgments

I would like to thank Anton Goloborodko, Christine Miller, Geoffrey Fudenberg, Maxim Imakaev, and Nezar Abdennur for helpful discussions and comments.

Disclosure statement

No potential conflict of interest was reported by the author.

Funding

This work was supported by the National Cancer Institute under grant U54CA143874.

References

[1] L'Ecuyer P. Testing random number generators. In: Winter Simulation Conference. Piscataway, NJ: Institute of Electrical and Electronics Engine; 1992. p. 305–313.
[2] Marsaglia G, Tsang WW. A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions. SIAM J Sci Stat Comput. 1984;5(2):349–359.
[3] Marsaglia G, Tsang WW. The ziggurat method for generating random variables. J Stat Softw. 2000;5(8):1–7.
[4] Doornik JA. An improved ziggurat method to generate normal random samples. Oxford: University of Oxford; 2005.
[5] Zhang G, Leong PHW, Lee DU, Villasenor JD, Cheung RC, Luk W. Ziggurat-based hardware gaussian random number generator. In: International Conference on Field Programmable Logic and Applications. Tampere: IEEE; 2005. p. 275–280.
[6] Rubin H, Johnson BC. Efficient generation of exponential and normal deviates. J Stat Comput Simul. 2006;76(6):509–518.
[7] Thomas DB, Luk W, Leong PH, Villasenor JD. Gaussian random number generators. ACM Comput Surv. 2007;39(4):11–es. Available from: http://portal.acm.org/citation.cfm?doid = 1287620.1287622.
[8] Oved I. Computing transcendental functions. 2003 [cited 2014 Mar 24]; Available from: http://math.arizona.edu/∼aprl/teach/iriso/transcend.ps.
[9] de Schryver C, Schmidt D, Wehn N, Korn E, Marxen H, Korn R. A new hardware efficient inversion based random number generator for non-uniform distributions. In: 2010 International Conference on Reconfigurable Computing and FPGAs (ReConFig). Cancun, Mexico: IEEE; 2010. p. 190–195.
[10] Smith WD. How to sample from a probability distribution. 2002 Apr [cited 2014 Mar 24]; Available from: http://scorevoting.net/WarrenSmithPages/homepage/sampling.ps.
[11] Saito M, Matsumoto M. Simd-oriented fast mersenne twister: a 128-bit pseudorandom number generator. Monte carlo and quasi-monte carlo methods 2006. Berlin: Springer-Verlag; 2008. p. 607–622.
[12] Marsaglia G, Tsang WW. A simple method for generating gamma variables. ACM Trans Math Softw. 2000;26(3):363–372.
[13] Story S, Tak P, Tang P. New algorithms for improved transcendental functions on ia-64. IEEE Symposium on Computer Arithmetic. 1999;4.

Appendix

A.1. *Source code, installation, profiling script and usage*

See https://bitbucket.org/cdmcfarland/fast_prng. The Python package fast_prng is available for automatic installation via the Python Package Index at https://pypi.python.org/pypi/fast_prng.

A.2. *Demonstration of quality*

To affirm that the above implementation is mathematically correct, a statistical test 'quality_test.c' was created and provided. This script allows users to sample the raw moments of generated PRNs. The raw moments of a sample are always unbiased estimators of the raw moments of the generating distribution. Therefore, they provide a quick confirmation of the random properties of a distribution. Below is a sample output of the first five raw moments of 10^{12} trial PRNs:

```
Created 1000000000000 exponential distributed pseudo-random numbers...
X1: 1.000001 (Expected 1)
X2: 2.000003 (Expected 2)
X3: 6.000015 (Expected 6)
X4: 24.000099 (Expected 24)
```

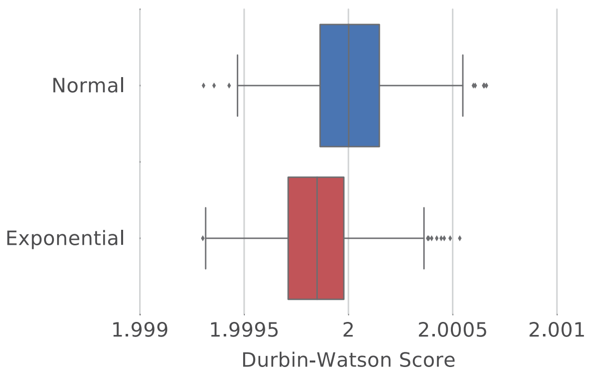

Figure A1. Distribution of 1000 Durbin–Watson Autocorrelation scores for the new PRNGs. The Durbin–Watson score is a measure of autocorrelation, where values of 2 denote no autocorrelation. Ideal PRNGs have no autocorrelation. Despite generating 10^{11} PRNs (in 10^3 sequences of 10^8 PRNs), no autocorrelation could be found.

```
X5: 120.000657 (Expected 120)
X6: 720.002455 (Expected 720)

Created 1000000000000 standard normal distributed pseudo-random numbers...
X1: -0.000000 (Expected 0)
X2: 1.000044 (Expected 1)
X3: 0.000003 (Expected 0)
X4: 3.000010 (Expected 3)
X5: 0.000045 (Expected 0)
X6: 15.000131 (Expected 15)
```

Deviation of these moments from expectation should scale as $1/\sqrt{N} = 10^{-6}$, i.e. to six significant digits. As this is the magnitude of observed deviations, the algorithm is as precise as can be reasonably measured.

Rounding errors were avoided by calculating values for the pre-computed lookup tables: X, A, and $f(X)$, to long double precision. Afterwards, these values are rounded to double precision.

Lastly, because this PRNG generates numbers deterministically from a uniform PRN generator, many random properties of the generator (e.g. periodicity) will be similar to the underlying uniform PRNG. The uniform generator used here is interchangeable and was previously demonstrated to exhibit excellent random properties.[11] Nevertheless, sequential randomness of the new PRNGs was affirmed by measuring 1000 Durbin–Watson Score for the autocorrelation of 10^8 consecutive PRN sequences (Figure A1). Despite sensitivities greater than one part in a thousand, no autocorrelation was detectable.

A.3. *Additional modifications to the old PRNG algorithms that mildly increased performance*

To ensure that the new algorithms were benchmarked against the best alternatives available, we found several slight modifications that accelerated traditional algorithms. For the fastest normal PRNG in the literature, these modifications were described previously.[4] The fastest alternative exponential PRNG was optimized iteratively, identifying three additional improvements that collectively reduced runtime by an average of 16% (Figure A2).

The benefit of these steps were profiled (Table A1) and compared to the original traditional algorithm. While each change improved performance on average, some exceptions exist. Two iterations of the traditional normal algorithm [4] were also created: a slower ZIGNOR version, and a heavily optimized VIZIGNOR version (presented in [4]). This optimized version was used for benchmarking and is provided with the source code. VIZIGNOR contains three optimizations: (a) $i_{\max} = 256$, (b) double-precision uniform PRNs (52-bit mantissa) were generated with only 32-bits of randomness, which (c) was accomplished using a $I(0, 2^{32})$ PRNG (rather than the $U(0, 1)$ PRNG), that was multiplied by X_i and k_i values rescaled accordingly, i.e. $X_i, k_i \rightarrow 2^{-32}X_i, 2^{-32}k_i$. This last technique was first described in [3].

A.4. *Additional modifications to the algorithm that mildly increased performance*

(1) Replacing the floating-point uniform PRNG for $U(0, 1)$ with a faster uniformly distributed 64-bit integer (either $I(0, 2^{63})$ for the exponential PRNG, or $I(-2^{63}, 2^{63})$ for the normal PRNG. This strategy, of using integers rather than floating-point numbers, has been used previously.[6] As in Figure A2, signed integer PRNs on the domain $[0, 2^{63})$

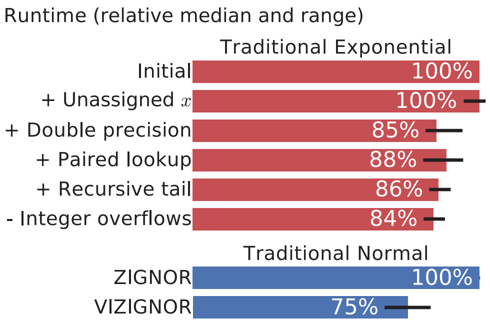

Figure A2. Improvements to traditional PRNGs for benchmarking. We identified five effective ways to accelerate the traditional exponential ZA [3]: (i) An unnecessary $x \leftarrow U_1 X_i$ assignment was eliminated, (ii) Floating-point math was executed at double-precision, (iii) k_i and X_i were paired into a single structure array to maximized memory prefetching, (iv) The tail of the exponential distribution was sampled by calling the exponential PRNG recursively,[6] (v) Integer overflows checks were avoided by using the signed $I(0, 2^{63})$ uniform integer PRNG. This signed PRNG was created by assigning the sign-bit of 64-bit inter representations to 0 (rather than calculating an absolute value). Although this inverts a negative integer's magnitude (in Two's Complement representations), the resulting distribution remains uniformly distributed.

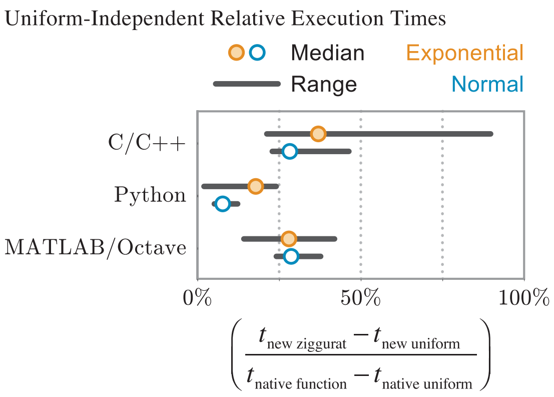

Figure A3. Uniform PRNG-Independent benchmarking of Python/Matlab extensions. We investigated the speedup that was attributable to the various ziggurat algorithms, independent of the underlying Uniform PRNG that fed each algorithm. This was accomplished by measuring the execution time required to generate and accumulate 10^9 Uniform PRNs (Table A1) and subtracting these times from the time required to generate and accumulate 10^9 exponentially or normally distributed PRNs. Variability in execution time between the various programming languages was reduced after this correction, as differences in the underlying uniform PRNGs exaggerated runtime differences in the ziggurat algorithms.

execute faster than unsigned integer PRNs because overflow checks are avoided. To keep output unaltered, X_i and F_i are rescaled accordingly.

(2) Sampling $I(0, 256)$ from the last 8 bits of the PRN returned by $I(0, 2^{63})$. This technique was also employed previously [3,4] and does not alter the output because the last 12 bits of $I(0, 2^{63})$ are squashed during floating-point multiplication.
(3) For normally distributed PRNs, Steps 4–8 (overhang sampling) were executed via a do-while loop.
(4) For exponentially distributed PRNs, Steps 4–8 were executed via a tail-recursive function.

Note that these modifications often swap floating point operations for integer operations and exploit compiler tendencies, so they may not increase performance for all architectures/compilers. These optimizations were all made before profiling and quality demonstrations.

A.5. *Modifications to the code that did not increase performance*

(1) Increasing $i_{\max}$ to 512 increased computational speed by only 4%, and was deemed unworthy of the greater cache consumption. Using an $i_{\max}$ that is not a power of two drastically slows computation because i cannot be sampled directly from a random bit sequence.

Table A1. New and traditional ZA runtimes vary across architectures, compilers, and programming language.

Architecture[a]	i5-4258U	i7-3770K	i5-4258U	E5620	E5620	i5-M540	i5-3470
Clock Speed (GHz)	2.40	3.50	2.40	2.40	2.40	2.53	3.20
Operating System	darwin	linux	darwin	darwin	darwin	windows	linux
Compiler[b]	clang 6	gcc 4.8	gcc 4.9	gcc 4.9	clang 6	MinGW	gcc 4.8
New exponential ZA							
All optimizations	7.06	2.79	4.32	4.46	8.84	6.29	3.03
Traditional overhangs	7.1	3.9	5.36	6.12	9.28	8.06	4.25
To infer early exit	7.46	3.11	4.77	5.0	9.28	7.16	3.39
New normal ZA							
All optimizations	3.98	3.29	5.05	5.54	5.47	5.23	3.58
Traditional overhangs	4.25	3.59	5.09	5.76	5.04	5.57	3.91
To infer early exit	4.5	3.64	5.54	6.08	6.08	5.34	3.96
Traditional exponential ZA [3]							
Initial algorithm	9.27	6.27	9.24	10.43	10.97	11.11	6.84
No x assignment	9.33	6.27	8.73	10.23	10.98	11.34	6.8
Double precision	8.71	5.31	7.9	8.86	9.44	9.02	5.74
Paired X_i, k_i lookup	8.74	5.3	8.41	9.45	9.7	8.92	5.74
Recursive tail	8.34	5.33	7.92	9.05	9.69	9.27	5.64
No integer overflows	8.15	5.26	7.44	8.91	9.49	9.22	5.71
Traditional normal ZA [4]							
ZIGNOR	11.3	7.55	11.26	13.34	13.53	17.14	8.23
VIZIGNOR (Accelerated)	8.61	6.26	8.0	8.92	9.74	12.84	6.82
For uniform-PRNG independent estimates							
Uniform and accumulate	2.53	2.12	2.49	3.15	3.13	2.96	2.31
New Python functions							
Exponential	18.8	3.45	18.76	13.21	13.42	–	4.29
Normal	17.27	4.92	18.08	12.62	12.6	–	5.71
Uniform	15.61	2.75	16.72	8.18	8.37	–	3.2
Native Python (Numpy) functions							
Exponential	38.74	45.82	41.05	45.25	45.38	–	49.99
Normal	52.04	39.54	51.53	60.44	60.34	–	43.26
Uniform	24.77	10.56	25.14	24.37	24.23	–	11.57
New MATLAB/Octave functions							
Exponential	9.79	3.93	9.82	–	–	–	4.18
Normal	7.3	5.39	7.31	–	–	–	5.6
Uniform	5.64	3.23	5.66	–	–	–	3.56
Native MATLAB R2013B functions							
Exponential	23.93	12.4	24.07	–	–	–	12.64
Normal	20.76	13.29	20.87	–	–	–	14.48
Uniform	14.04	7.56	14.02	–	–	–	8.24

[a] All architectures are Intel ® Core™ or Xeon ® products with ≥ 4 MB cache.
[b]Compiled with ‘O2’ optimizations enabled.

(2) Calculating a table of ϵ_i for every overhang (Figure 2). Instead, a single, maximal possible deviation $\epsilon^{\max} = \max_i[\epsilon_i]$ (and minimum deviation $\epsilon^{\min} = \min_i[\epsilon_i]$ for concave overhangs) was used. This avoids caching a fourth lookup table.
(3) Usage of most SIMD/ternary intrinsics, e.g. ‘fma’ found in ‘math.h’ in the C Standard Library. Modern compilers tend to leverage these instructions effectively without additional direction.
(4) Generating single-precision PRNs.

A.6. *Details of profiling*

Two profiling scripts provided in the source code along with accompanying output, titled ‘profile_all.py’ and ‘profile.c’, timed all routines. In every case, 10^9 PRNs were generated in five trials and the median trial runtime was saved to a comprehensive table (Table A1). This table was used to construct Figure 3 as described in the figure legend.

To infer the mean execution time $\bar{t}_{\text{Accelerated Common Case*}}$ of a modified ZA that moves layers beneath $f(x)$ without increasing the probability of early exit (presented in Figure 3), we assumed:

$$\bar{t} = P_{\text{early}-\text{exit}} t_{\text{early}-\text{exit}} + (1 - P_{\text{early}-\text{exit}}) t_{\text{overhang}},$$

where $\bar{t}$ is the average execution time, $P_{\text{early}-\text{exit}}$ is the probability that the algorithm exits at Step 2, and $t_{\text{early}-\text{exit}}$, t_{overhang} are the unknown execution times of early-exit and overhang PRN generation. The hybrid timing $\bar{t}_{\text{Accelerated Common Case*}} = P^{(\text{traditional})}_{\text{early}-\text{exit}} t^{(\text{modified})}_{\text{early}-\text{exit}} + (1 - P^{(\text{modified})}_{\text{early}-\text{exit}}) t^{(\text{modified})}_{\text{overhang}}$, where the *traditional* and *modified* superscripts denote values specific to the traditional and modified ZAs, was inferred. While all probabilities of early exit are known a priori, the two execution times (early-exit and overhang) must be inferred from observed profiling times. These two variables are under-determined when just one measurement of the overall execution time $\bar{t}$ is made, so an artificial modified algorithm, where $P_{\text{overhang}} \rightarrow 2P_{\text{overhang}}$ and $P_{\text{overhang}} = 1 - P_{\text{early}-\text{exit}}$, was profiled as well. These two measurements then allow us to solve for both $t_{\text{early}-\text{exit}}$ and t_{overhang}, and thus $\bar{t}_{\text{Accelerated Common Case*}}$, using a system of linear equations.

Python and Matlab extensions generally executed faster than the fastest C implementations. This is primarily because a particular emphasis was placed on finding the best C alternative, while the scripting language extensions were simply compared against native functionality – all prior ziggurat algorithms considered here were designed in C. However, the native Python and Matlab extensions also utilized different uniform PRNGs than our algorithms, which affected profiling. Thus, we calculated uniform-independent PRNG execution times for the new algorithm and native functions.