Transactions of the Korean Nuclear Society Spring Meeting
Jeju, Korea, May 12-13, 2016

Generating log-normally distributed random numbers by using the Ziggurat algorithm

Jongsoo Choi
Korea Institute of Nuclear Safety, 62 Gwahak-ro, Yuseong-gu, Daejeon 34142; k209cjs@kins.re.kr

1. Introduction

The quantification of a Probabilistic Safety Assessment(PSA) of a Nuclear Power Plant(NPP) always needs the uncertainty analysis(UA). Uncertainty analyses are usually based on the Monte Carlo method. Using an efficient random number generator(RNG) is a key element in success of Monte Carlo simulations. Log-normal distributed variates are very typical in NPP PSAs.

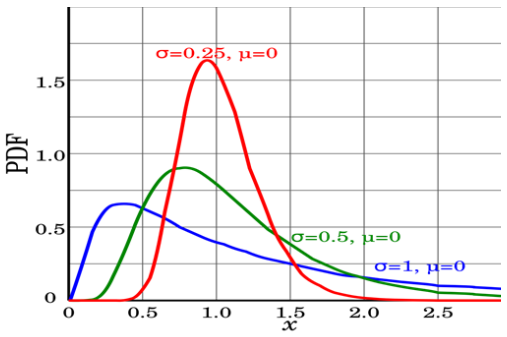
This paper proposes an approach to generate log-normally distributed variates based on the Ziggurat algorithm and evaluates the efficiency of the proposed Ziggurat RNG. The proposed RNG can be helpful to improve the uncertainty analysis of NPP PSAs.

2. RNGs for Monte Carlo Methods

2.1 Log-normal distribution

In probability theory, a log-normal distribution is a continuous probability distribution of a random variable whose logarithm is normally distributed. Thus, if the random variable X is log-normally distributed, then Y = ln(X) has a normal distribution. Likewise, if Y has a normal distribution, then x = exp(Y) has a log-normal distribution. A random variable which is log-normally distributed takes only positive real values. Its probability distribution function is:

$$\ln\mathcal{N}(x;\mu,\sigma) = \frac{1}{x\sigma\sqrt{2\pi}}\exp\left[-\frac{(\ln x-\mu)^2}{2\sigma^2}\right],\quad x>0$$



In NPP PSAs, log-normal distributions are typically specified using the mean of the distribution itself and a term called the ‘error factor’. The error factor for a log-normal distribution is defined as the ratio of the 95th percentile to the median, or, equivalently, the ratio of the median to the 5th percentile. The mathematical relationships between the mean(M) and error factor(EF), and the parameters of the underlying normal distribution (μ and σ) are shown by the following equations:

$$\sigma = \frac{\log(EF)}{1.64485363},\quad \mu = \log(M) - \frac{\sigma^2}{2}$$

In probability theory, the normal (or Gaussian) distribution is a very common continuous probability distribution. The probability density of the normal distribution is: $f(y) = \exp[-\frac{(y-\mu)^2}{2\sigma^2}] \Big/ \sigma\sqrt{2\pi}$.

Here, μ is the mean or expectation of the distribution (and also its median and mode). The parameter σ is its standard deviation with its variance then. If μ = 0 and σ = 1, the distribution is called the standard normal distribution or the unit normal distribution denoted by $N(0,1)$ and a random variable with that distribution is a standard normal deviate.

2.2 RNGs of N(0,1)

RNGs are very useful in developing Monte Carlo simulations. The generation of pseudo-random numbers is an important and common task in computer programming.

There are several methods to generate a random number based on a probability density function. These methods involve transforming a uniform random number in some way. Because of this, these methods work equally well in generating random numbers.

Probably the most important transformation functions for a normal pdf is known as the Box-Muller(1958) transformation[1]. It allows us to transform uniformly distributed random variables(U_1, U_2), to a new set of random variables with a normal distribution. The most basic form of the transformation looks like:

$$y_1 = \sqrt{-2\ln U_1}\cos(2\pi U_2),\ y_2 = \sqrt{-2\ln U_1}\sin(2\pi U_2)\cdot$$

It is known that this particular form of the transformation has two problems with it. 1) It is slow because of many calls to the math library. 2) It can have numerical stability problems when U_1 is very close to zero.

The polar form[2] of the Box-Muller transformation is both faster and more robust numerically. The algorithmic description of it is:

1. choosing random points (x, y) in the square $-1 < x < 1$, $-1 < y < 1$ until $s = x^2 + y^2 < 1$,
2. and then returning the required pair of normal random variables as

$$x\sqrt{-2\ln(s)/s},\ \ y\sqrt{-2\ln(s)/s}\ .$$

Wikipedia, the free encyclopedia, tells us that the polar method (attributed to George Marsaglia, 1964) is a pseudo-random number sampling method for generating a pair of independent standard normal random variables. While it is superior to the Box–

Muller transform, the Ziggurat algorithm is even more efficient.

2.3 Ziggurat algorithm

The Ziggurat algorithm[3] is a method for efficient random sampling from a probability distribution such as Normal distribution. The Ziggurat algorithm is a hybrid method that obtains its efficiency principally by using rejection sampling with some less efficient calculations performed for less commonly executed corner cases. The algorithm is used to generate values from a monotone decreasing probability distribution. It can also be applied to symmetric unimodal distributions, such as the normal distribution, by choosing a value from one half of the distribution and then randomly choosing which half the value is considered to have been drawn from. It was developed by George Marsaglia and others.

The naive rejection method has two main sources of inefficiency. 1) A large proportion of samples will be rejected. 2) We must evaluate $f(x)$ for each candidate point, which for many pdfs is computationally expensive.

The Ziggurat algorithm addresses these two issues by covering the pdf with a series of horizontal rectangles rather than a single square, and in an arrangement that attempts to cover the pdf as efficiently as possible, i.e with minimum area outside of the pdf curve. The following diagram (Fig. 1) demonstrates the approach. Note that we operate on one side of the pdf ($x \geq 0$), generating both positive and negative sample values requires that as a final step we randomly flip the sign of the generated non-negative values.

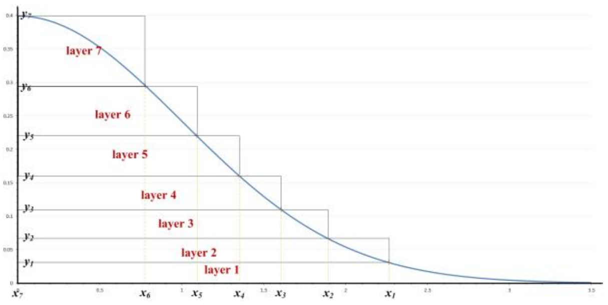


Fig. 1. Example ziggurat with 7 layers of $N(0,1)$

The Ziggurat algorithm gives good performance by using a very simple rejection sampling execution path for the majority of sample points generated, but with more expensive calculations performed to maintain mathematical exactness in some specific corner cases represented by the distribution tail and the far edges of the Ziggurat's rectangles.

The Ziggurat algorithm randomly generates a point in a distribution slightly larger than the desired distribution, then tests whether the generated point is inside the desired distribution. If not, it tries again. Given a random point underneath a probability density curve, its x coordinate is a random number with the desired distribution.

The distribution the Ziggurat algorithm chooses from is made up of n equal-area regions; $n-1$ rectangles that cover the bulk of the desired distribution, on top of a non-rectangular base that includes the tail of the distribution.

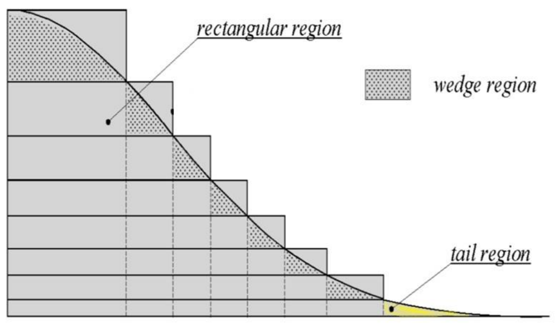


Fig. 2. Structure of a Ziggurat

Given a monotone decreasing probability density function $f(x)$, defined for all $x \geq 0$, the base of the Ziggurat is defined as all points inside the distribution and below $y_1 = f(x_1)$. This consists of a rectangular region from (0, 0) to (x_1, y_1), and the (typically infinite) tail of the distribution, where $x > x_1$ (and $y < y_1$). This layer (call it layer 1) has area A. On top of this, add a rectangular layer of width x_1 and height A/x_1, so it also has area A. The top of this layer is at height $y_2 = y_1 + A/x_1$, and intersects the density function at a point (x_2, y_2), where $y_2 = f(x_2)$. This layer includes every point in the density function between y_1 and y_2, but (unlike the base layer) also includes points such as (x_1, y_2) which are not in the desired distribution. Further layers are then stacked on top. To develop a Ziggurat of size n we choose x_1 such that $x_n = 0$, meaning that the top box, layer n, reaches the distribution's peak at (0, $f(0)$) exactly. Layer i extends vertically from y_{i-1} to y_i, and can be divided into two regions horizontally: the (generally larger) portion from 0 to x_i which is entirely contained within the desired distribution, and the (small) portion from x_i to x_{i-1}, which is only partially contained. Fig. 1 shows an example Ziggurat of size 7 ($n = 7$).

Ignoring for a moment the problem of layer 1, and given uniform random variables U_0 and $U_1 \in [0,1]$, the Ziggurat algorithm can be described as:

1. Choose a random layer $1 \leq i \leq n$.
2. Let $x = U_0 x_{i-1}$ and $x = U_0 A/y_1$ for $i = 1$.
3. If $x < x_i$, return x. (Rectangular region)
4. If $i = 1$, generate a point from the tail using the fallback algorithm. (Tail region)
5. Let $y = y_{i-1} + U_1 (y_i - y_{i-1})$.
6. Compute $f(x)$. If $y < f(x)$, return x. (Wedge region)
7. Otherwise, choose new random numbers and go back to step 1. (Rejected)

Whenever $x > x_1$, the Ziggurat algorithm requires a fallback. The fallback algorithm, of course, depends on the distribution. For a normal distribution, Marsaglia suggests a compact algorithm:

1. Let $x = -\ln(U_1)/x_1$.
2. Let $y = -\ln(U_2)$.
3. If $2y > x^2$, return $x + x_1$.
4. Otherwise, go back to step 1.

2.4 Comparison of Sampling Methods

In order to evaluate the efficiency of the Ziggurat RNG, four new versions of the Ziggurat methods are created. The first, labelled Ziggurat (64 layers), is a Ziggurat of size 64. The last, labelled Ziggurat (2000 layers), is a Ziggurat of size 2000.
To make timing comparisons more robust, I consider same computer platform(64-bit Windows 7 operating system with Intel Core i7-2600K CPU @ 3.4GHz , 16GB RAM) and compiler(Intel Fortran) and two reference method. The first is a sampling method based on Marsaglia polar method. The polar method produces a pair of independent normal variates by using 2 uniform random numbers(URN), but the sampling method provides 1 normal variate from 1 trial like a common approach. The other is a uniform RNG using RAN intrinsic function of Fortran compiler.

Table I compares the CPU time that is required to generate 1E10 random numbers. It is shown that a Ziggurat with large number of layers is faster than a Ziggurat with small number of layers.

Table I. Problem : generating 1E10 samples of N(0,1)

sampling method	CPU time(sec.)
Marsaglia polar	126.84
Ziggurat (64 layers)	65.28
Ziggurat (128 layers)	56.64
Ziggurat (256 layers)	51.91
Ziggurat (2000 layers)	45.40
URN(for reference)	34.32

Table II compares the average numbers of calculations for generating 1 normal sample for two Ziggurats. It shows the characteristics of the Ziggurat algorithm regarding number of layers.

The calculations of wedge and tail cases are more expensive than those of rectangular cases.

Table II. Average number of calculations for generating 1 sample

	Ziggurat (64 layers)	Ziggurat (2000 layers)
# URNs	1.0758	1.0034
# Rectangular cases	0.9881	0.9996
# Wedge cases	0.0504	0.0018
# Tail cases	0.0013	2.467E-5

3. Conclusions

This paper focuses on evaluating the efficiency of the Ziggurat algorithm from a NPP PSA point of view. From this study, we can draw the following conclusions.

- The Ziggurat algorithm is one of perfect random number generators to product normal distributed variates.
- The Ziggurat algorithm is computationally much faster than the most commonly used method, Marsaglia polar method.

REFERENCES

[1] G. Box and M. Muller, A Note on the Generation of Random Normal Deviates, The Annals of Mathematical Statistics, Vol. 29, No. 2, 610–611, 1958.
[2] G. Marsaglia and T. A. Bray, A convenient method for generating normal variables, SIAM Rev. 6, 260–264, 1964.
[3] G. Marsaglia and W. Tsang, The Ziggurat Method for Generating Random Variables, Journal of Statistical Software, Vol. 5, No. 8, 2000.