Ziggurat algorithm

From Wikipedia, the free encyclopedia

The **ziggurat algorithm** is an algorithm for pseudo-random number sampling. Belonging to the class of rejection sampling algorithms, it relies on an underlying source of uniformly-distributed random numbers, typically from a pseudo-random number generator, as well as precomputed tables. 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 in the 1960s.

A typical value produced by the algorithm only requires the generation of one random floating-point value and one random table index, followed by one table lookup, one multiply operation and one comparison. Sometimes (2.5% of the time, in the case of a normal or exponential distribution when using typical table sizes)[*citation needed*] more computations are required. Nevertheless, the algorithm is computationally much faster than the two most commonly used methods of generating normally distributed random numbers, the Marsaglia polar method and the Box–Muller transform, which require at least one logarithm and one square root calculation for each pair of generated values. However, since the ziggurat algorithm is more complex to implement it is best used when large quantities of random numbers are required.

The term *ziggurat algorithm* dates from Marsaglia's paper with Wai Wan Tsang in 2000; it is so named because it is conceptually based on covering the probability distribution with rectangular segments stacked in decreasing order of size, resulting in a figure that resembles a ziggurat.

Contents

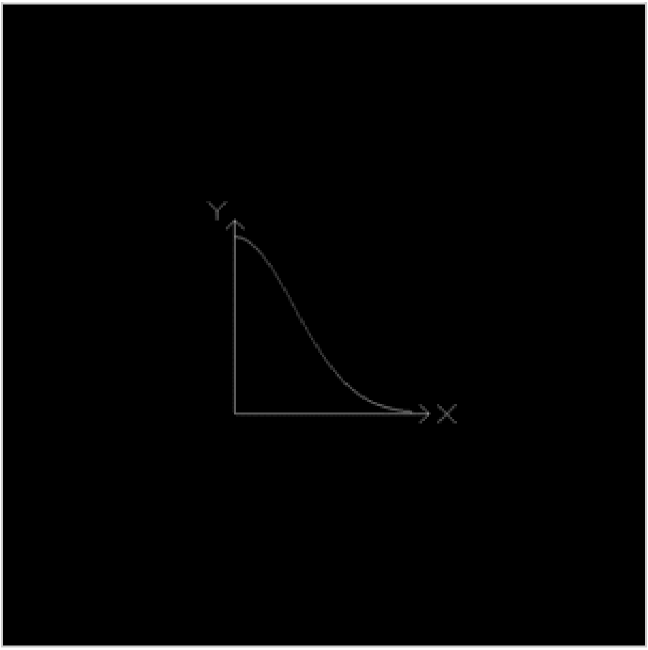
Theory of operation

The ziggurat algorithm is a rejection sampling algorithm; it 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.

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 0) 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.



The Ziggurat algorithm used to generate sample values with a normal distribution. (Only positive values are shown for simplicity.) The pink dots are initially uniform-distributed random numbers. The desired distribution function is first segmented into equal areas "A". One layer *i* is selected at random by the uniform source at the left. Then a random value from the top source is multiplied by the width of the chosen layer, and the result is *x* tested to see which region of the slice it falls into with 3 possible outcomes: 1) (left, solid black region) the sample clearly under the curve and is passed directly to output, 2) (right, vertically striped region) the sample value may lie under the curve, and must be tested further. In that case, a random *y* value within the chosen layer is generated and compared to *f(x)*. If less, the point is under the curve and the value *x* is output. If not, (the third case), the chosen point *x* is rejected and the algorithm is restarted from the beginning.

Further layers are then stacked on top. To use a precomputed table of size n ($n = 256$ is typical), one chooses x_1 such that $x_n = 0$, meaning that the top box, layer $n - 1$, reaches the distribution's peak at $(0, f(0))$ exactly.

Layer i extends vertically from y_i to y_{i+1}, and can be divided into two regions horizontally: the (generally larger) portion from 0 to x_{i+1} which is entirely contained within the desired distribution, and the (small) portion from x_{i+1} to x_i, which is only partially contained.

Ignoring for a moment the problem of layer 0, 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 $0 \leq i < n$.
2. Let $x = U_0 x_i$.
3. If $x < x_{i+1}$, return x.
4. Let $y = y_i + U_1(y_{i+1} - y_i)$.
5. Compute $f(x)$. If $y < f(x)$, return x.
6. Otherwise, choose new random numbers and go back to step 1.

Step 1 amounts to choosing a low-resolution y coordinate. Step 3 tests if the x coordinate is clearly within the desired density function without knowing more about the y coordinate. If it is not, step 4 chooses a high-resolution y coordinate, and step 5 does the rejection test.

With closely spaced layers, the algorithm terminates at step 3 a very large fraction of the time. Note that for the top layer $n - 1$, however, this test always fails, because $x_n = 0$.

Layer 0 can also be divided into a central region and an edge, but the edge is an infinite tail. To use the same algorithm to check if the point is in the central region, generate a fictitious $x_0 = A/y_1$. This will generate points with $x < x_1$ with the correct frequency, and in the rare case that layer 0 is selected and $x \geq x_1$, use a special fallback algorithm to select a point at random from the tail. Because the fallback algorithm is used less than one time in a thousand, speed is not essential.

Thus, the full ziggurat algorithm for one-sided distributions is:

1. Choose a random layer $0 \le i < n$.
2. Let $x = U_0 x_i$
3. If $x < x_{i+1}$, return x.
4. If $i = 0$, generate a point from the tail using the fallback algorithm.
5. Let $y = y_i + U_1(y_{i+1} - y_i)$.
6. Compute $f(x)$. If $y < f(x)$, return x.
7. Otherwise, choose new random numbers and go back to step 1.

For a two-sided distribution, of course, the result must be negated 50% of the time. This can often be done conveniently by choosing $U_0 \in (-1,1)$ and, in step 3, testing if $|x| < x_{i+1}$.

Fallback algorithms for the tail

Because the ziggurat algorithm only generates *most* outputs very rapidly, and requires a fallback algorithm whenever $x > x_1$, it is always more complex than a more direct implementation. The fallback algorithm, of course, depends on the distribution.

For an exponential distribution, the tail looks just like the body of the distribution. One way is to fall back to the most elementary algorithm $E = -\ln(U_1)$ and let $x = x_1 - \ln(U_1)$. Another is to call the ziggurat algorithm recursively and add x_1 to the result.

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.

Since $x_1 \approx 3.5$ for typical table sizes, the test in step 3 is almost always successful. Note also that $-\ln(U)$ is just a simple way to generate an exponentially distributed random number; if you have a ziggurat exponential distribution generator available, you can use it instead.

Optimizations

The algorithm can be performed efficiently with precomputed tables of x_i and $y_i = f(x_i)$, but there are some modifications to make it even faster:

- Nothing in the ziggurat algorithm depends on the probability distribution function being normalized (integral under the curve equal to 1), removing normalizing constants can speed up the computation of $f(x)$.
- Most uniform random number generators are based on integer random number generators which return an integer in the range $[0,\ 2^{32} - 1]$. A table of $2^{-32}x_i$ lets you use such numbers directly for U_0.
- When computing two-sided distributions using a two-sided U_0 as described earlier, the random integer can be interpreted as a signed number in the range $[-2^{31}, 2^{31} - 1]$, and a scale factor of 2^{-31} can be used.
- Rather than comparing $U_0 x_i$ to x_{i+1} in step 3, it is possible to precompute x_{i+1}/x_i and compare U_0 with that directly. If U_0 is an integer random number generator, these limits may be premultiplied by 2^{32} (or 2^{31}, as appropriate) so an integer comparison can be used.
- With the above two changes, the table of unmodified x_i values is no longer needed and may be deleted.

- When generating IEEE 754 single-precision floating point values, which only have a 24-bit mantissa (including the implicit leading 1), the least-significant bits of a 32-bit integer random number are not used. These bits may be used to select the layer number. (See the references below for a detailed discussion of this.)
- The first three steps may be put into an inline function, which can call an out-of-line implementation of the less frequently needed steps.

Generating the tables

It is possible to store the entire table precomputed, or just include the values n, y_1, A, and an implementation of $f^{-1}(x)$ in the source code, and compute the remaining values when initializing the random number generator.

As previously described, you can find $x_i = f^{-1}(y_i)$ and $y_{i+1} = y_i + A/x_i$. Repeat $n - 1$ times for the layers of the ziggurat. At the end, you should have $y_n = f(0)$. There will, of course, be some round-off error, but it is a useful sanity test to see that it is acceptably small.

When actually filling in the table values, just assume that $x_n = 0$ and $y_n = f(0)$, and accept the slight difference in layer $n - 1$'s area as rounding error.

Finding x_1 and A

Given an initial (guess at) x_1, you need a way to compute the area t of the tail for which $x > x_1$. For the exponential distribution, this is just e^{-x_1}, while for the normal distribution, assuming you are using the unnormalized $f(x) = e^{-x^2/2}$, this is $\sqrt{\pi/2}\operatorname{erfc}(x/\sqrt{2})$. For more awkward distributions, numerical integration may be required.

With this in hand, from x_1, you can find $y_1 = f(x_1)$, the area t in the tail, and the area of the base layer $A = x_1 y_1 + t$.

Then compute the series y_i and x_i as above. If $y_i > f(0)$ for any $i < n$, then the initial estimate x_1 was too low, leading to too large an area A. If $y_n < f(0)$, then the initial estimate x_1 was too high.

Given this, use a root-finding algorithm (such as the bisection method) to find the value x_1 which produces y_{n-1} as close to $f(0)$ as possible. Alternatively, look for the value which makes the area of the topmost layer, $x_{n-1}(f(0) - y_{n-1})$, as close to the desired value A as possible. This saves one evaluation of $f^{-1}(x)$ and is actually the condition of greatest interest.

References

- George Marsaglia; Wai Wan Tsang (2000). "The Ziggurat Method for Generating Random Variables" (http://www.jstatsoft.org/v05/i08/paper). *Journal of Statistical Software* **5** (8). Retrieved 2007-06-20. Note that this paper numbers the layers from 1 starting at the top, and makes layer 0 at the bottom a special case, while the explanation above numbers layers from 0 at the bottom.
- C implementation of the ziggurat method for the normal density function and the exponential density function (http://www.jstatsoft.org/v05/i08/supp/1), that is essentially a copy of the code in the paper.
- A java implementation (http://web173.cletus.kundenserver42.de/update/Ziggurat.java) of the ziggurat method which is a port of the c implementation.
- A C# implementation (http://heliosphan.org/zigguratalgorithm/zigguratalgorithm.html) of the ziggurat

algorithm and overview of the method.

- Jurgen A. Doornik (2005). *An Improved Ziggurat Method to Generate Normal Random Samples* (http://www.doornik.com/research/ziggurat.pdf). Nuffield College, Oxford. Retrieved 2007-06-20. Describes the hazards of using the least-significant bits of the integer random number generator to choose the layer number.
- Ziggurat algorithm generates normally distributed random numbers (http://www.mathworks.com/company/newsletters/news_notes/clevescorner/spring01_cleve.html) describing the ziggurat algorithm introduced in MATLAB version 5.
- David B. Thomas; Philip H.W. Leong; Wayne Luk; John D. Villasenor (October 2007). "Gaussian Random Number Generators" (http://www.cse.cuhk.edu.hk/~phwl/mt/public/archives/papers/grng_acmcs07.pdf). *ACM Computing Surveys* **39** (4): 11:1–38. doi:10.1145/1287620.1287622 (http://dx.doi.org/10.1145%2F1287620.1287622). ISSN 0360-0300 (//www.worldcat.org/issn/0360-0300). Retrieved 2009-07-27. "[W]hen maintaining extremely high statistical quality is the first priority, and subject to that constraint, speed is also desired, the Ziggurat method will often be the most appropriate choice." Comparison of several algorithms for generating Gaussian) random numbers.
- Boaz Nadler (2 March 2006). *Design Flaws in the Implementation of the Ziggurat and Monty Python methods (and some remarks on Matlab randn)* (pdf). The Journal of Business. arXiv:math/0603058v1 (//arxiv.org/abs/math/0603058v1). Illustrates problems with underlying uniform pseudo-random number generators and how those problems affect the ziggurat algorithm's output.

Retrieved from "http://en.wikipedia.org/w/index.php?title=Ziggurat_algorithm&oldid=593350211"

Categories: Pseudorandom number generators | Non-uniform random numbers | Statistical algorithms

- This page was last modified on 31 January 2014 at 22:43.
-