The Ziggurat Method for Random Number Generation

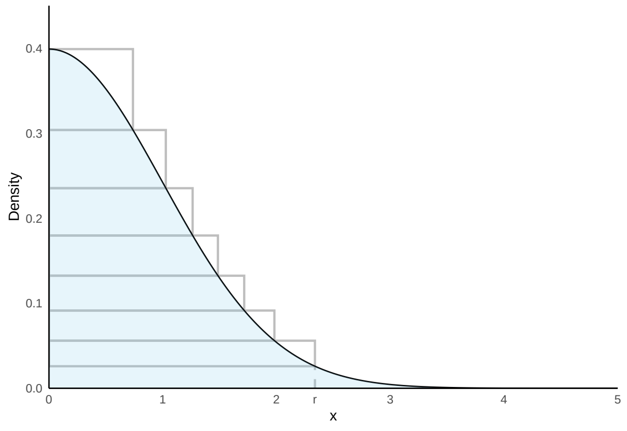
Ethan Alteza, Barron Bronson, Holden Ellis, Charlotte Huang

2026-02-12

https://docs.google.com/document/d/1HZTEfuxbhKEsMSGoFqDvulxrU5hy9rIVp31Wf9el38g/edit?usp=sharing

```r
source("Ziggurat.R")
source("Chisquare.R")
```

The Ziggurat Method for Generating Random Variables (2000) by Marsaglia and Tsang outlines a strategy for sampling from probability distributions that are either decreasing or symmetric and unimodal. The method itself was first published by the same authors in 1984, but was made more efficient with this publication. The namesake comes from the Ziggurat structures built in ancient Mesopotamia; these massive, rectangular pyramid structures were temple towers that featured several steps. The Ziggurat method functions by dividing the density into 256 regions of equal area (v). For all regions besides the base layer, the regions are in the shape of a rectangle, but the lowest region includes both a rectangular component and the tail beyond the cutoff, which is sampled using a specialized accept–reject method. These rectangles are mostly encapsulated under the curve, only a small portion of the right edge sticks out of the density.



Sampling is done by picking a random level of the ziggurat and then drawing from a uniform within it. The optimization here is that most of these points (over 99%) fall under the rectangle above and we are able to immediately accept them by a simple check, without ever computing the density function. This paper is quite significant to RNG, since the Ziggurat method became the standard technique in sampling for many distributions; it also is one of the fastest general methods. In NumPy, one of the most popular Python packages, generators for the Normal, Exponential, and Gamma distributions now utilize the Ziggurat method as of 2019. These generators are 2-10 times faster than the original implementation, which goes to show how efficient this method is.

Our Application

We implemented two Ziggurat R functions, one for the Normal distribution and one for the Exponential distribution: rnormzig and rexpzig. Separate functions need to be written for each distribution for two reasons. First, the most effective method for drawing from the tail distribution is unique. For the normal, Marsaglia recommends

$$X = -ln(U_1)/r, Y = -ln(U_2)U_1, U_2 \sim \text{Uniform}(0,1)\text{if } 2Y > X^2, Z_{tail} = r + X$$

and for the exponential he recommends

$$X = r - log(U), U \sim \text{Uniform}(0,1)$$

Second, the Normal distribution is symmetric about zero, whereas the Exponential distribution is not. After generating a positive Normal x value using the Ziggurat construction, symmetry is enforced by randomly assigning a sign to that value. This step is not necessary when sampling from the Exponential distribution because the distribution is defined only on $[0, infty)$. To draw from the rectangles, the code just requires the pdf and the inverse pdf for the distribution. This is done by first computing the breakpoints of each rectangle, which is handled by the function zigtable(). This is done by choosing a random value,r, in the domain such that

$$v = rf(r) + \int_r^{\text{inf}} f(x)dx$$

r becomes x_n and x_{n-1} is generated iteratively by

$$x_{i-1} = f^{-1}(\frac{v}{x_i} + f(x_i))$$

This process repeats until $x_0 = 0$. The challenge here is choosing the optimal v so that the iterative process doesn't overshoot or undershoot. This is done R's `uniroot` function Once the breakpoints are computed, it is easy to choose a random area and draw from the uniform within. First, to choose what rectangle we are sampling from, we randomly sample an integer between 2 and 257 (first index is the tail region) from the discrete uniform distribution using `sample()`. `sample()` is relatively slow compared with C implementations, where the index can be extracted cheaply from a single 32-bit random integer using the last 9 bits. After obtaining the rectangle we are sampling from, we check if the rectangle is the base layer. If the rectangle selected is the base layer then we randomly choose a x-value in the using `runif()` and check if $x_i < x_{i-1}$ or that the x-value is in the fast-accept region. If the x-value is not in the fast-accept region the code switches to the distribution-specific tail generator (defined for exponential and normal above) rather than using the rectangular accept/reject logic. The process is the same if the chosen rectangle is not the bottom layer, but if the x-value is not in the fast-accept region we generate an additional uniform height and accept the point (x, y) only if it lies under the pdf curve. In the end, our code can construct a Ziggurat for any distribution that meets the mathematical criteria. Here is an example of the code for the exponential distribution with 8 levels:

```r
zigtable(function(x) { exp(-x) }, function(y) { -log(y) },8)
```

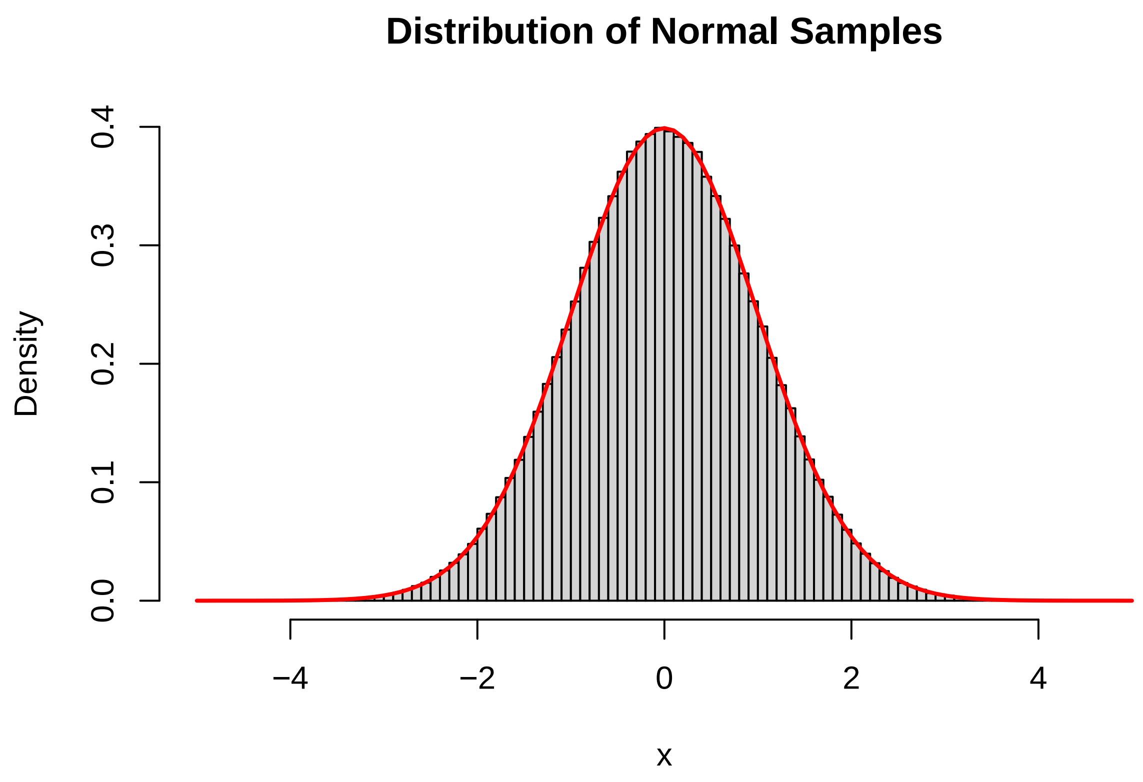
```
## $x
## [1] 0.0000000 0.4338930 0.7888599 1.1387148 1.5163812 1.9560291 2.5166676
## [8] 3.3489677
##
## $v
## [1] 0.1527383
```

Testing

After programming the Ziggurat method, we wanted to test it for the desired statistical properties, which can be done by generating a very large number of samples.

```r
set.seed(777)
x_1 <- rnormzig(1000000)
```

We can plot the distribution of the samples with the actual normal pdf overlayed, which shows good fit.



We can also check $\bar{x}$ and σ_x, which we expect to be very close to 0 and 1, respectively.

```r
mean(x_1)  # close to 0
```

```
## [1] -0.001120037
```

```r
sd(x_1)    # close to 1
```

```
## [1] 0.9997686
```

Testing For Accuracy

In order to formally validate that the Ziggurat method is actually generating the random variables of interest, we performed the chi-square goodness of fit test (Knuth 1997). This test is used to check the distribution of the random variables generated by quantizing the horizontal axis of the probability density function into k bins and deriving a single value as a quality metric from the determined actual and expected number of samples in each bin. We constructed an R function based on the chi-square statistic formula:

$$\chi^2_{k-1} = \sum_{i=1}^{k} \frac{(Y_i - tp_i)^2}{tp_i}$$

t = number of observations

$p_i = $probability that each observation falls into the the category i

Y_i = the number of observation that actually do fall into the category i

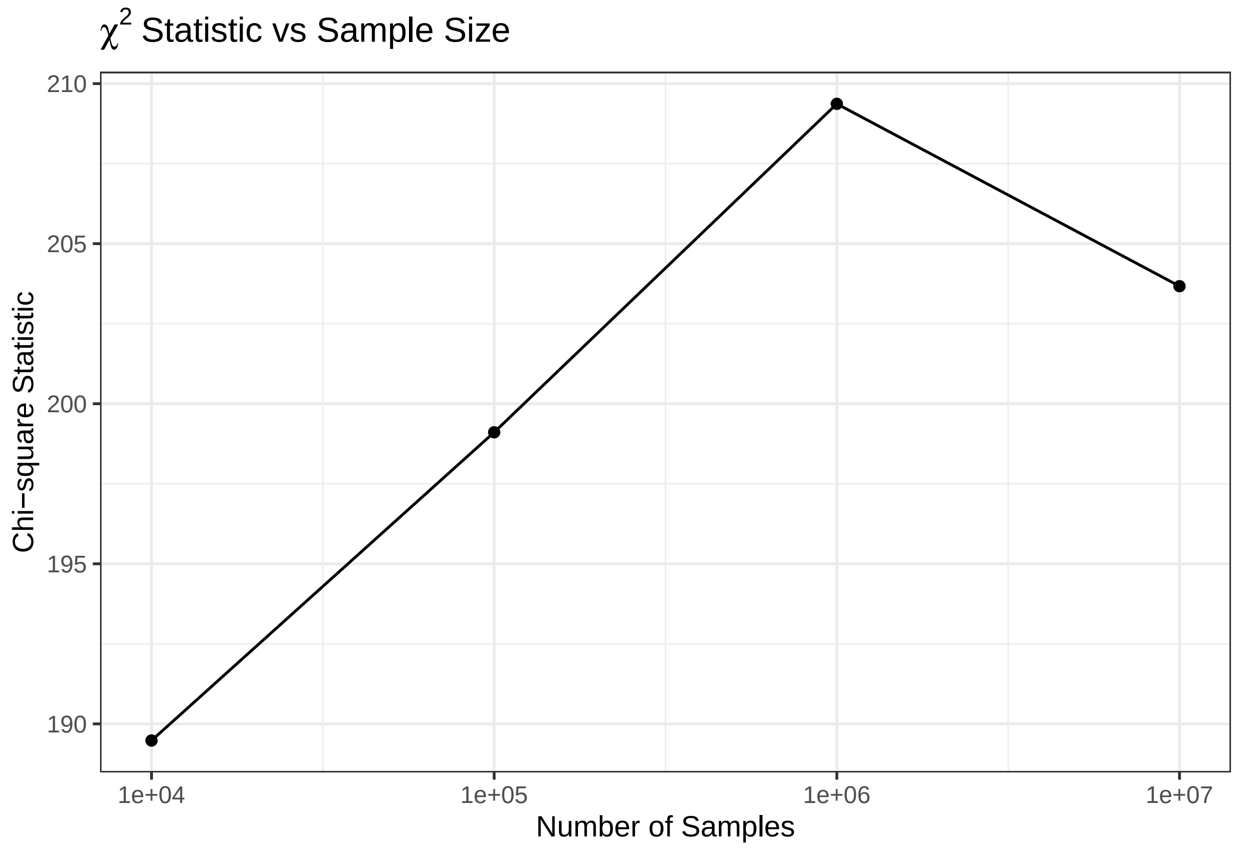
Using the rnormzig() function to generate 1,000,000 random variates, the distribution generated was tested based on 200 bins spaced uniformly over $[-7, 7]$ which we chose based on the paper that also analyzes the Ziggurat method (Leong et al. 2005). The calculated 2 value was 174.2026 and 184.4201 for the Ziggurat method and the rnorm() function in R, respectively. The critical value for a chi-square test with 199 degrees of freedom at 95% confidence level ($\alpha = 0.05$) is 232.912. With the rexpzig() function, 1,000,000 random variatates were generated and tested based on 200 bins now spaced uniformly over [0, 7]. The calculated chi-square values were 167.7520 and 199.7628 for the Ziggurat method and the rexp() function in R, respectively. Our implementation was also ran over increasing number of samples– one hundred thousand, a million, etc– which all had a chi-square value less than the critical value of 232.912. Overall, because all of the chi-square values were under the critical value we have successfully generated random variables that follow a normal and exponential distribution.

Testing for the Normal

```r
set.seed(777)

k <- 200
normal = rnorm(1000000) # for comparison
results <- chi_squared_test_norm(x_1, k)
results2 <- chi_squared_test_norm(normal, k)

results$Method <- "Ziggurat"
results2$Method <- "Rnorm"

combined <- rbind(results, results2)
combined <- combined[, c("Method", setdiff(names(combined), "Method"))]
df <- k - 1
combined$p_value <- pchisq(combined$chi_square, df = df, lower.tail = FALSE)
#Format for readability
combined$p_value_formatted <- format.pval(combined$p_value, digits = 3)
kable(combined, caption = "Chi-squared value: Ziggurat vs rnorm")
```

Table 1: Chi-squared value: Ziggurat vs rnorm

Method	chi_square	df	p_value	p_value_formatted
Ziggurat	220.8692	199	0.1375111	0.138
Rnorm	221.8924	199	0.1272802	0.127

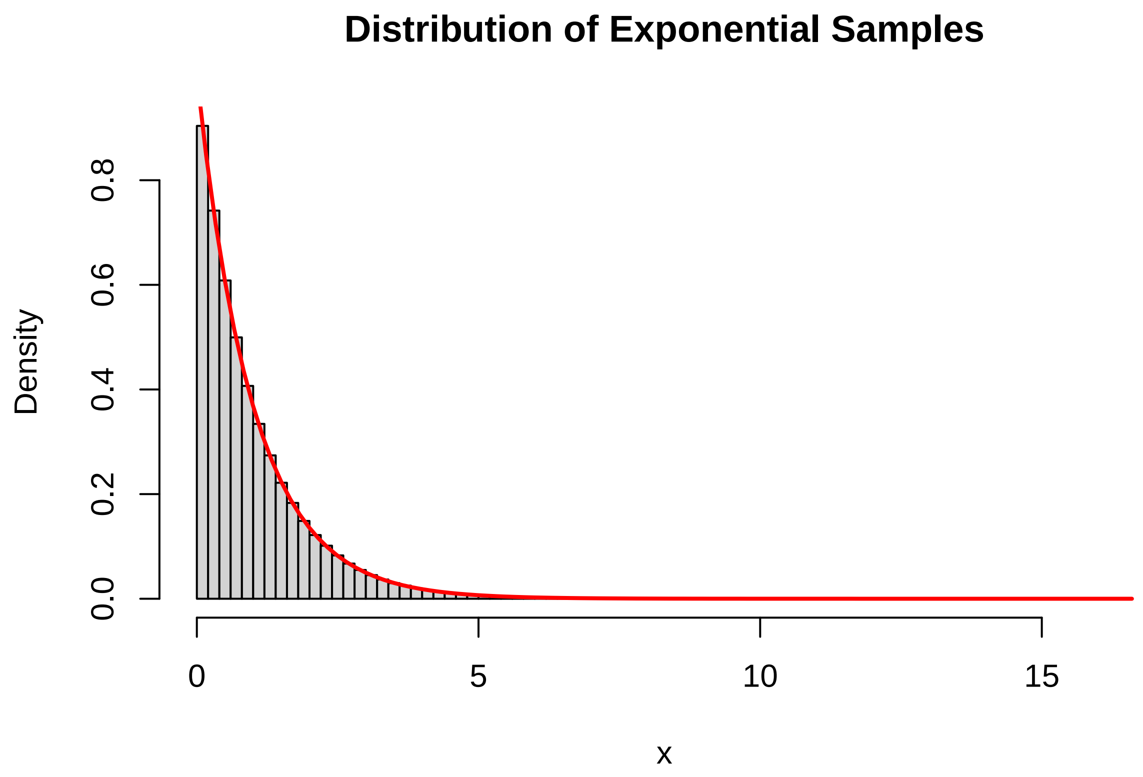


Testing for the Exponential

We can generate exponential samples using our Ziggurat function.

```r
set.seed(77)
x_2 <- rexpzig(1000000)
```

And again, look at the histogram of the samples in comparison to the real distribution function.



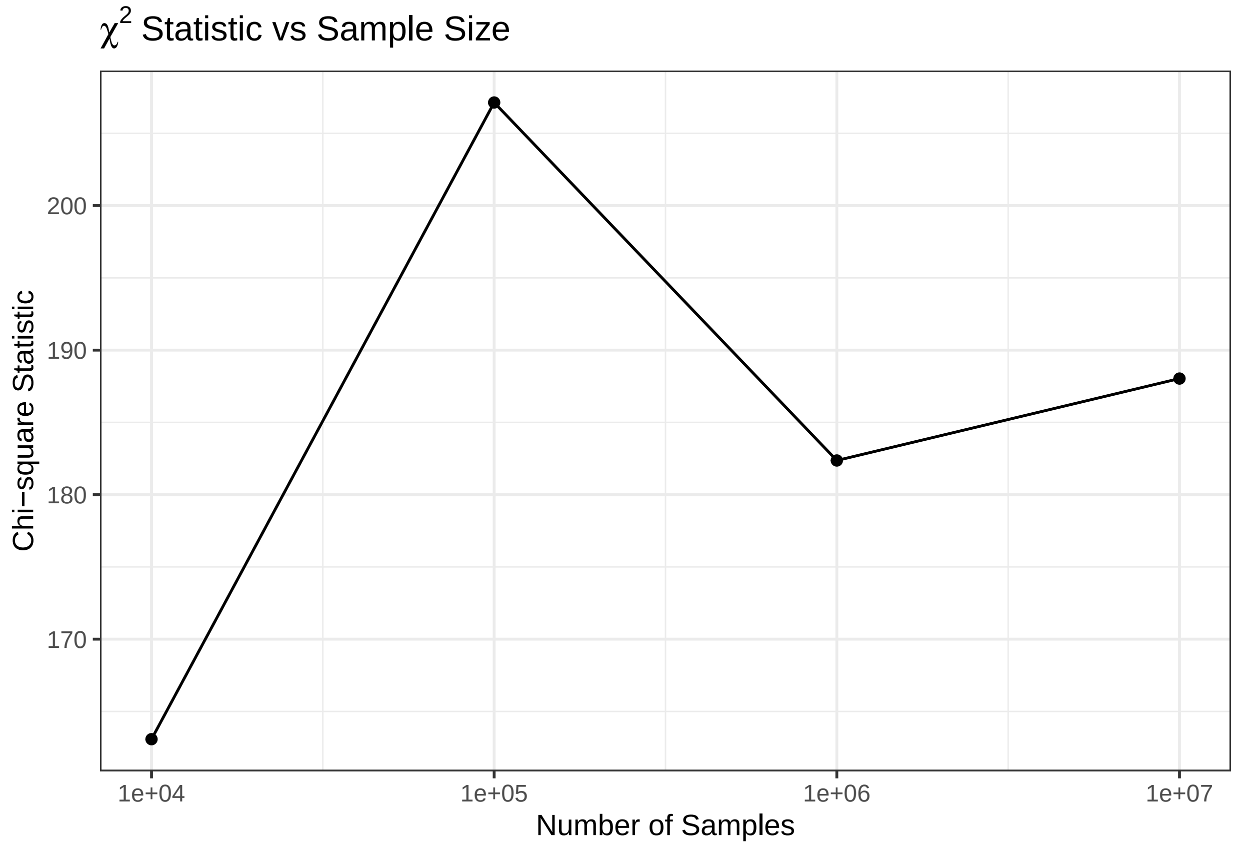
Repeating the steps from the previous section, we can write a function to divide up our distribution into bins, this time starting from 0 instead of -7.

```r
set.seed(777)

k <- 200
exp <- rexp(1000000)
results_exp <- chi_squared_test_exp(x_2, k)
results2_exp <- chi_squared_test_exp(exp, k)

results_exp$Method <- "Ziggurat"
results2_exp$Method <- "Rexp"

combined <- rbind(results_exp, results2_exp)
combined <- combined[, c("Method", setdiff(names(combined), "Method"))]
df <- k - 1
combined$p_value <- pchisq(combined$chi_square, df = df, lower.tail = FALSE)
#Format for readability
combined$p_value_formatted <- format.pval(combined$p_value, digits = 3)
kable(combined, caption = "Chi-squared value: Ziggurat vs rexp")
```

Table 2: Chi-squared value: Ziggurat vs rexp

Method	chi_square	df	p_value	p_value_formatted
Ziggurat	167.7520	199	0.9476905	0.948
Rexp	199.7628	199	0.4714596	0.471

And last, we can see how the chi-square statistic varies by sample size.



Challenges

One of the challenges faced working on this Midterm would be some coding obstacles. Although code was provided within the article, it was not as easy as copying and pasting the code onto R. Their implementation was in C and utilized a lot of low-level optimizations such as shift registers and bitwise operations that are very efficient but distract from the theory of the Ziggurat method. Additionally, C has base 0 indexing while R has base 1, so all the indexing had to be modified from what was described in the paper and implemented in their C code, which caused a lot of trouble. In the end, we did a mix of Marsaglia's simpler 1984 implementation along with some of the sampling optimizations from the 2000 paper so that the code is intuitive.

Unfortunately, the code runs very slowly because R is an incredibly slow programming language. The thing it is best at is handing off operations to C, which works well for repeated mathematical operations. The sampling optimization of the Ziggurat is that it does if-else comparisons instead of math, which works wonders in low-level CPU code but is actually slower in R. Ultimately, this project serves as a lesson of understanding your platform. The best sampling method is not universal and neither is any statistical method.

Works Cited

“What’s New or Different — NumPy V2.4 Manual.” Numpy.org, 2025, numpy.org/doc/stable/reference/random/new-or-different.html.

Leong, Philip H. W., et al. “A Comment on the Implementation of the Ziggurat Method”. Journal of Statistical Software, vol. 12, no. 7, Feb. 2005, pp. 1-4, doi:10.18637/jss.v012.i07.

Marsaglia, George, and Wai Wan Tsang. “The Ziggurat Method for Generating Random Variables”. Journal of Statistical Software, vol. 5, no. 8, Oct. 2000, pp. 1-7, doi:10.18637/jss.v005.i08.

Marsaglia, G., & Tsang, W. W. (1984). A Fast, Easily Implemented Method for Sampling from Decreasing or Symmetric Unimodal Density Functions. SIAM Journal on Scientific and Statistical Computing, 5(2), 349–359. https://doi.org/10.1137/0905026

McFarland C. D. (2016). A modified ziggurat algorithm for generating exponentially- and normally-distributed pseudorandom numbers. Journal of statistical computation and simulation, 86(7), 1281–1294. https://doi.org/10.1080/00949655.2015.1060234