The Ziggurat Algorithm and Implementation for Normal Distribution Sampling

Boxi Lin
Tianyuan Zhou

Abstract

The Ziggurat Algorithm is one of the most efficient rejection sampling methods that has been implemented as the built-in random number generator by many programming languages. It addresses the problem of high rejection rate and high calculation complicity for every candidate sample point. Ziggurat Algorithm divided the target area under the density function to several rectangulars and a tail area. Given number of these rectangulars be 255, only less than 2% of candidate random sample will be rejected. This continuously increases the efficiency "gap" between other algorithms and Ziggurat as the sample size increases. Apart from the good performance on generating huge data sets, Ziggurat also performs an architecture beauty.

In this Project, with the realization of the Ziggurat algorithm, we successfully generated 100,000 numbers from standard normal using 5 seconds. We justified that, theoretically, Ziggurat can efficiently generate a large set of data with low computational complicity. Initially, we intend to compare Ziggurat with other sampling methods, like Box-Muller. However, due to technical difficulties, the result don't reflect what we expected. A detailed discussion addresses this problem in the conclusion part. We also provide our own version of generating the initial table for Ziggurat to start in the appendix.

1. Introduction

Ziggurat Algorithm, developed by *Marsaglia and Tsang*[1] in the early 1980s, is a rejection sampling method for generating a random variable from a given monotone decreasing or symmetric unimodal distributions. Refined over the years, it is one of the most efficient algorithms for generating Normal variates and has been applied by many modern programming languages as built-in Normal number generator. In 2000, *Marsaglia and Tsang*[2] provided a faster and simpler version of Ziggurat method which will produce, for example, normal variates at the rate of 15 million per second with a C version on a 400MHz PC; *Leong et al.*(2005)[3] showed that the short period of the uniform random number generator of Ziggurat method could lead to poor distributions, and changing the uniform random number generator used in its implementation fixes this issue.

In this paper, the methodology and details of the Ziggurat Algorithm are given in Section 2, together with a specific operation for genearationg from standrad normal distribution.

Due to this algorithm's elegant mathematical form, it is not hard to be implemented in R. As a rejection sampling algorithm relying on a precomputed tables, however, Ziggurat Algorithm's major implementation challenge lies in setting up this table. Therefore, we also developed a numerical method to find the table before actually running Ziggurat Algorithm in Section 3. We also check Ziggurat Algorithm's efficiency and normality of generated samples in Section 3, followed by a summary discussion in Section 4.

2. Methodology

For the general Acceptance-Rejection method that we discussed on class, where the target probability density function $f(x)$ is covered by a candidate density $g(x) \cdot M$, there are 3 main sources of inefficiency:

1. High rejection rate;
2. For every candidate sample point x_i, we have to evaluate $f(x_i)$ which may be compuationally complicated and expensive.

Therefore, 3 basic criteia can be concluded for choosing a covering area C for $f(x_i)$ to enhance the effciency in Acceptance-Rejection sampling:

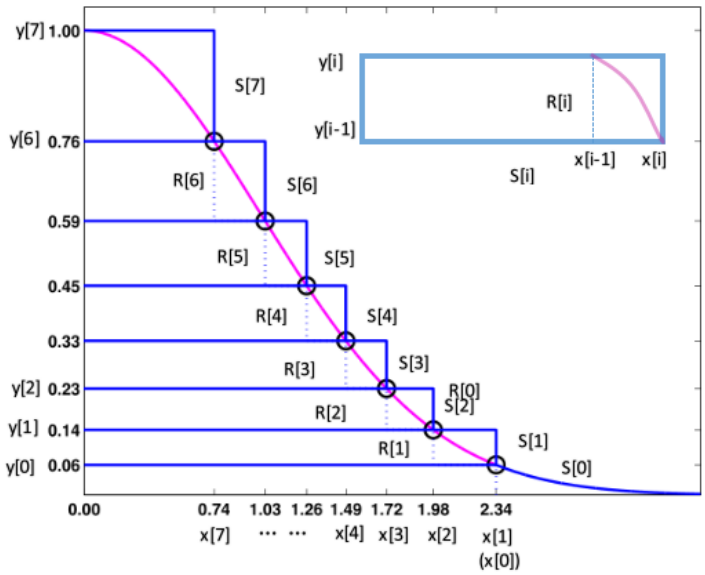
1. Reduce rejection rate by reducing the area in C that is not under $f(x)$;
2. Easy to generate sample point in C;
3. Easy and inexpensively to decide wether x_i is accepted or not, try to avoid calling $f(x)$ if it is expensive.

The **Ziggurat Method** is an algorithm well catering for these 3 criteria.

[Aside]: A ziggurat were massive structures built in ancient Mesopotamia and the western Iranian plateau. It had the form of a terraced step pyramid of successively receding stories or levels.[wiki] . Mathematically ziggurats resembles a two-dimensional step function. The Ziggurat Algorithm here is based on the one-dimensional version of the structure.[4]

source: http://i.imgur.com/eeYVO.jpg

The Ziggurat Algorithm is a highly efficient rejection sampling method based on covering the target probability density function $f(x)$ with a series horizontal rectangulars $S[1], \cdots, S[N-1]$, and a bottom tail section $S[0]$ which is the part of $f(x)$ its own underneath $S[1]$ (for simplicity of description, we still call $S[0]$ a (nonrepresentational) rectangular. Note here $f(x)$ is monotone decreasing; if we want to generate from symmetric unimodal distributions, such as the normal distribution, just simply sample a value from one half of the distribution and then randomly choosing which half the value is considered to have been drawn from.



source: https://www.mathworks.com/moler/chapters.html

Here we use $N = 8$ for the simplicity of graphing.

Notation:

- $S[i]$ is the ith covering rectangular, $i = 0, 1, \cdots, N$;
- $(x[i], y[i])$ is the coordination of the top right corner of $S[i]$, $i = 1, 2, \cdots, N$;
- $R[i]$ is the rectangular to the left of $x = x[i+1]$ in $S[i]$.
- F is the region under f;
- C is the union of $S[i]$, i.e. the whole ziggurat structure covering.

These N covering rectangulars are required to satisfy the following properties:

1. All N rectangulars have the same area. When N is given, the algorithm to precompute $A, x[i]$ and $y[i]$ is given in Appendix 1. We would pre-store the value of $x[i]$ and $y[i]$ in a table.
2. The bottom left corner of the each rectangular should be on the p.d.f, i.e. $f(x[i]) = y[i-1]$ to ensure that p.d.f in $[x_i + 1, x_i]$ is **entirely** covered by $S[i]$ as **efficiently** as possible.

As N increasing, the region of C/F squeezes, the rejection rate decrease correspondingly. In practice, the implementation of the algorithm usually use $N = 128$ *or* 256:

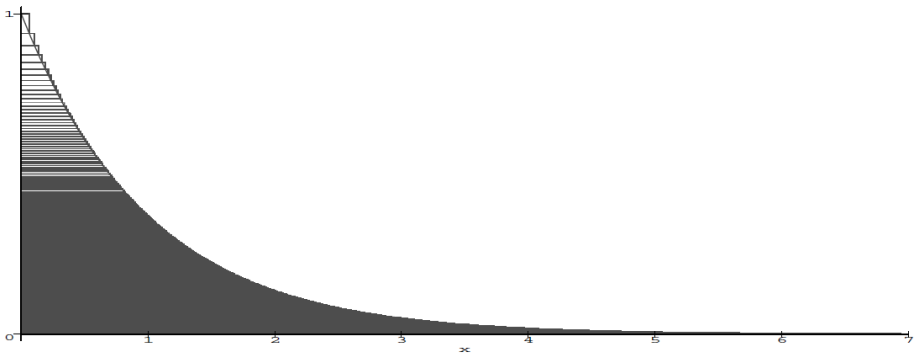


Figure 2: The ziggurat, showing every 10′th after the first 20 of 255 rectangles.

Source: https://core.ac.uk/download/pdf/6287927.pdf

To illustrate the logic behind the Ziggurat Algorithm, we perform an operation of generating stand Normal distribution:

1. We randomly select a rectangular to sample our point from. Based on the property that all covering rectangulars have the same area A, each has $1/N$ chance to be selected, so we generate random number $K_i = k$ in {0, 1, ... , N - 1} indicating that $S[k]$ is selected;
2. If $k > 0$, generate a random number $U_1 \sim U(0, 1)$;

- If $x_i = u_1 \cdot x[k-1] \leq x[k]$, then point we generate (x_i, y_i) must fall in R_k no matter with y_i; we accept x_i as our sample point immediately;
- If$x_i = u_1 \cdot x[k-1] > x[k]$, then point we generate (x_i, y_i) must be in S_k/R_k; in this case, we have to generate U_2 $U(0, 1)$;
  - If $y_i = u_2 \cdot (y[k] - y[k-1]) + y[k-1] < f(x_i)$, then (x_i, y_i) falls under the p.d.f in S_k/R_k, we accept x_i;
  - Otherwise, we reject this sample point and back to step I.

3. If $k = 0$, we have to generate frOm R_0. Sample a random number U_1 between 0 and $x[1] * y[0]$;

- If $x_i = \frac{u_1}{y[0]} \leq x[1]$, then we accept x_i immediately;
- Otherwise, we have to generate from tail $x \geq x[1]$; Marsaglia (1963) gave a simple algorithm.

4*. To generate from symmetric distributon, we firstly generate a sample x_i from the right side of the distribution; then further generate $z \in \{-1, 1\}$; then $z \cdot x_i$ is accepted in our sample.

The code that generates 10000 normal variates and a series of statistical tests are provided in Apendix II.

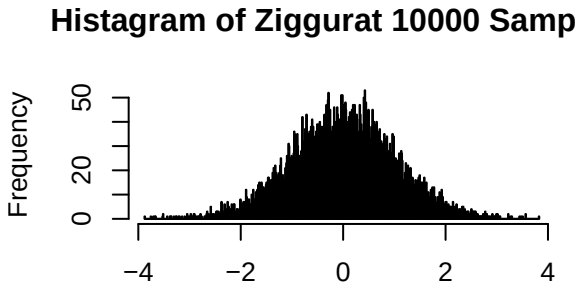
3. Implementation

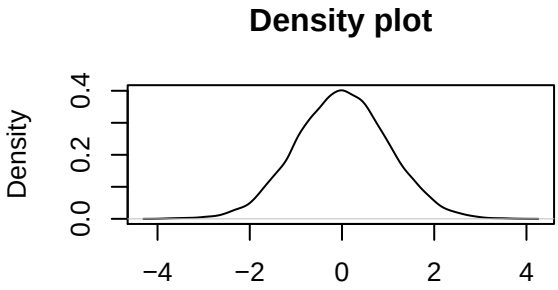
I. Efficiency

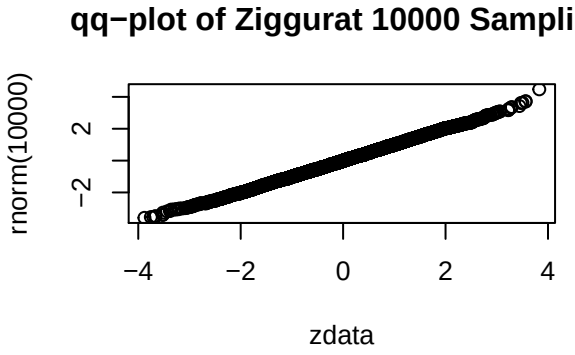
```
## [1] EFFICIENCY RATIO:
## [1] rejection rate: 0.0132
## [1] immediate accept rate: 0.9994
```

Ziggurat Algorithm performs well in terms of efficiency with a low rejection rate. With $N = 256$, more than 99.9% of times we do not need to call $f(x)$.

II. Normality







The density plot and qqplot indicate that the generated data are plausibly normal.

```
##
##  Lilliefors (Kolmogorov-Smirnov) normality test
##
## data:  zdata
## D = 0.0060672, p-value = 0.4989
```

Based on Lilliefors (Kolmogorov-Smirnov) normality test, there is no evidence against that the data is normally distributed.

III. Comparison with Box-Muller method

	Elapsed time(10 times averagely): s	
Sample Size	*Ziggurat*	*Box-Muller*
1000	0.09	0.11
10000	0.77	0.44
100000	5.17	3.16
1000000	61.53	32.37

Hardare: Intel(R) Core(TM)i5-2520M CPU@2.50GHz Installed RAM: 4.00GB

According to the comparison, our implementation of Ziggurat Algorithm does not beat the Box-Muler method.

4. Conclusion

The low rejection rate indicates that Ziggurat algorithm is significantly more efficient than any other Acceptance and Rejection sampling methods we learned on STAT341 lecture; this beautiful algorithm's high efficiency is achived due to the fact that when the precomputed table for $N = 256$ is given, more than 98% of sampling poins are required only the generation of one random floating-point value and one random table index, followed by one table lookup, one multiply operation and one comparison.

However, we have find some problems:

Theoretically, Ziggurat algorithm is supposed to work better than Box-Muller Method which require at least one logarithm and one square root calculation for each pair of generated values. In our implementation, however, the result is the reverse. Our analysis about the possible reasons for inefficiency are:

1. The implementation may be well optimized by better usage of local built-in function and if... else loop;
2. We applied a convienient but inefficient algorithm to generate from tail. The actual time required may be longer than our expectation.
3. The Ziggurat method may work better than Box-Muller method only with sample larger than a specific level which beyond our testing CPU condition.
4. A great portion of times witnessed our generated sample has a slightly lower number of points near $x = 0$, although it passed the normality test. This may due to the numerical precise limitation when setting up precomputed table.

we are eager to solve these problems in future research.

References

[1] Marsaglia, George and Tsang, Wai Wan (1984), A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions, SIAM Journ. Scient. and Statis. Computing, 5, 349-359.

[2] George Marsaglia; Wai Wan Tsang (2000). "The Ziggurat Method for Generating Random Variables". Journal of Statistical Software. 5 (8). Retrieved 2007-06-20.

[3] Philip H. W. Leong, Ganglie Zhang, Dong-U Lee, Wayne Luk, John Villasenor (2005). "A Comment on the Implementation of the Ziggurat Method". Journal of Statistical Software.10.18637/jss.v012.i07.

[4] Wikipedia, Ziggurat Algorithm, https://en.wikipedia.org/wiki/Ziggurat_algorithm

[5] Heliosphan, The Ziggurat Algorithm for Random Gaussian Sampling, http://heliosphan.org/zigguratalgorithm/zigguratalgorithm.html

Appendix I

Process to Finding the area A and x[0]~x[N-1], y[0]~y[N-1]}

Starting at * :We would first initialize a $x[0]$. The finding process start with the base layer $S[0]$, which contains a rectangular $R[0]$ and a tail for all $x > x[0]$. so $A = R[0]$ + tail. For the normal distribution, the tail area $t = P(x > x[0])$. This can be obtained from the table or integrate using formulas. (We need to find an algorithm to get the value of tail if the target distribution is awkward, typically using numerical integration.)

Also we have $y[0] = f(x[0])$ is the top right corner of the rectangular $R[0]$. then we can set $A = x[0]*y[0]+tail = S[0]$

Then, we can calculate $x[i], y[i]$ for $i = 1, 2, ., N-1$, recursively using STEP 1.

STEP 0:

$x[1] = x[0], y[1] = y[0] + \frac{A}{x[1]}$,

then we can have

$$\begin{aligned} S[1] &= x[1] * (y[1] - y[0]) \\ &= x[0] * (y[0] + \frac{A}{x[0]} - y[0]) \\ &= x[0] * \frac{A}{x[0]} \\ &= A \end{aligned}$$

We have the calculated pair $x[1], y[1]$

STEP 1:

If we already calculated $x[0]$ $x[i]$ and $y[0]$ $y[i], i >= 1$, then let

$x[i+1] = f^{-1}(y[i]), y[i+1] = y[i] + \frac{A}{x[0]}$,

then we can verify

$$\begin{aligned} S[i] &= x[i] * (y[i] - y[i-1]) \\ &= x[i] * (y[i] + \frac{A}{x[0]} - y[i]) \\ &= x[i] * \frac{A}{x[0]} \\ &= A \end{aligned}$$

We get the new pair of $x[i], y[i]$.

We will recursively do STEP 1 for $N-1$ times, until we find $y[N-1]$ OR if $y[i]$ is already larger than $f(0)$, that we cannot find $f^{-1}(y[i])$, then need to restart at * with a bigger $x[0]$

STEP 2:

Then we want to camper this with f(0). ideally we want to find $y[N-1] = f(0)$. If $y[N-1] == f(0)$, then STOP, and we find the target A and $x[0]$ $x[N-1], y[0]$ $y[N-1]$ Else, if $y[N-1] > f(0)$, it means that the initial guess $x[0]$ was too small, then we shall pick a bigger $x[0]$, and restart from . *Else, if* $y[N-1] < f(0)$, *it means that the initial guess* $x[0]$ *was too big, then we shall pick a smaller* $x[0]$, *and restart from .*

As for choosing the new $x[0]$, I set a cap and floor for the x[0] value at the beginning and use the average of cap and floor. Whenever larger $x[0]$ needed, I would update the floor with current; otherwise update the cap to get a smaller $x[0]$

This whole process will generate the $A, x[0]$ $x[N-1]$, and $y[0]$ $y[N-1]$ based on the input N (number of rectangles to cover and separate the whole area of target distribution). As N goes to infinity, the total area

of all rectangles will convergent to the area under pdf. The acceptance rate for Ziggurat Algorithm will raise, but calculating $A, x[0]$ $x[N-1]$, and $y[0]$ $y[N-1]$ can be expensive. However, since this calculating process is done before the sampling process, this won't affect the performance of Ziggurat Algorithm during runtime.

R Code for gennerating Table:

```
dnorminv<-function(y) sqrt(-2*log(sqrt(2*pi)*y))
findyN <- function (N,xinit) {
  x <- numeric (N)
  y <- numeric (N)

  flag = 0

  x[1] <- xinit
  tail <- 1-pnorm(x[1])
  y[1] <- dnorm(x[1])
  A <- x[1] * y[1] + tail

  x[2]= x[1]
  y[2] = y[1]+A/x[2]

  for (i in 2:N-1) {
    if ((y[i] > dnorm(0))) {
      flag = -1
      break
    }

    x[i+1] = dnorminv(y[i])
    y[i+1] = y[i] + A/x[i]

  }

  if (flag == -1) {flag = 100}
  else if(abs(y[N] - dnorm(0))<=1e-6) {
    flag =1   #FOUND
  } else if (y[N] > dnorm(0)) {
    flag = 100
  } else if (y[N] < dnorm(0)) {
    flag = -100
  }
  return (list (flag,A,x,y))
}
findAxy <- function(N) {
  x1 <- numeric(N)
  y1 <- numeric(N)
  flag1 = 0
  A1 = 0

  xcap = 5
  xflo = 0
  xcur = (xcap + xflo)/2
  flag1 = findyN(N,xcur) [[1]]
```

```
  while (flag1 != 1) {
    if (flag1 == 100) {
      xflo = xcur
    } else if (flag1 == -100)  {
      xcap = xcur
    }
    xcur = (xcap+xflo)/2
    flag1= findyN(N,xcur) [[1]]
  }

  A1= round(findyN(N,xcur) [[2]],digits = 6)
  x1 = round(findyN(N,xcur) [[3]],digits = 6)
  y1 = round(findyN(N,xcur) [[4]],digits = 6)
  list (A = A1 , x = x1 , y = y1)
}
```

Appendix II

Implementation of Ziggurat Algorithm and Statistical Tests

```
##I. Efficiency
dnorminv<-function(y) sqrt(-2*log(sqrt(2*pi)*y))
findyN <- function (N,xinit) {
  x <- numeric (N)
  y <- numeric (N)

  flag = 0

  x[1] <- xinit
  tail <- 1-pnorm(x[1])
  y[1] <- dnorm(x[1])
  A <- x[1] * y[1] + tail

  x[2]= x[1]
  y[2] = y[1]+A/x[2]

  for (i in 2:N-1) {
    if ((y[i] > dnorm(0))) {
      flag = -1
      break
    }

    x[i+1] = dnorminv(y[i])
    y[i+1] = y[i] + A/x[i]

  }
```

```
  if (flag == -1) {flag = 100}
  else if(abs(y[N] - dnorm(0))<=1e-6) {
    flag =1   #FOUND
  } else if (y[N] > dnorm(0)) {
    flag = 100
  } else if (y[N] < dnorm(0)) {
    flag = -100
  }
  return (list (flag,A,x,y))
}
findAxy <- function(N) {
  x1 <- numeric(N)
  y1 <- numeric(N)
  flag1 = 0
  A1 = 0

  xcap = 5
  xflo = 0
  xcur = (xcap + xflo)/2
  flag1 = findyN(N,xcur) [[1]]

  while (flag1 != 1) {
    if (flag1 == 100) {
      xflo = xcur
    } else if (flag1 == -100)  {
      xcap = xcur
    }
    xcur = (xcap+xflo)/2
    flag1= findyN(N,xcur) [[1]]
  }

  A1= round(findyN(N,xcur) [[2]],digits = 6)
  x1 = round(findyN(N,xcur) [[3]],digits = 6)
  y1 = round(findyN(N,xcur) [[4]],digits = 6)
  list (A = A1 , x = x1 , y = y1)
}

#ziggurat

#Genearating from standard normal
GenFromTail<-function(a) {
  u1<-runif(1)
  u2<-runif(1)
  while (u2 > a*sqrt((a^2-2*log(u1)))){
    u1<-runif(1)
    u2<-runif(1)
  }
  x = sqrt((a^2-2*log(u1)))
  return(x)
}

table <- findAxy(256)
xtable <-table$x
```

```
ytable <-table$y
Atable <-table$A

myziggurat <- function(total,N) {
  x <- xtable
  y <- ytable
  A <- Atable
  immediate = 0
  rejection = 0
  R0 <- x[1]*y[1]
  data <- numeric(total)
  signpn <- (2*rbinom(total,1,0.5)-1)
  i = 1
  while (i <= total) {
    #index for rectangular
    k <- sample(0:N-1,1)
    u1<-runif(1)
    if (k>0){
      #generating from rectangular S[k]
      xi<-u1*x[k]
      if(xi<(x[k+1])){
        #xi in R[k], accept
        data[i] = signpn[i]*xi
        immediate = immediate + 1
        i = i+1
      }else{
        #generating yi
        u2<-runif(1)
        yi <- y[k]+u2*(y[k+1]-y[k])
        if(yi <=  dnorm(xi)) {
          data[i] = signpn[i]*xi
          i = i + 1
        } else{
          rejection = rejection + 1
        }
      }
    }
    else{
      #generating from S0
      w<-runif(1,max = A)
      if (w<= R0) {
        data[i] = signpn[i]*w/y[1]
        immediate = immediate + 1
        i = i + 1
      } else {
        data[i] = signpn[i]*GenFromTail(x[1])
        i = i+1
      }
    }
  }
  print(noquote(paste("EFFICIENCY RATIO:")))
  print(noquote(paste("rejection rate:",rejection/total)))
  print(noquote(paste("immediate accept rate:",immediate/total)))
```

```
  return(data)
}

zdata <- myziggurat(10000,256)

##II. Normality

par(mfrow = c(2,2))
hist(zdata,breaks=1000, main = "Histagram of Ziggurat 10000 Sampling")
plot(density(zdata), main = "Density plot")
qqplot(zdata,rnorm(10000), main = "qq-plot of Ziggurat 10000 Sampling")

library(nortest)
lillie.test(zdata)
```