EEE 554 Random Signal Theory (Fall 2020)

Midterm Project: The Ziggurat Algorithm

Due: October 8, 2020

Have you ever wondered how Matlab generates Gaussian random numbers? The method it uses, called the Ziggurat algorithm, is the topic of this midterm project. Each of you will implement the Ziggurat algorithm for a different continuous distribution. (See the end of this document for your assigned distribution.)

First, read and understand this document explaining the algorithm. Then write code implementing it. (As usual, Matlab is recommended but not required.) In addition, the project requires you to do various analytical tasks to correctly set up the algorithm and analyze it. You should turn in a report which includes your well-documented code, as well as a write-up describing your implementation details, and the answers to specific questions asked below. Specific tasks that *must* be covered in your project report will be labeled by the bold word **Task**.

Your grade for the midterm project will be 50% based on the correctness of your algorithm implementation, and 50% based on the clarity of your project report. You may discuss the project in broad terms with other students, but you must work independently.

1 Ziggurat algorithm overview

The Ziggurat algorithm is a version of rejection sampling, optimized in a way so that some significant pre-computation is required, but once this pre-computation is done, generating each sample from the distribution is, on average, extremely fast. For example, Matlab can generate Gaussian random numbers almost as fast as uniform random numbers, even though the Gaussian distribution is much more complicated.

As described in Homework 4 problem 3, the idea of rejection sampling is to define a two-dimensional region $\mathcal{R}$ that contains the target PDF f_S. This idea can be generalized slightly by using a function $g(x)$ with is proportional to the target PDF $f_S(x)$; that is, $f_S(x) = cg(x)$ for some constant c. Then the region $\mathcal{R}$ should be such that it includes the function $g(x)$. (See Figure 1.) A random pair X, Y is generated uniformly in the region $\mathcal{R}$. If $Y < g(X)$, then the sample X is accepted. Otherwise the sample is rejected and another random pair is generated, repeating until a sample is accepted. To make rejection sampling efficient, two objectives must be satisfied:

1. The region $\mathcal{R}$ should be designed so that it is easy to generate random pairs X, Y uniformly.
2. The probability of accepting a given pair should be as large as possible. This requires that the region $\mathcal{R}$ does not include much area above $g(x)$.

The Ziggurat algorithm assumes that the function $g(x)$ is monotonically decreasing. The region $\mathcal{R}$ consists of n sub-regions stacked on top of each other: $n-1$ of these sub-regions are rectangles, and the nth is rectangle adjoined to the tail of the distribution. This idea is illustrated in Figure 2, where each of the n regions (in the figure $n = 4$) is shaded in a different color.

The rectangles are set up so that each of the n sub-regions (including the nth region, which is not a rectangle) have exactly the same area. Thus, a point X, Y can be uniformly chosen from the full region as follows: First, choose a sub-region from a discrete uniform distribution—i.e.,

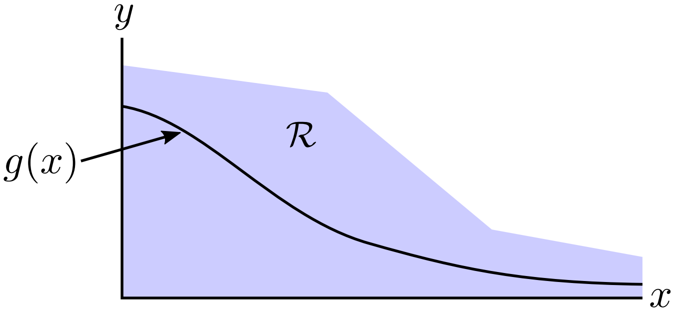

Figure 1: The basic rejection sampling concept.

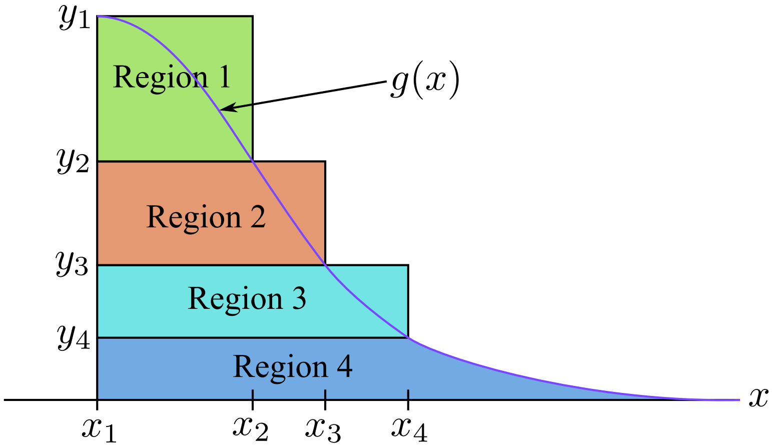

Figure 2: Diagram of the Ziggurat algorithm with $n = 4$. The name "Ziggurat algorithm" comes from the fact that this picture looks a little bit like a Ziggurat.

each with probability $1/n$—in Matlab this can be done with the function `randi`. Then, for the rectangular regions, X can be chosen from a uniform distribution along the length of the rectangle, and Y can be chosen from a uniform distribution along the height of the rectangle. The nth region requires a more complicated algorithm, but since this is only needed with probability $1/n$, this more complicated algorithm only needs to be run rarely. Thus, choosing a candidate X, Y is easy, which satisfies objective 1. If the number of regions n is large enough, then the stack of regions closely matches the function $g(x)$, which satisfies objective 2.

If we focus on a single one of the rectangular regions, we can see another feature that makes the Ziggurat algorithm so fast. Figure 3 shows the kth region, for some $k < n$. The kth region consists of a rectangle extending horizontally from x_1 to x_{k+1}, and vertically from y_{k+1} to y_k. As shown in Figure 2, $y_k = g(x_k)$ for each k. Since $g(x)$ is decreasing, $y_{k+1} < y_k$. We can easily choose a point uniformly from this rectangle by generating $X \sim \mathcal{U}(x_1, x_{k+1})$ and $Y \sim \mathcal{U}(y_{k+1}, y_k)$. After generating this point, we need to check whether $Y < g(X)$. However, if $X < x_k$, then $Y < g(X)$ no matter what Y is! This means that we do not even need to generate Y, nor evaluate the function $g(X)$, unless $X > x_k$. If n is chosen to be large enough, each rectangle will be much wider than it is tall, so the case where $X > x_k$ occurs only rarely.

To implement the Ziggurat algorithm, first the values $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$ must be pre-computed. This pre-computation requires some effort, but it only needs to be done once. After

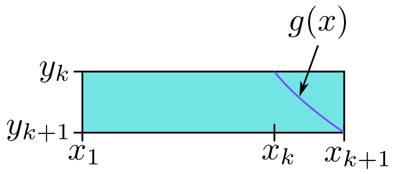

Figure 3: Diagram of the kth region of the Ziggurat algorithm.

these values are computed, the algorithm to generate one sample from the target distribution is summarized by the following pseudo-code.

```
Result: Returns one sample from distribution $f_S$
while true do
    Generate $K$ uniformly from $\{1, 2, \ldots, n\}$;
    if $K < n$ then
        Generate $X \sim \mathcal{U}(x_1, x_{K+1})$;
        if $X < x_K$ then
            return $X$
        else
            Generate $Y \sim \mathcal{U}(y_{K+1}, y_K)$;
            if $Y < g(X)$ then
                return $X$
            end
        end
    else
        Generate a point from the $n$th region;
    end
end
```

The above pseudo-code does not give details about how do deal with the nth region. We'll get back to that! Note that, most of the time, to generate a sample of the target distribution, all we need to do is generate one uniform discrete variable K, generate one continuous uniform random variable X, and do one comparison $X < x_k$. Even though the alternative paths—when $k = n$, or $X > x_k$, or $Y > g(X)$—are more complicated, these alternative paths are rare, so most of the time we do not need to execute them.

2 Implement a baseline sampling algorithm

Each of you is assigned a continuous distribution with a PDF of the form

$$f_S(x) = \begin{cases} cg(x), & x > x_1 \\ 0, & x < x_1 \end{cases}$$

where $g(x)$ is a monotonically decreasing function in the range $x > x_1$. See Section 6 for your distribution assignments.

Task 1 *Find the constant c so that your PDF is normalized correctly.*

Calculating c requires evaluating the integral

$$\int_{x_1}^{\infty} g(x)dx.$$

For some distributions, this integral is solvable in closed form. If it is not, you may use numerical integration software to calculate c numerically. (In Matlab, you should use the `integral` function.)

Task 2 *Write a function that computes the CDF of your distribution.*

As above, for some distributions the CDF will have a closed form expression, but for others it will require numerical integration.

Task 3 *Implement a baseline sampling algorithm via the transformation* $S = F_S^{-1}(U)$ *where* $U \sim \mathcal{U}(0, 1)$.

This baseline algorithm will be used to compare against the more efficient Ziggurat algorithm. To implement this baseline algorithm requires inverting the CDF. If you cannot find a closed-form expression for this inverse function, you may need to implement the bisection algorithm, which is summarized below. This algorithm can be used for any monotonic function $h(x)$. The idea is, in order to find a value x where $h(x) = u$, maintain values $x_{\min}$ and $x_{\max}$ where $h(x_{min}) < u$ and $h(x_{\max}) > u$. A point x is chosen at the midpoint between $x_{\min}$ and $x_{\max}$. Based on the value of $h(x)$, either $x_{\min}$ or $x_{\max}$ is updated to halve the difference between them. (This pseudocode assumes that h is increasing; the same algorithm can be used for monotonically decreasing functions with a slight variation.)

```
Input: $u$, $x_{\min}$, $x_{\max}$ where $h(x_{\min}) < u$ and $h(x_{\max}) > u$
Output: $x$ where $h(x) = u$
while $x_{\max} - x_{\min} > tol$ do
    $x \leftarrow \frac{x_{\min}+x_{\max}}{2}$;
    if $h(x) > u$ then
        $x_{\max} \leftarrow x$;
    else
        $x_{\min} \leftarrow x$;
    end
end
```

The tolerance parameter tol determines exactly how precise the result is. Setting tol $= 10^{-12}$ or smaller is often a good choice.

Task 4 *Generate at least 1000 samples from your baseline sampling algorithm. Keep track of the time it takes to run on your computer. (In Matlab, this can be done with the commands `tic` and `toc`.) Plot an estimated PDF from these samples, and make sure it matches the true PDF.*

3 Set up the Ziggurat

Before generating samples with the Ziggurat algorithm, the values $x_1, x_2, \ldots, x_n$ must be precomputed, as well as the corresponding values $y_1, y_2, \ldots, y_n$, where $y_k = g(x_k)$ for each k. These

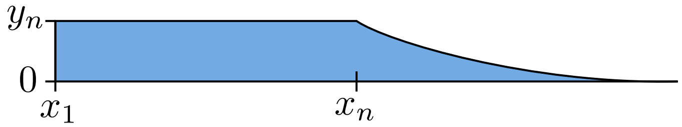

Figure 4: Diagram of the nth region of the Ziggurat algorithm.

numbers should be selected so that each of the n regions have exactly the same area. For $k < n$, the kth region is a rectangle, so its area is

$$(x_{k+1} - x_1)(y_k - y_{k+1}).$$

For the nth region, the area is

$$(x_n - x_1)y_n + \int_{x_n}^{\infty} g(x)dx.$$

To set up the Ziggurat, first choose a guess A for the area of each region. Based on this A, find x_2 so that the 1st region has area A. This can be done using the bisection algorithm. Then find x_3 so that the 2nd region has area A. Continue computing $x_4, \dots, x_n$. Given the value that you get for x_n, compute the resulting area of the nth region. If this area is less than A, then the original guess for A must have been too large; if greater than A, then the original choice of A must have been too small. Again, the bisection algorithm can be used to find the exact value of A.

Task 5 *Calculate $x_1, x_2, \dots, x_n$ and $y_1, y_2, \dots, y_n$ for three values of n: $4, 32, 256$. Check to make sure $x_1 < x_2 < \cdots < x_n$. Be sure to include the code that you used. Also list the x_k and y_k numbers for the $n = 4$ case. (Your report does not need to list the numbers for $n = 32, 256$.)*

4 Sampling from the nth region

Since the nth region of the Ziggurat is not a rectangle, it requires a different algorithm to sample from. However, since this region only comes up with probability $1/n$, the algorithm does not need to be especially efficient. Figure 4 shows the nth region. It can be separated into two parts: (i) a rectangle extending horizontally from x_1 to x_n and vertically from 0 to y_n, and (ii) an infinitely long tail extending from x_n to ∞, and upper bounded by $g(x)$.

Task 6 *If the point X, Y is chosen uniformly from this region, compute the probability p that it falls into the rectangular part of the nth region.*

Remember that we do not really need Y, we only need X. When X, Y is in the rectangular part, then X is simply uniform from x_1 to x_n. Thus, we can do the following: with probability p, generate $X \sim \mathcal{U}(x_1, x_n)$. Otherwise, generate X from the tail distribution.

Generating X from the tail distribution is the last piece of the Ziggurat algorithm. This requires generating a random variable from the PDF

$$f_T(x) = \begin{cases} c'g(x), & x > x_n \\ 0, & x < x_n \end{cases}$$

where the constant c' is chosen appropriately.

Task 7 *Find and implement a method to sample from the tail distribution f_T. One of two methods can be used, depending on your distribution:*

1. *If $g(x)$ has a closed-form integral, then a version of the inverse-CDF transformation can be used, as in your baseline sampling algorithm.*
2. *If $g(x)$ does not have a closed-form integral, you can use another rejection sampling algorithm. Here you should use the method of Homework 4 problem 3: choose a distribution f_R defined on $x > x_n$ that is easy to sample from, and a constant M such that $Mf_R(x) > g(x)$ for all $x > x_n$. For the distribution f_R, you may wish to use the exponential distribution restricted to $x > x_n$; another option is the Pareto distribution, given by PDF*

$$f_R(x) = \begin{cases} \frac{\alpha x_n^\alpha}{x^{\alpha+1}}, & x > x_n \\ 0, & x < 0 \end{cases}$$

where $\alpha > 0$ is a parameter. The exact choice of distribution f_R is up to you, but be sure to explain your choice. To implement this rejection sampling algorithm, generate $X \sim f_R$, and $Y \sim \mathcal{U}(0, Mf_R(X))$. Accept if $Y < g(X)$, and repeat if not. Note that this rejection sampling loop should be done inside a single iteration of the outer rejection sampling loop.

5 Implement and analyze the Ziggurat algorithm

Task 8 *Complete the implementation of the Ziggurat algorithm for $n = 4$, $n = 32$, and $n = 256$.*

Task 9 *For each of the three n values, run your Ziggurat algorithm to generate at least 1,000,000 samples. Make sure you compute the $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ constants only once. For each of the n values, use these samples to plot an estimated PDF, and compare against the true PDF. Carefully check that these match for your $n = 4$ algorithm, to make sure that your tail algorithm is working correctly.*

Task 10 *As you generate samples from the three variants of your algorithm, keep track of the following data:*

1. *How long it takes to run. Compare the time per sample with your baseline algorithm.*[1] *Make sure you run your code on the same computer, so that the comparison in meaningful.*
2. *How often each of the following possible outcomes occurs in the rejection loop:*
   - *(a) X is accepted because $X < x_k$,*
   - *(b) $X > x_k$ but X is accepted because $Y < g(X)$,*
   - *(c) $Y > g(X)$ so X is rejected,*
   - *(d) $k = n$ and a sample is drawn from the rectangular part of the nth region,*
   - *(e) $k = n$ and the tail algorithm is run.*

Task 11 *Can you think of any ways to further improve the performance of your algorithm?*

[1] Some of you may find that your baseline algorithm actually runs faster than the Ziggurat algorithm. If this is happens even for $n = 256$, be sure to check your Ziggurat implementation to make sure it is working correctly. However, even if everything is working right, the baseline algorithm may still run faster. This could happen for two reasons. First, for some distributions with closed-form CDFs, computing the inverse CDF F_S^{-1} can be quite fast. Second, since Matlab is an interpreted language, it sometimes runs quite slowly compared to the same algorithm implemented in a language like C. For example, Matlab is notoriously slow when it comes to loops. Since the goal of this project is for you to understand the algorithm rather than to build a commercial product, this is nothing to worry about.

6 Distribution assignments

The following table lists the distribution assignments for each student. Recall that the distribution you are simulating is

$$f_S(x) = \begin{cases} cg(x), & x > x_1 \\ 0, & x < x_1 \end{cases}$$

where c is a constant that you must determine. The table lists the function $g(x)$ and the boundary point x_1.

Student	Distribution name	$g(x)$	x_1
Arkan Abuyazid	Half-Student's t	$(1+4x^2)^{-5/8}$	0
Curtis Anderson	Half-Cauchy	$\frac{1}{1+x^2}$	0
Cooper Bertke	Half-generalized Gaussian	e^{-x^3}	0
Shao-Chun Chao	Restricted Frechet	$x^{-3}e^{-x^{-2}}$	$\sqrt{\frac{2}{3}}$
Dhaval Dalal	Inverse Gaussian	$x^{-3/2}\exp\left(-\frac{25}{2x}(\frac{x}{40}-1)^2\right)$	8
Richard Gerbino	Restricted Erlang	xe^{-2x}	0.5
Mihir Kotak	Half-generalized Gaussian	e^{-x^4}	0
Vasundhara Venkata Krishna	Burr	$\frac{1}{(1+x)^3}$	0
Juntong Liu	Half-Student's t	$(1+\frac{x^2}{2})^{-3/2}$	0
Siyang Liu	Restricted Gumbel	$e^{-x-e^{-x}}$	0
Shruthi Nagamalla	Exponential-logarithmic	$\frac{e^{-x}}{1-0.3e^{-x}}$	0
Amruthavarshini Vasundhara Narayanan	Half-q-Gaussian	$(1+\frac{3}{2}x^2)^{-2/3}$	0
Adeoluwa Ogunmefun	Gompertz	e^{x-e^x}	0
Sanggu Park	Restricted Frechet	$x^{-4}e^{-x^{-3}}$	$(\frac{3}{4})^{1/3}$
Yuye Ran	Restricted inverse-gamma	$x^{-2}e^{-1/x}$	0.5
Samuel Roark	Restricted Erlang	x^3e^{-x}	3
Yongho Seo	Restricted Rayleigh	$xe^{-x^2/2}$	1
Huiliang Shao	Restricted Chi-squared	$x^2e^{-x/2}$	4
Sang-Hun Sim	Half-Logistic	$\frac{e^{-x}}{(1+e^{-x})^2}$	0
Jingbo Sun	Restricted Burr	$\frac{x}{(1+x^2)^2}$	$\sqrt{1/3}$
Naga Sai Aishwarya Tallapragada	Restricted Weibull	$x^3e^{-x^4/4}$	$3^{1/4}$
Zhengjie Tang	Inverse Gaussian	$x^{-3/2}\exp\left(-\frac{(x-1)^2}{x}\right)$	0.5
Victor Isaac Torres Muro	Restricted Weibull	$x^2e^{-x^3/3}$	$2^{1/3}$
Vishnu Vaidya	Restricted Chi-squared	$x^{1/2}e^{-x/2}$	1
Brent Wallace	Restricted skew-normal	$\Phi(-2x)e^{-x^2/2}$	-0.5
Note: Φ is the standard Gaussian CDF, which can be computed in Matlab using the `normcdf` function			
Donald Wilson	Restricted Log-normal	$\frac{1}{x}e^{-(\ln x)^2/2}$	$1/e$
Yunlei Zhao	Restricted Levy	$x^{-3/2}e^{-1/x}$	$2/3$
Wenhui Zhu	Half-q-Gaussian	$(1+\frac{x^2}{2})^{-2}$	0
Jonathan Zilberman	Gompertz	e^{x-2e^x}	0