POLYTECHNIC UNIVERSITY
Department of Computer and Information Science

NORMAL DISTRIBUTION

K. Ming Leung

Abstract: The conventional polar (box-Muller) algorithm as well as the latest version of the Ziggurat algorithm for generating normally distributed random numbers are discussed.

Directory

Last Revision Date: March 3, 2004 Version 2.0

Table of Contents

1. Introduction

The normal distribution has a probability density given by

$$f(x) = c\, e^{-x^2/2},$$

with normalization factor c. Normalization requires that

$$\int_{-\infty}^{\infty} dx f(x) = c \int_{-\infty}^{\infty} dx\, e^{-x^2/2} = 1.$$

Therefore

$$c = I^{-1},$$

where I is the integral

$$I = \int_{-\infty}^{\infty} dx\, e^{-x^2/2}.$$

The amazing thing about this integral is that it is easier to compute I^2 than I itself. The integral for I^2 is a two-dimensional one, which because of the rotational symmetry should be performed in polar co-

ordinates. The calculation gives:

$$I^2 = \int_{-\infty}^{\infty} dx\, e^{-x^2/2} \int_{-\infty}^{\infty} dy\, e^{-y^2/2} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} dx\, dy\; e^{-(x^2+y^2)/2}$$
$$= \int_0^{\infty} dr\, r \int_0^{2\pi} d\theta\, e^{-r^2/2} = 2\pi \int_0^{\infty} dr\, r\; e^{-r^2/2}.$$

Changing the integration variable to $u = r^2/2$ then gives

$$I^2 = 2\pi \int_0^{\infty} du\; e^{-u} = 2\pi\; e^{-u}|_{\infty}^{0} = 2\pi.$$

Therefore $I = \sqrt{2\pi}$ and $c = 1/\sqrt{2\pi}$. The normalized form of the probability density function is then given by

$$f(x) = \frac{1}{\sqrt{2\pi}}\, e^{-x^2/2}.$$

2. The Polar (box-Muller) Algorithm

We discuss next methods for generating random numbers that are normally distributed. One can in fact use the inverse function method

for this distribution. The indefinite integral of $f(x)$ involves the error function and therefore the normally distributed random number are given in terms of the inverse error function. It turns out that faster methods are available.

One of those methods is the polar (box-Muller) method[1, 2, 3] which generates two values at a time. It involves finding a random point in the unit circle by generating uniformly distributed points in the [-1, 1] × [-1, 1] square and rejecting any outside of the circle. For each point accepted, a polar transformation produces two independent normally distributed numbers.

The following C/C++ code is an implementation of the box-Muller algorithm. Here we assume that `u1` and `u2` are two independent uniformly distributed numbers.

```
do { x1 = 2.0*double(rand())/RAND_MAX - 1;
     x2 = 2.0*double(rand())/RAND_MAX - 1;
     w = x1*x1 + x2*x2;
   } while (w >= 1);
w = sqrt((-2.0*log(w))/w);
```

```
y1 = x1*w;
y2 = x2*w;
```

This algorithm does not involve any approximations, so it has the proper behavior even in the tail of the distribution. However it is moderately expensive since the efficiency of the rejection method is

$$e = \frac{\pi}{4} \approx 0.785,$$

so about 21% of the uniformly distributed points within the square are discarded. The square root and the logarithm also contribute significantly to the computational cost.

3. The Ziggurat Method

The ziggurat method has been developed and refined over the years by Marsaglia and co-workers. An early version of it appeared in one of Knuth's *The Art of Computer Programming* book.[4]

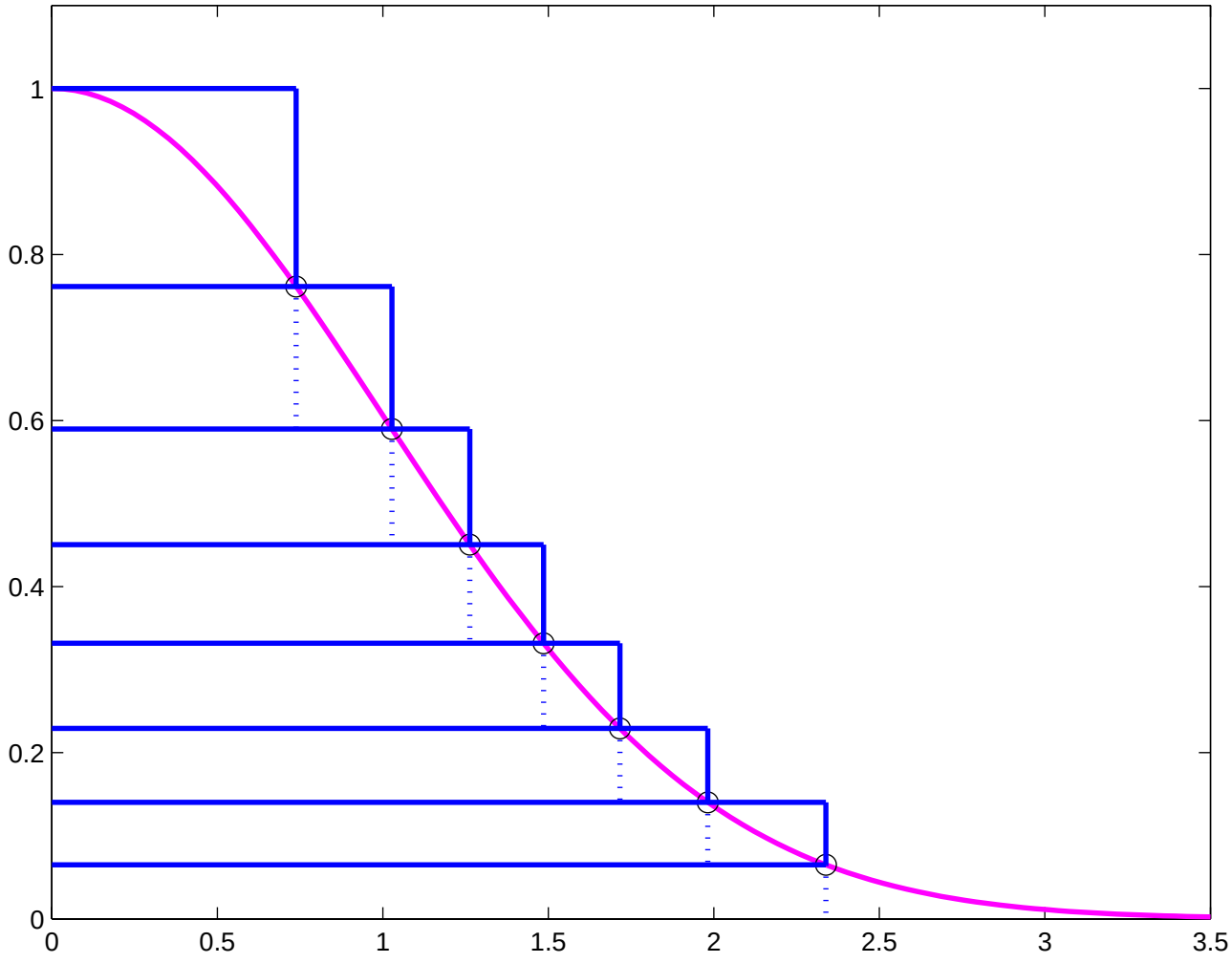
Ziggurats were a form of temple common to the Sumerians, Babylonians and Assyrians. The earliest examples date from the end of

the third millenium BC, the latest from the 6th century, BC. The ziggurat was a step-like pyramidal structure, Notable examples are the ruins at Ur and Khorsabad in Mesopotamia (in today's Iraq). Step Pyramids resembling ziggurats were built by the Egyptians and the Mayan people of Central America. Mathematically ziggurats resembles a two-dimensional step function. The ziggurat algorithm here is based on the one-dimensional version of the structure.

The ziggurat method[5] is a highly efficient rejection method based on covering the target density with a set of horizontal rectangles and a bottom tail section. These sections are chosen so that it is easy to choose uniform points and to determine whether they should be retained or rejected.

We will illustrate the method for the case of $n = 8$ sections so that diagrams are clearer to see. In practice 64, 128 or 256 sections are used. The top 7 section are rectangles, and the bottom section consists of a rectangular part with a strip tailing off to infinity. All these 8 sections have the same area, v, so that it is easy to choose one of the sections at random. Furthermore, 7 of the 8 section are rectangles from which it is easy to to get a random point (x, y), and

further yet, for those rectangles, it is easy to decide if if (x, y) falls below $f(x)$. Because if the rightmost coordinates of the rectangles are $0 = x_0 < x_1 < x_2 < \ldots < x_7$, and rectangle R_i is selected, $i > 0$, then the x-coordinate of a random point in R_i is ux_i, where u is a uniform deviate in $(0, 1)$, and if $x < x_{i-1}$ then the random point (x, y) must be below $f(x)$, confirming the acceptance of x without having to generate y. Of course if $x \geq x_{i-1}$ then we have to generate y and and compare it with $f(x)$ to decide if that random point is to be accepted. In practice we choose a large enough n so that x_{i-1}/x_i is only slightly less than unity, so that $x < x_{i-1}$ nearly most of the time.



To obtain a random point (x, y) from the base strip, which consists of a rectangle with an adjoining infinite tail, we first define r to be the rightmost x_i, so that the tail corresponds to $x > r$. We may generate from the base strip as follows: generate $x = vu/f(r)$, with u uniform

in $(0, 1)$. If $x < x$ we then return x, else we return an x from the tail. For the normal tail, we can use the Marsaglia[6] procedure: generate $x = -\ln(u_1)/r, y = -\ln(u_2)$ until $y + y > x \times x$, then return $r + x$.

In most applications, uniform variates u are provided by floating a random integer (32-bit unsigned long), one may save time by incorporating the float operation into the step that forms the $x's$ that are to be returned. This can be done by forming the 32-bit integer $k_i = 2^{32}(x_{i-1}/x_i)$, and setting $w_i = .5^{32}x_i$. For the special index $i = 0$, set $k_0 = 2^{32}rf(r)/v$ and $w_0 = .5^{32}v/f(r)$. The Ziggurat algorithm[5] can then be stated as follows.

The Ziggurat Algorithm

1. Generate a random 32-bit integer j and let i be the index provided by the rightmost bits of j.
2. Set $x = jw_i$. If $j < k_i$ return x.
3. If $i = 0$ return an x from the tail.
4. If $[f(x_{i-1}) - f(x_i)]u < f(x) - f(x_i)$, return x.
5. Go to step 1.

3.1. Setting up the Ziggurat

We have to first set up the ziggurat.[5] This is done only once during the initialization part of the program. Given a probability density $f(x)$, we want to find $n-1$ equal-area rectangles, and an equal-area base strip consisting of a rectangle plus an infinite tail. The union of these sections must cover entirely $f(x)$, such as pictured in the figure. Note that we have only shown the positive x portion of the curve. It is also clear that we can work with the un-normalized probability density function, so that we do not have to carry the normalization

factor, c. Clearly $f(x)$ and v scale as c, and $x_1, x_2, \ldots$ and $x_{n-1} = r$ are independent of the scale.

Given an n, how does one find the common area v, and the right edges of the rectangles: $0 = x_0 < x_1 < x_2 < \ldots < x_{n-1} = r$. It is clear from the figure that

$$x_i[f(x_{i-1}) - f(x_i)] = v,$$

for $i = 1, 2, \ldots, n-1$. and

$$v = rf(r) + \int_r^\infty dx\ f(x).$$

Therefore we can define a function $z(r)$ by the following steps.

First we set

$$v = rf(r) + \int_r^\infty dx\ f(x).$$

. Then for i from $n-1$ by steps of -1 to 1, we compute $x_i = f^{-1}(v/x_{i+1} + f(x_{i+1}))$. Finally we return $(v - x_1 + x_1 f(x_1))$ as the value of the function.

Then the problem is to find the value of r that makes $z(r) = 0$.

This root-finding task clearly has to be done numerically. It turns out that the numerical work requires a bit of care. For r large, v is small, all the x_i can be computed readily, but $z(r)$ is negative. However when r small, v is relatively large, $x_1, x_2, \ldots$ become complex because of the computation of f^{-1}. For a given n, let r_0 be the value of r at which x_1 is zero. Thus x_1 becomes complex for $r < r_0$.

When n is large (the usual case) the correct value for x_1 lies very close to zero. Thus for r very slightly below the correct, x_1 become complex and numerical computation runs into problems. This makes finding the root somewhat difficult. It turns out that there is an effective way to alleviate the problem.[7]

For $n = 256$, one finds that $r = 3.654152885361009$, and the common area $v = 0.00492867323399$. The efficiency of the rejection procedure, given by

$$e = \frac{\frac{\sqrt{2\pi}}{2n}}{v},$$

gives 99.33% for this n. Clearly the efficiency increases with increasing n. The results for r, v and efficiency for various n are shown in the

following table.

n	r	v	% efficiency
8	2.338371698247252 4	1.761736401187775 9e-1	88.93
16	2.675536765737613 5	8.398946374782730 0e-2	93.26
32	2.961300121264019 3	4.075874443221987 1e-2	96.09
64	3.213657627158895 5	2.002445715735170 0e-2	97.80
128	3.442619855896651 9	9.912563035336472 6e-3	98.78
256	3.654152885361008 8	4.928673233974657 1e-3	99.33
512	3.852046150368391 6	2.456766351541352 9e-3	99.64

The ziggurat algorithm was implemented and run-times were compared with two other methods said to be fast. The ziggurat method was found to be the fastest, faster than the runner-up by a factor of three. MATLAB 6 in fact uses an older version of this algorithm to generate normally distributed random numbers. A 800 MHz Pentium laptop PC can generate over 10 million random numbers in less than one second.

References

[1] D. D. Wackerly, W. Mendenhall III, and R. L. Scheaffer, *Mathematical Statistics with Applications*, Sec. 8.3, (Wadssworth Publishing, 1996). 5

[2] P. Bratley, B. L. Fox and E. L. Schrage, *A Guide to Simulation*, (Springer-Verlag, New York, 1983 5

[3] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery *Numerical Recipes*, Second Edition, (Cambridge University Press, 1992). 5

[4] D. E. Knuth, *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*, Second Edition, (Addison Wesley, Reading, MA, 1969). 6

[5] G. Marsaglia and W. W. Tsang, *The ziggurat method for generating random variables*, J. of Statistical Software, **5**, p. 1-7 (2000). 7, 10, 11

[6] G. Marsaglia, *Generating a variable from the tail of the normal distribution*, Technometrics, **6**, p. 101-102 (1964). 10

[7] K. M. Leung, unpublished result.

13