AN ABSTRACT OF THE THESIS WRITTEN BY

John David Keighley for the Master of Science in

Mathematics presented on July 21, 1989.

Title: The Exact Approximation Method in Distribution Simulation

Abstract approved: [illegible]

An algorithm for generating random variates quickly, called the exact approximation method is the subject of this thesis. Two other algorithms, the acceptance rejection method, and the inversion method for generating random variates are also included. The exact approximation method uses the acceptance rejection method. The inversion method is a special case of the exact approximation method.

To generate random variates quickly using the exact approximation method a function which is a close approximation to the inverse cummulative probability function must be found. The choice of this function is a comprimise between it's ease of computation, and it's closeness to the inverse cummulative distribution function. Since, both factors affect the efficiency of the exact approximation algorithm.

THE EXACT APPROXIMATON METHOD IN DISTRIBUTION SIMULATION

A Thesis

Presented to

The Division of Mathematical and Physical Sciences

EMPORIA STATE UNIVERSITY

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

John David Keighley

July 1989

thesis
[illegible]
K

James L Wolfe

Approved for the Graduate Council

L. Scott

Approved for the Major Department

ACKNOWLEDGMENTS

I would like thank Dr. Larry Scott for his time, assistant, and patiance throughout the writting of this paper. Also I want to thank Dr. Zorabi Honargohar, Dr. William Simpson, and Dr. Charles Greenlief.

Finally, I would like to thank my parents for their faith and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter I

INTRODUCTION

As computers become more powerful scientists are using them more frequently to simulate or model stochastic events. To simulate an event a probability density function that describes the occurrances of that event must be found. Then an algorithm that will generate random variates having the observed probability density function must be written.

The purpose of this thesis is to look at some of the many methods that can be used to generate random variates from a desired distribution. This chapter states the problem and gives an example. In Chapter 2 some basic definitions and theorems of probability and real analysis will be stated. Chapter 3 covers the theory of random variate generation that this paper uses with Chapter 4 giving some applications and results. Chapter 5 is a summary of the paper.

Statement of the problem

This paper explains an efficient method for generating random variates Many random variate generators require the use of tables or complex algorithms. The disadvantage of table-aided procedures is the large amount of memory needed to hold the tables. The problem with complex algorithms is that with increasing complexity in the code it becomes very difficult to debug a program. The exact approximation method developed by George Mardsaglia(1961) avoids major programming problems and promises to use small numbers of uniform deviates.

Applications

As an example of where the exact approximation method could be used think about a traffic simulation. Assume that the flow of traffic has been observed long enough to determine that the time between arrivals of cars can be described by an exponential distribution. The easy way to generate random variates in this case would be to find the cummulative distribution function and then invert it to get the inverse cummulative distribution function. Since the cummulative distribution function of any probability density function must be uniformly dense from 0 to 1. To generate random variates that will have the desired probability density function [0, 1) uniform deviates can be generated and put into the inverse cummulative distribution function. This is a description of the inversion method which has a 1-1 relationship between the number of [0, 1) uniform deviates used and random variates generated. The difficulity with this method for the exponential distribution is that the inverse cummulative distribution function will use the natural logrithm function to generate random variates. Computation of the natural logrithm function is relatively slow. With a small increase in programming complexity the exact approximation method can be used which should generate random variates in less time.

Chapter II

DEFINITIONS AND THEOREMS

This paper explains a method of generating random variates. Some basic definitions and theorems in probability and real analysis needed to understand Chapter 3 are explained in this chapter. The first definition is the concept of a random variable.

Definition 2.1: A random variable is a function which maps the sample space S to the real line. Random variables will be denoted by uppercase letters.

An illustration of Definition 2.1 is given in Figure 2.1

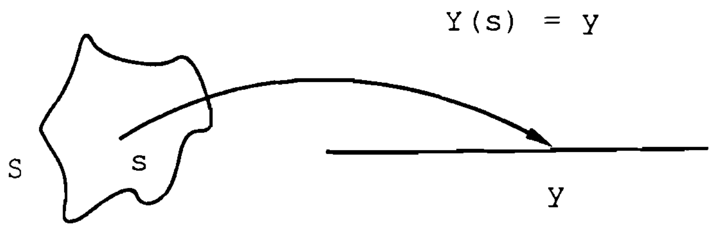

Fig 2.1

The random variable is considered to be a discrete random variable if the range is finite or it contains a countably infinite number of values. A continuous random variable has a range which includes an interval of real numbers, bounded or unbounded.

For every discrete random variable there exists a probability distribution function (pdf) $f_Y(y)$. Pdfs are either discrete or continuous and they are defined in different ways.

Definition 2.2 : For any discrete random variable Y, $f_Y(y)$ is the sum of probabilities that maps all $s \in S$ to y by the random variable Y.

That is:

$$f_Y(y) = P(\{s \in S \mid Y(s) = y\}).$$

For any point $s \notin S$ $f_Y(y) = 0$.

Also, as is true with any probability function, the summation of $f_Y(y)$ over all possible outcomes must be equal to one.

For example, to find the probability that the sum of two dice is 6. Define the random variable Y such that $Y(s) = Y((a,b)) = a+b$ where a and b are the numbers on the dice. Then

$$\begin{aligned} f_Y(y) &= P(\{s \in S \mid Y(s) = 6\}) \\ &= P((1, 5), (2, 4), (3, 3), (4, 2), (5, 1)) \\ &= P(1, 5) + P(2, 4) + P(3, 3) + P(4, 2) + P(5, 1) \\ &= 1/36 + 1/36 + 1/36 + 1/36 + 1/36 \\ &= 5/36 \end{aligned}$$

Figure 2.2 shows $f_Y(y)$ for all y:

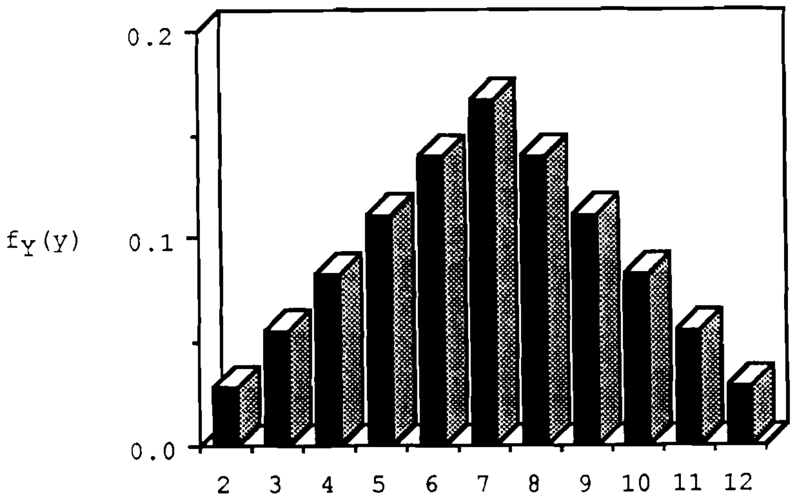

Fig 2.2

If Y is a continuous random variable there exists a probability density function (pdf) $f_Y(y)$.

Definition 2.3: For any continuous random variable Y, $f_Y(y)$ is a continuous curve over the sample space S which contains the points a and b such that;

$$P(a \leq Y \leq b) = P(\{s \in S \mid a \leq Y(s) \leq b\}) = \int_a^b f_Y(y)\, dy.$$

Where $f_Y(y)$ must satisfy the following two conditions.

$$f_Y(y) \geq 0$$

and

$$\int_S f_Y(y)\, dy = 1.$$

Figure 2.3 gives a graphical representation of the definition.

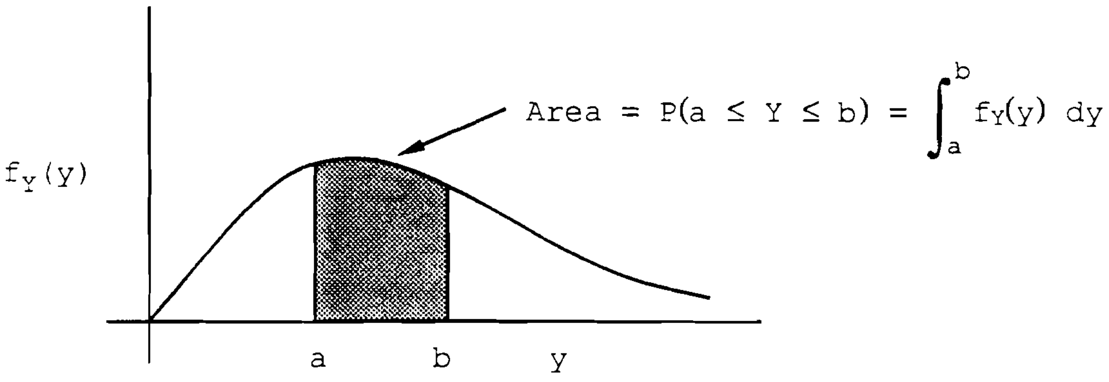

Fig 2.3

Note that from the definition for a continuous random variable $P(Y(s) = y) = 0$ where $s \in S$ since

$$P(Y(s) = y) = \int_y^y f_Y(y)\, dy = 0.$$

Associated with every random variable Y whether discrete or continuous is a cummulative density function (cdf) $F_Y(y)$.

Definition 2.4: For any random variable Y defined on a sample space S with probability function P. The cummulative density function (cdf) $F_Y(y)$ is the probability corresponding to the set of sample points in S that are mapped by Y into values on the real line less than or equal to y.

More formally.

$$F_Y(y) = P(\{s \in S \mid Y(s) \leq y\}).$$

For a discrete random variable

$$F_Y(y) = \sum_{s \leq y} f_Y(y)$$

and for the continuous random variable.

$$F_Y(y) = \int_{-\infty}^{y} f_y(y)\, dy$$

Remembering the previous dice example

$$F_Y(6) = P(\{s \in S \mid Y(s) \le 6\}) = \sum_{s \le 6} f_Y(y)$$

$$= P(Y(s)=2) + P(Y(s)=3) + P(Y(s)=4) + P(Y(s)=5) + P(Y(s)=6)$$

$$= 1/36 + 2/36 + 3/36 + 4/36 + 5/36$$

$$= 15/36$$

$$= 5/12$$

Figure 2.4 shows $F_Y(y)$ for all Y.

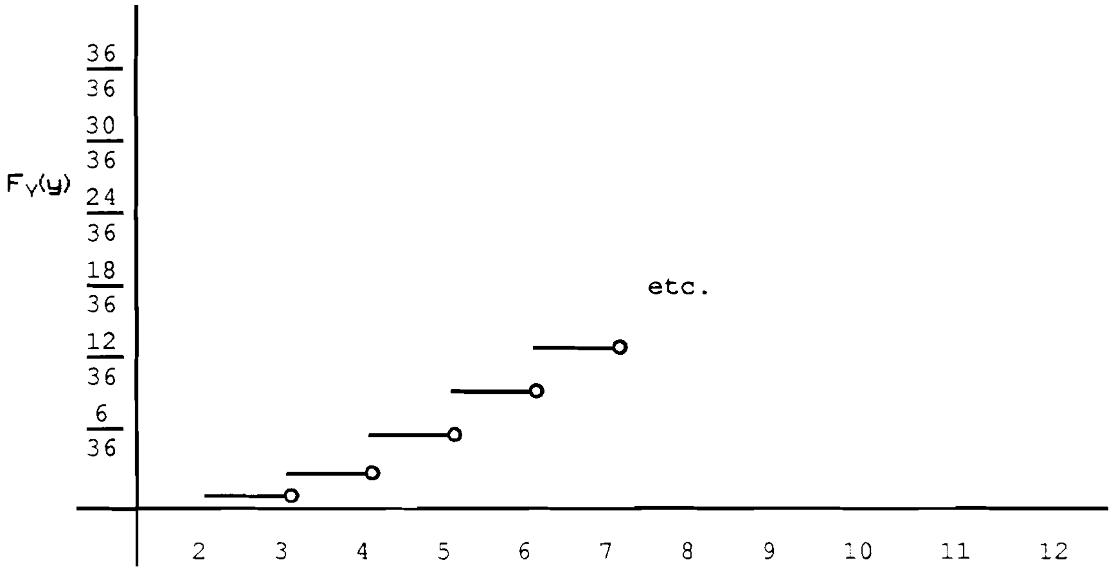

Fig 2.4 Dice Sum y

As Figure 2.4 shows, when Y is a discrete random variable $F_Y(y)$ will be a step function with steps occuring at values of y for which $f_Y(y) > 0$.

When Y is a continuous random variable, the following assertions can be made about $F_Y(y)$. First, $F_Y(y)$ is continuous and second, $F_Y(y)$ is monotonicly increasing.

Definition 2.5 is the the definition of a function that is pointwise continuous. For $F_Y(y)$ to be continuous it must at least satisfy definition 2.5.

Definition 2.5: Suppose $S \subset \Re$ and that $g: S \rightarrow \Re$. If $x_0 \in S$, then g is continuous at x_0 iff for each $\epsilon > 0$, there is a $\delta > 0$ such that if

$$|x - x_0| < \delta, \qquad x \in S,$$

then

$$|g(x) - g(x_0)| < \epsilon.$$

If g is continuous at x for every $x \in S$, then g is continuous.

From Definition 2.3 $f_Y(y)$ is continuous, if $f_Y(y)$ is defined over a closed interval [a,b], then $f_Y(y)$ must also bounded. This leads to Theorem 2.1.

Theorem 2.1: For all $y \in [a,b]$ with $f_Y(y) : [a,b] \rightarrow \Re$ with $f_Y(y)$ bounded and $f_Y(y) \in R$. Define $F_Y(y) = \int_a^y f_Y(y)dy$ for $a \le y \le b$. Then $F_Y(y)$ is continuous on [a,b].

Proof: Choose $M > 0$ such that $| f_Y(y) | \le M$ for all $y \in [a, b]$. Choose $\epsilon > 0$.

Let $\delta = \epsilon/M$. Thus, if $| x - y | < \delta$, $x, y \in [a, b]$, then

$$| F_Y(x) - F_Y(y) | = \left| \int_a^x f_Y(y)\, dy - \int_a^y f_Y(y)\, dy \right|$$

$$= \left| \int_x^y f_Y(y)\, dy \right|$$

$$\le | x - y | M$$

$$< \delta M = (\epsilon/M)(M) = \epsilon$$

Therefore $F_Y(y)$ is continuous on [a,b]. In fact, $F_Y(y)$ is uniformly continuous on [a,b].

To prove that $F_Y(y)$ is monotonicly increasing we look at any two events y_1 and y_2 with $y_1 < y_2$. Since $y_1 < y_2$, if the event $y < y_1$ has occured, then the event $y < y_2$ has also occured, since $y < y_1 < y_2$.

By definition

$$F_Y(y_1) = P(Y \le y_1) = \int_y^{y_1} f_y(y)dy$$

$$F_Y(y_2) = P(Y \le y_2) = \int_y^{y_2} f_Y(y)dy = \int_{y_1}^{y_2} f_Y(y)\, dy + \int_y^{y_1} f_Y(y)dy$$

$$F_Y(y_2) - F_Y(y_1) = \int_{y_1}^{y_2} f_y(y)dy \ge 0.$$

A useful property of cdf's that we will use often is the relation between a continuous pdf and it's derivative. Note that the following theorem is only proved for the case when $f_Y(y)$ is defined over a closed interval [a,b]. The case where $f_Y(y)$ is defined over an open interval or $f_Y(y)$ is piecewise continuous or piecewise monotonic is beyond the scope of this thesis.

Theorem 2.2: Let $f_Y(y)$ be an integrable function on [a,b]. For $y \in [a,b]$, let

$$F_Y(y) = \int_a^y f_Y(y)\, dy.$$

Then $F_Y(y)$ is continuous on [a,b]. If $f_Y(y)$ is continuous at y_0 in [a,b] then $F_Y(y)$ is differentiable at y_0 and

$$F'_Y(y_0) = f_Y(y_0).$$

Proof: Suppose that $f_Y(y)$ is continuous at $y_0 \in$ [a,b]. Note that

$$\frac{F_Y(y) - F_Y(y_0)}{(y - y_0)} = \frac{1}{y - y_0} \int_{y_0}^{y} f_Y(y)\, dy$$

for $y \neq y_0$.

$$\frac{F_Y(y) - F_Y(y_0)}{(y - y_0)} - f_Y(y_0) = \frac{1}{y - y_0} \int_{y_0}^{y} f_Y(y)\, dy - f_Y(y_0) \qquad (2.1)$$

and since

$$f_Y(y_0) = \frac{1}{y - y_0} \int_{y_0}^{y} f_Y(y_0)\, dy$$

$$\frac{F_Y(y) - F_Y(y_0)}{(y - y_0)} - f_Y(y_0) = \frac{1}{y - y_0} \left[\int_{y_0}^{y} f_Y(y)\, dy - \int_{y_0}^{y} f_Y(y_0)\, dy \right]$$

$$= \frac{1}{y - y_0} \int_{y_0}^{y} \left[f_Y(y) - f_Y(y_0) \right] dy$$

Chose $\epsilon > 0$. Since $f_Y(y)$ is continuous at y_0, there exists $\delta > 0$ such that

$y \in (a,b)$ and $|y - y_0| < \delta$ imply $|f_Y(y) - f_Y(y_0)| < \epsilon$

therefore it follows from (2.1) that

$$\left| \frac{F_Y(y) - F_Y(y_0)}{(y - y_0)} - f_Y(y_0) \right| \leq \epsilon.$$

Showing that

$$\lim_{y \to y_0} \frac{F_Y(y) - F_Y(y_0)}{y - y_0} = f_Y(y_0)$$

or

$$F'_Y(y_0) = f_Y(y_0)$$

Again note that what is shown is only proven for the case where $f_Y(y)$ is continuous over a closed interval. In other words Theorem 2.2 is a proof that $F'_Y(y_0) = f_Y(y_0)$ when definite integrals are being used. The proof for indefinite integrals is beyond the scope of this paper although indefinite integrals are used. The definitions and theorems in this chapter will be used in Chapter 3.

Chapter III

DESCRIPTION OF THE ALGORITHMS

The Acceptance Rejection Method

The first algorithm for random variate generation is the acceptance rejection algorithm. This method was first suggested by John Von Neumann(1951). To implement this algorithm, the probability density function must be known. In it's simplest form the acceptance rejection method requires that the probability density function be bounded from above by some constant which will be called c on a closed interval [a,b]. The definition of c is

$$c = \max \{ f_Y(y) \mid a \leq y \leq b \}.$$

Figure 3.1 is an example of a pdf bounded by c.

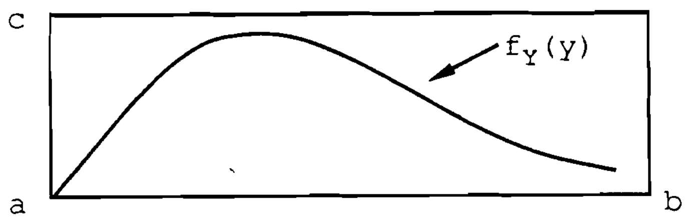

Fig 3.1 y

When these conditions are met the following algorithm will generate a random variable with the same pdf as $f_Y(y)$.

1.) Generate x where X is uniform on [a,b].

2.) Generate y where Y is uniform on [o,c].

3.) If $y \leq f_Y(x)$, then output x, else goto step 1.

The probability that $y \leq f_Y(x)$ will be equal to $f_Y(x)/c$. To minimize the number of points rejected, c must be the least upper bound of $f_Y(y)$. Two things should be noted about this algorithm. First, it requires a minimum of two [0, 1) uniform deviates to generate one random variate. Second, if $f_Y(x)$ is small in comparison to c such as in the tail of $f_Y(y)$ in Figure 3.1 the probability of accepting x will be small thereby wasting two [0, 1) uniform deviates. The advantage of this method is, as long as $f_Y(y)$ is known this algorithm is very easy to implement.

<u>The Inversion Method</u>

To implement the inversion method the probabiltiy density function, cummulative density function, and the inverse cummulative density function must all be known. If all three functions are known the implementation of the inversion method will be trivial. Starting with a known pdf, $f_Y(y)$ the cdf, $F_Y(y)$ is found. Then the inverse of $F_Y(y)$ is found. The domain of the inverse cdf will be the range of $F_Y(y)$ which implies that the domain of $F_Y^{-1}(y)$ will be the interval $[0,1)$. By generating a random variate y from the range of $F_Y(y)$ and placing y into $F_Y^{-1}(y)$ the random variate will have the same pdf as $f_Y(y)$. DeGroot(1975) gives a theorem showing the random variables Y must have a uniform distribution.

Theorem 3.1: If a random variable X has a continuous cumulative distribution $F_X(x)$, and $Y = F_X(x)$ the distribution of Y must be a uniform distribution on the interval $(0, 1)$.

Proof: First, since $F_X(x)$ is the cdf of a random variable, then $0 \le F_X(x) \le 1$ for $-\infty < x < \infty$. Therefore, $P(Y < 0) = P(Y > 1) = 0$. Next, for any given value of y in the interval $0 < y < 1$, let x_0 be a number such that $F_X(x_0) = y$. If $F_X(x)$ is strictly increasing, there will be a unique number x_0 such that $F_X(x_0) = y$. However, if $F_X(x_0) = y$ over an entire interval of values of x, the x_0 can be chosen artibrarily from this interval. If $G_Y(y)$ denotes the cdf of Y, then

$$G_Y(y) = P(Y \le y) = P(X \le x_0) = F_X(x_0) = y.$$

Hence, $G_Y(y) = y$ for $0 < y < 1$. Since this function is the cdf of a uniform distribution of the interval $(0, 1)$, this uniform distribution is the distribution of Y.

The following illustrates how to generate random variates from the exponential distribution using the inversion method. The general form of the exponential pdf is

$$f_Y(y) = \frac{1}{\lambda} e^{-\frac{y}{\lambda}}, \quad y > 0.$$

The cdf is found by integrating $f_Y(y)$ giving

$$F_Y(y) = \int_0^y \frac{1}{\lambda} e^{-\frac{y}{\lambda}} dy$$

$$= - e^{-\frac{y}{\lambda}} \Big|_0^y$$

$$= \left[- e^{-\frac{y}{\lambda}} - (- e^{-\frac{0}{\lambda}})\right]$$

$$= (1 - e^{-\frac{y}{\lambda}}).$$

The inverse cdf $F_Y^{-1}(y)$ can be found by setting

$$x = F_Y(y) = 1 - e^{-\frac{y}{\lambda}},$$

and solving for y in terms of x

$$x = 1 - e^{-\frac{y}{\lambda}}$$

$$1 - x = e^{-\frac{y}{\lambda}}$$

$$\ln(1 - x) = -\frac{y}{\lambda}$$

$$y = - \lambda \ln(1 - x). \qquad (3.1)$$

The random variable X has a range of (0, 1). In fact from Theorem 3.1, X must be uniformly distributed over the interval (0, 1). This implies that the difference (1 − X) is also distributed uniformly over (0, 1), therefore replaceing (1 − X) by X removes one more calculation. So the final form of (3.1) is

$$y = -\lambda \ln(x)$$

where X is uniformly distributed over ([0, 1). A random variate y from an exponential distribution could be generated using a simple Turbo C statement like

$$y = -\lambda \log(\text{random}(\text{seed}))$$

where random(seed) returns a (0,1) uniform variate. The problem with the inversion method is that the pdf, cdf, and the inverse cdf must be known to implement it. Sometimes the cdf or the inverse cdf are impossible to determine thereby excluding inversion as a possible means to generate random variates from that distribution. Another problem with inversion is that sometimes functions like ln, sin, cos, etc.. must be used. The computation of these functions are relatively slow when compared to addition, subtraction, multiplication and division.

The Exact Approximation Method

The exact approximation method will use an easy to calculate approximation of the inverse cdf which doesn't need to be known. Even though the procedure will use an approximation to the inverse cdf the random variates generated will have the required distribution. Since the function being used to generate the random variates is an approximation to the inverse cdf the uniform distribution can't be used to generate the random variates since the random variates would not follow the pdf exactly. A function $g_Y(y)$ will be used to adjust the aprroximate inverse cdf to generate random variates that have the required pdf.

The basic idea of exact approximation is as follows: let $w_U(u)$ be a monotonicly increasing differentiable function where the derivative is continuous on the interval [0, 1). The transformed random variable $X \leftarrow w_U(u)$ will have the desired probability density $f_X(x)$ if the density of the U-derviates is $g_U(u) = f_X(w_U(u)) \cdot w_U'(u)$. Hogg and Craig(1965) gives the following theorem.

Theorem 3.1: Let X be a random variable of the continuous type having pdf $f_X(x)$. Let A be the one dimensional space where $f_X(x) > 0$. Consider the random variable $Y = u(X)$, where $y = u(x)$ defines a one to one tranformation which maps the set A onto the set B. Let the inverse of $y = u(x)$ be denoted by $x = w(y)$, and let the derivative $dx/dy = w'(y)$ be continuous and not vanish for all points y in B. Then the pdf $g_Y(y)$ of the random variable $Y = u(X)$ is given by

$$g_Y(y) = f_X[w(y)] \cdot |w'(y)|, \quad y \in B$$
$$= 0 \text{ elsewhere}$$

Proof:

$$G_Y(y) = P(Y \leq y) = P(u(X) \leq y)$$

$$= P(u^{-1}[u(X)] \leq u^{-1}(y))$$

$$= P(X \leq u^{-1}(y))$$

$$= P(X \leq w(y)) \quad \text{since } u^{-1}(y) = w(y)$$

$$= F_X(w(y))$$

$$F_X(x) = \int_a^x f_X(x)\, dx \qquad \text{def}^n \text{ of cdf}$$

$$F_X(x) = F_X(w(y)) = \int_{w(a)}^{w(y)} f_X(w(y))\, w'(y)\, dy \qquad \text{change of variable theorem}$$

$$g_Y(y) = \frac{dx}{dy}\int_{w(a)}^{w(y)} f_X(w(y))\, w'(y)\, dy \qquad \text{def}^{n} \text{ of pdf}$$

$$= f_X(w(y))\, w'(y) \qquad \text{fundamental theorem of Calculus.}$$

As long as $w(y)$ is a monotonicly increasing function

$$g_Y(y) = f_X[w(y)]\, w'(y)$$

Since $w'(y) > 0$ for all y

An interesting side note is what happens when $w_U(u) = F_X^{-1}(y)$ First from Gaughan(1975) the inverse-function theorem.

Theorem 3.2: Suppose $F:[a,b]\rightarrow\Re$ is continuous and differentiable with $F'(x) \neq 0$ for all $x \in [a,b]$. Then F is 1–1, F^{-1} is continuous and differentiable on $F([a,b])$, and

$$(F^{-1})'(F(x)) = \frac{1}{F'(x)}$$

for all $x \in [a,b]$.

Proof: Since $f(x) \neq 0$ for all $x \in [a,b]$, F is 1–1. Let us suppose that $F([a,b]) = [c,d]$. Choose $y_0 \in [c,d]$ and $\{y_n\}_{n=1}^{\infty}$ any sequence in $[c,d]\setminus\{y_0\}$ converging to y_0. Let

$$x_0 = F^{-1}(y_n)$$

for $n = 0, 1, 2, \ldots$. Since $F:[a,b]\rightarrow\Re$ is continuous and 1–1 and $[a,b]$ compact F^{-1} must be continuous and $\{x_n\}_{n=1}^{\infty}$ converges to $x_0 = F^{-1}(y_0)$, and, since F^{-1} is 1–1, $x_n \neq x_0$ for all n. By the differentiability of F,

$$\left\{ \frac{F(x_n) - F(x_0)}{x_n - x_0} \right\}_{n=1}^{\infty}$$

converges to $f(x_0)$. By hypothesis, $f(x_0) \neq 0$ and

$$\frac{F(x_n) - F(x_0)}{x_n - x_0} \neq 0$$

for n = 0, 1, 2, ... Hence

$$\left\{ \frac{F^{-1}(y_n) - F^{-1}(y_0)}{y_n - y_0} \right\}_{n=1}^{\infty} = \left\{ \frac{x_n - x_0}{F(x_n) - F(x_0)} \right\}_{n=1}^{\infty}$$

converges to $1/(F'(x_0))$. Thus, F^{-1} is differentiable and

$$(F^{-1})'(F(x_0)) = \frac{1}{F'(x_0)}. \qquad (3.2)$$

Then Equation 3.2 can be rewritten as

$$(F^{-1})'(y_0) = \frac{1}{F'(F^{-1}(y_0))}.$$

Theorem 3.3: If $w_U(u) = F_X^{-1}(y_0)$ where $F_X(x)$ is a function that satisfies the conditions of Theorem 3.2 the $g_U(u)$ in Theorem 3.1 will be the uniform distribution and exact approximation reduces to the inversion method.

$$g_U(u) = f_X(w_U(u)) \cdot w_U'(u)$$

where $F_X(x) = F(x) = y_0$, $f_X(x) = F'(x) = f(x)dx$, and $w_U(u) = F_X^{-1}(y_0) = F^{-1}(y_0)$

$$(F^{-1})'(y_0) = \frac{1}{F'(F^{-1}(y_0))}.$$

$$= \frac{1}{f_X(F_X^{-1}\ (y_0))}$$

$$f_X(F_X^{-1}(y_0)) = \frac{1}{(F_X^{-1})'(y_0)}$$

$$g_U(u) = f_X(F_X^{-1}(y_0))\ (F_X^{-1})'(y_0)$$

$$= f_X(F_X^{-1}(y_0)) \left[\frac{1}{f_X(F_X^{-1}\ (y_0))} \right]$$

$= 1$

This implies that $g_U(u)$ must be the uniform distribution.

Theorem's 3.2 and 3.3 together implies that exact approximation is a generalization of inversion. If the function $F_X^{-1}(y_0)$ can't be found then $w_U(u)$ a function that approxiamtes $F_X^{-1}(y_0)$ will be used. From Theorem 3.1 any function that is monotonicly increasing, and has a continuous derivative that exists over the interval [0, 1) will suffice. Although any function in this class will work a function that will generate the random variates as quickly as possible is desired. If $w_U(u)$ is close to the inverse cdf than $g_Y(y)$ will be close to uniform This allows $g_Y(y)$ to be expressed as a mixture of a dominant uniform and a residual density. Define the constant p such that $p = \min\{g_Y(y) \mid 0 \leq y \leq 1\}$. Define another constant r such that $r = \max\{g_Y(y) \mid 0 \leq y \leq 1\}$. Also let $h = r - p$ which from the definition of p and r means $p \leq g_Y(y) \leq r$ for $0 \leq y \leq 1$. To implement the algorithm a uniform deviate U is generated and checked for $U < p$. If $U < p$ then U is also less than $g_Y(y)$ therefore accept U and use U/p in $w_U(u)$ to generate a random variate. If $U > p$ a random variate needs to be taken from the region $p \leq g_Y(y) \leq r$ Use the the acceptance rejection method to generate another variate. Specificly generate U and determine $g_Y(U)$ then generate another uniform deviate U' and find $U'h + p$. If $U'h + p \leq g_Y(U)$ accept U and use U in $w_U(u)$ to generate another random variate. If $U'h + p > g_Y(U)$ then generate two uniform deviates to be used in $U'h + p \leq g_Y(U)$ and continue until it's accepted.

Chapter IV

APPLICATIONS

This chapter will demonstrate the application of the exact approximation method as it is explained in Chapter 3. Two different probability density functions will be used. For each of the pdf's at least two different approximations for each inverse cdf will be demonstrated. Also where possible the inversion and acceptance rejection methods will be used. The time it takes the different methods will be tabled for comparison.

In Chapter 3 the exponential distribution was inverted and it was shown that random variates that have a exponential distribution can be generated using the equation

$$x = -\lambda \ln(\text{random}(\text{seed})).$$

If $\lambda = 1$ then generating x simplifies to

$$x = -\ln(\text{random}(\text{seed})).$$

Where random(seed) would generate a (0,1) uniform deviate. This function will generate one exponential distributed random variate for each (0,1) uniform deviate used. The problem with this is the time required to determine the value of ln(random(seed)).

In this chapter, two functions approximating the truncated inverse cdf of the exponential distribution will be used. The first approximation to the truncated inverse cdf was suggested by Ahrens and Dieter(1988). The exponential distribution e^{-y} can be rewritten as

$$f_Y(y) = 2^{-(k+1)}\, 2e^{-z}, \qquad (4.1)$$

where

$$z = y - k\ln(2), \qquad k = 0,1,\dots \text{ and } 0 \le z \le \ln(2).$$

Then $2^{-(k+1)}$ is a geometric distribution where $P(K = k) = \left(\frac{1}{2}\right)^{k+1}$ The leading bit zeros of a (0, 1) uniform deviate has this distribution. The second half of equation (4.1) is a truncated exponential distribution with pdf $h_Z(z) = 2e^{-z}$ in the interval [0, ln(2)] . Inverting $h_Z(z) = 2e^{-z}$ we obtain

$$H_Z(z) = \int_0^z 2e^{-z}\,dz = -2\left[e^{-z} \Big|_0^z \right] = -2\left[e^{-z} - 1\right] = 2\left[1 - e^{-z}\right]$$

this implies that $H_Z^{-1}(u) = -\ln(1 - u/2)$ where u is a [0,1) uniform deviate. Ahrens and Dieter(1988) used the following function as an approximation for $H_Z^{-1}(z)$:

$$w_U(u) = \frac{a}{b - u} + c \qquad u \sim U(0,\ 1) \qquad (4.2)$$

where

$$w_U(0) = 0 \quad \text{and} \quad w_U(1) = \ln(2)$$

since $H_Z^{-1}(0) = 0$ and $H_Z^{-1}(1) = \ln(2)$.

The conditions $w_U(0) = 0$ and $w_U(1) = \ln(2)$ results in

$$w_U(0) = \frac{a}{b - 0} + c = 0$$

$$\frac{a}{b} + c = 0$$

$$a = -bc$$

$$w_U(1) = \frac{a}{b - 1} + c = \ln(2)$$

$$\frac{a}{b - 1} = \ln(2) - c$$

$$a = (b - 1)(\ln(2) - c)$$

Setting $a = -bc$ and $a = (b - 1)(\ln(2) - c)$ equal to each other and solving for c yields

$$-bc = (b - 1)(\ln(2) - c)$$

$$-bc = b\ln(2) - \ln(2) - bc + c$$

$$0 = (b - 1)\ln(2) + c$$

$$c = -(b - 1)\ln(2).$$

Solving $a = -bc$ in terms of b gives

$$a = -b(-(b - 1)\ln(2))$$

$$a = b(b - 1)\ln(2)$$

The last free parameter b must be fixed such that the minimum of

$$g_U(u) = h_Z[\, w_U(u)\,]\cdot[w_U'(u)]$$

$$= 2e^{-w_U(u)} \cdot w_U'(u)$$

is close to one for $0 \le u < 1$. Where $g_U(u)$ has a distribution that is close to the uniform distribution.

Solving $w_U(u)$ in terms of b gives:

$$w_U(u) = \frac{b(b - 1)\ln(2)}{b - u} + [(-(b - 1)\ln(2)\,]$$

$$= \frac{b(b - 1)\ln(2)}{b - u} + \frac{[(-(b - 1)\ln(2)(b - u)]}{b - u}$$

$$= \frac{\ln(2)\,[\,b(b - 1) - (b - 1)(b - u)]}{b - u}$$

$$= \frac{\ln(2)\,[b^2 - b - (b^2 - bu - b + u)]}{b - u}$$

$$= \frac{\ln(2)\,[b^2 - b^2 - b + b + bu - u]}{b - u}$$

$$= \frac{\ln(2)\,[\,u\,(b - 1)]}{b - u}$$

Solving $w_U'(u)$ in terms of b gives

$$w_U'(u) = \ln(2)\,(b - 1)\,\frac{d}{du}[\,u(b - u)^{-1}]$$

$$= \ln(2)\,(b - 1)\left[(b - u)^{-1} + (-1)(-1)\,u\,(b - u)^{-2}\right]$$

$$= \ln(2)\,(b - 1)\left[(b - u)^{-1} + u\,(b - u)^{-2}\right]$$

$$= \ln(2)\,(b - 1)\left[\frac{1}{b - u} + \frac{u}{(b - u)^2}\right]$$

$$= \ln(2)\,(b - 1)\left[\frac{b - u + u}{(b - u)^2}\right]$$

$$= \ln(2)\,(b - 1)\left[\frac{b}{(b - u)^2}\right]$$

$$= \frac{b(b - 1)\ln(2)}{(b - u)^2}$$

Thus the form of $g_U(u)$ in terms of b is

$$g_U\,(u) = 2e^{\left[-\frac{\ln(2)\left[u\,(b-1)\right]}{b-u}\right]}\left[\frac{b(b - 1)\ln(2)}{(b - u)^2}\right]$$

The values p, r, and h are found from $g_U(u)$. The value of b must be found such that p will be maximumized and r will be minimumized. To determine the conditions that will maximumize p the function $g_U(u)$ was graphed using several artbitary values of b The resulting curve was a parabola opening downward. This implies that the maximum p will occur when $g_U(0) = g_U(1)$.

$$g_U\,(0) = 2e^{\left[-\frac{\ln(2)\,[0]\,(b-1)}{b-0}\right]}\left[\frac{b(b - 1)\ln(2)}{(b - 0)^2}\right]$$

$$= 2e^{0}\left[\frac{(b - 1)\ln(2)}{b}\right]$$

$$= \frac{2(b - 1)\ln(2)}{b} \qquad (4.3)$$

$$g_U\,(1) = 2e^{\left[-\frac{\ln(2)\,[1]\,(b-1)}{b-1}\right]}\left[\frac{b(b-1)\ln(2)}{(b-1)^2}\right]$$

$$= 2e^{\left[-\ln(2)\right]}\left[\frac{b\ln(2)}{(b-1)}\right]$$

$$= 2\left[\frac{1}{2}\right]\left[\frac{b\ln(2)}{(b-1)}\right]$$

$$= \frac{b\ln(2)}{(b-1)} \qquad (4.4)$$

Setting Equations 4.3 and 4.4 equal to each other and solving for b results in

$$\frac{2\,(b-1)\ln(2)}{b} = \frac{b\ln(2)}{(b-1)}$$

$$2\,(b-1)(b-1) = b^2$$

$$2\,(b^2 - 2b + 1) - b^2 = 0$$

$$2b^2 - b^2 - 4b + 2 = 0$$

$$b^2 - 4b + 2 = 0$$

Using the quadratic formula to solve for b gives

$$b = \frac{4 \pm \sqrt{(-4)^2 - 4(1)(2)}}{2}$$

$$= \frac{4 \pm \sqrt{16 - 8}}{2}$$

$$= \frac{4 \pm \sqrt{8}}{2}$$

$$= \frac{4 \pm 2\sqrt{2}}{2} = 2 \pm \sqrt{2}$$

The value $b = 2 - \sqrt{2}$ must be discarded since $g_U(u)$ will generate negative values for that value. Thus the values of the three parameters are

$$a = b(b-1)\ln(2)$$

$$= (2 + \sqrt{2})(2 + \sqrt{2} - 1) \ln(2)$$

$$= (2 + \sqrt{2})(1 + \sqrt{2}) \ln(2)$$

$$= (2 + 3\sqrt{2} + 2) \ln(2)$$

$$= (4 + 3\sqrt{2}) \ln(2)$$

$$b = 2 + \sqrt{2}$$

$$c = -(b - 1) \ln(2)$$

$$= -(2 + \sqrt{2} - 1) \ln(2)$$

$$= -(1 + \sqrt{2}) \ln(2)$$

The p value for $g_U(u)$ can be found by solving $g_U(0)$ or $g_U(1)$ since the curve is a parabola opening downward and $g_U(0) = g_U(1)$. From Equation (4.3)

$$g_U(0) = \frac{2(b - 1) \ln(2)}{b}$$

$$= \frac{2(2 + \sqrt{2} - 1) \ln(2)}{(2 + \sqrt{2})}$$

$$= \frac{2(1 + \sqrt{2}) \ln(2)}{(2 + \sqrt{2})}$$

$$= 0.9802581434$$

The value of r can be found by setting $g_U'(u) = 0$ and solving for u.

$$g_U'(u) = 0 = \frac{d}{du}\left[2e^{-\left[\frac{a}{b-u} + c\right]}\left[\frac{a}{(b - u)^2}\right]\right]$$

$$= 2\left[e^{-\left[\frac{a}{b-u} + c\right]}\left\{\frac{a}{(b - u)^2}\right\}^2 + e^{-\left[\frac{a}{b-u} + c\right]}\left\{\frac{2a}{(b - u)^3}\right\}\right]$$

$$= \left[e^{-\left[\frac{a}{b-u} + c\right]}\right]\left[\left\{\frac{a}{(b - u)^2}\right\}^2 + \left\{\frac{2a}{(b - u)^3}\right\}\right]$$

since $e^{-\left[w_U(u) \right]} > 0$ for all u it can be canceled out leaving

$$0 = \left[\left\{\frac{a}{(b-u)^2}\right\}^2 + \left\{\frac{2a}{(b-u)^3}\right\} \right]$$

$$= \frac{a^2}{(b-u)^4} + \frac{2a}{(b-u)^3}$$

$$= \frac{a^2}{(b-u)^4} + \frac{2a(b-u)}{(b-u)^4}$$

Multipling both sides by $(b-u)^4$

$$0 = a^2 + 2a(b-u)$$

$$= a^2 + 2ab - 2au$$

$$2au = a^2 + 2ab$$

$$u = \frac{a}{2} + b$$

So the maximum value r of $g_U(u)$ occurs when $u = b + \frac{a}{2}$ and that value is

$$r = g_U(b + \tfrac{a}{2}) = 1.010089582$$

The last constant $h = r - p$ is

$$h = 0.0298314386$$

The exact approximation method is used to generate observations from $h_Z(z) = 2e^{-z}$ for $0 \le z \le \ln(2)$ as follows. Generate a [0, 1) uniform deviate U, if $U < p$ then use U/p as a new [0, 1) uniform deviate and calculate $w_U(u)$:

$$w_U(u) = \frac{a}{b - U/p} + c = \frac{A}{B - U} + c$$

where

$$A = ap \quad \text{and} \quad B = bp.$$

The acceptance rejectance method will only be needed if $p > U$. This has a probability of 1 in $50.65 = 1/(1 - p)$. To use the acceptance rejectance method generate a new [0, 1) uniform deviate U and calculate $w_U(u)$ from Equation (4.2). Then generate another [0, 1) uniform deviate U' and accept $w_U(u)$ if $U' \le (g_U(u) - p)/h$ or $U'h + p \le g_U(u)$. The time needed to make each comparison of values can be shortened by simplifing $U'h + p \le g_U(u)$ in the following manner.

$$g_U(u) = 2e^{-W_U(u)} \cdot W_U'(u)$$

$$2e^{-W_U(u)} \cdot W_U'(u) \ge U'h + p$$

$$e^{-W_U(u)} \ge \frac{U'h + p}{2\,W_U'(u)}$$

$$\ge \frac{U'h + p}{2\left[\frac{a}{(b - U)^2}\right]}$$

$$\ge (U'h + p)\left[\frac{(b - u)^2}{2a}\right]$$

$$\ge \left[U'\frac{h}{2a} + \frac{p}{2a}\right](b - u)^2$$

$$\ge (U'H + P)(b - u)^2$$

where

$$H = \frac{h}{2a} \text{ and } P = \frac{p}{2a}.$$

x	$f_X(x)$	$x \cdot f_X(x)$
1	p	$1 \cdot p$
3	$(1 - p)\left[\frac{1 - p}{h}\right]$	$\frac{3(1 - p)^2}{h}$
5	$(1 - p)\left[1 - \left[\frac{1 - p}{h}\right]\right]\left[\frac{1 - p}{h}\right]$	$5\left[\frac{h - 1 + p}{h}\right]\left[\frac{(1 - p)^2}{h}\right]$
7	$(1 - p)\left[1 - \left[\frac{1 - p}{h}\right]\right]^2\left[\frac{1 - p}{h}\right]$	$7\left[\frac{h - 1 + p}{h}\right]^2\left[\frac{(1 - p)^2}{h}\right]$
⋮	.	.

Fig 4.1

The number of [0, 1) uniform deviates used per random variate generated is of great importance. For the algorithm to generate random variates quickly the number of random variates generated compared to the [0, 1) uniform deviates used must be close to 1-1. This value can be found by finding some probability distribution function which describes the number of [0, 1) uniform random deviates used. Figure 4.1 shows this distribution. Since this is a pdf, the summation over all x must be equal to 1.

$$\sum_{\text{all } x} f_X(x) = p + \sum_{k=1}^{\infty} \left[\frac{(1-p)^2}{h}\right] \left[\frac{(h-1+p)}{h}\right]^{(k-1)}$$

Letting $L = K - 1 \Rightarrow L + 1 = K$

$$\sum_{\text{all } x} f_X(x) = p + \sum_{L=0}^{\infty} \left[\frac{(1-p)^2}{h}\right] \left[\frac{(h-1+p)}{h}\right]^{L}$$

$$= p + \left[\frac{(1-p)^2}{h}\right] \sum_{L=0}^{\infty} \left[\frac{(h-1+p)}{h}\right]^{L}$$

$$= p + \left[\frac{(1-p)^2}{h}\right] \left[\frac{1}{1 - \left[\frac{(h-1+p)}{h}\right]}\right]$$

$$= p + \left[\frac{(1-p)^2}{h}\right] \left[\frac{1}{\left[\frac{h-h+1-p}{h}\right]}\right]$$

$$= p + \left[\frac{(1-p)^2}{h}\right] \left[\frac{1}{\left[\frac{1-p}{h}\right]}\right]$$

$$= p + \left[\frac{(1-p)^2}{h}\right] \left[\frac{h}{1-p}\right]$$

$$= p + 1 - p$$

$$= 1$$

Since the summation of $f_X(x) = 1$, $f_X(x)$ is a pdf and the expected value of $f_X(x)$ can be found. The expected number of uniform deviates used per random variate generated which is shown in Figure 4.1 under $x \cdot f_X(x)$ can be determined in the following manner. The x corresponds to the number of uniform deviates used and $f_X(x)$ is the probability of acceptance for using x number of uniform random deviates. The expected number of [0, 1) uniform deviates $E(X) = \sum_{\text{all } x} x \cdot f_X(x)$ can be expressed as

$$\sum_{\text{all } x} x \cdot f_X(x) = 1 \cdot p + \sum_{k=1}^{\infty} (2k + 1) \left[\frac{(1 - p)^2}{h}\right] \left[\frac{(h - 1 + p)}{h}\right]^{(k - 1)}.$$

Letting $L = K - 1 \Rightarrow L + 1 = K$

$$\sum_{\text{all } x} x \cdot f_X(x) = p + \left[\frac{(1 - p)^2}{h}\right] \sum_{L=0}^{\infty} (2L + 3) \left[\frac{(h - 1 + p)}{h}\right]^{L}. \quad (4.5)$$

To determine whether the summation converges the ratio test is used giving a result of

$$\lim_{L \to \infty} \left|\frac{a_{k+1}}{a_k}\right| = \lim_{L \to \infty} \left| \frac{(2L + 5) \left[\frac{(h - 1 + p)}{h}\right]^{(L + 1)}}{(2L + 3) \left[\frac{(h - 1 + p)}{h}\right]^{L}} \right|$$

$$= \lim_{L \to \infty} \left| \frac{(2L + 5) \left[\frac{(h - 1 + p)}{h}\right]}{(2L + 3)} \right|$$

$$= \lim_{L \to \infty} \left| \frac{(2L + 5)}{(2L + 3)} \right| \left[\frac{h - 1 + p}{h}\right]$$

$$= \lim_{L\to\infty} \left|\frac{(2 + 5/L)}{(2 + 3/L)}\right| \left[\frac{h - 1 + p}{h}\right]$$

$$= \left|\frac{(2 + 0)}{(2 + 0)}\right| \left[\frac{h - 1 + p}{h}\right]$$

$$= \left[\frac{h - 1 + p}{h}\right] < 1.$$

Therefore the sequence is convergent and the sequence can be split into the sum of two sequences, making it easier to find the sum of the sequence. Splitting Equation (4.5)

$$\sum_{\text{all } x} x \cdot f_X(x) = p + \left[\frac{(1 - p)^2}{h}\right] \sum_{L=0}^{\infty} \left[(2L) \left[\frac{(h - 1 + p)}{h}\right]^L + (3) \left[\frac{(h - 1 + p)}{h}\right]^L\right]$$

$$= p + \left[\frac{(1 - p)^2}{h}\right] \left[\sum_{L=0}^{\infty} (2L) \left[\frac{(h - 1 + p)}{h}\right]^L + (3) \sum_{L=0}^{\infty} \left[\frac{(h - 1 + p)}{h}\right]^L\right]$$

$$= p + \left[\frac{(1 - p)^2}{h}\right] \left[\sum_{L=0}^{\infty} (2L) \left[\frac{(h - 1 + p)}{h}\right]^L + (3) \left[\frac{1}{1 - \left\{\frac{(h - 1 + p)}{h}\right\}}\right]\right]$$

$$= p + \left[\frac{(1 - p)^2}{h}\right] \left[\sum_{L=0}^{\infty} (2L) \left[\frac{(h - 1 + p)}{h}\right]^L + (3) \left[\frac{1}{\left\{\frac{h - h + 1 - p}{h}\right\}}\right]\right]$$

$$= p + \left[\frac{(1 - p)^2}{h}\right] \left[\sum_{L=0}^{\infty} (2L) \left[\frac{(h - 1 + p)}{h}\right]^L + (3) \left[\frac{1}{\left\{\frac{1 - p}{h}\right\}}\right]\right]$$

$$= p + \left[\frac{(1 - p)^2}{h}\right] \left[\sum_{L=0}^{\infty} (2L) \left[\frac{(h - 1 + p)}{h}\right]^L + (3) \left[\frac{h}{1 - p}\right]\right]$$

$$= p + \left[\frac{(1 - p)^2}{h}\right] \sum_{L=0}^{\infty} (2L) \left[\frac{(h - 1 + p)}{h}\right]^L + (3) \left[\frac{(1 - p)^2}{h}\right] \left[\frac{h}{1 - p}\right]$$

$$= p + \left[\frac{(1-p)^2}{h}\right] \sum_{L=0}^{\infty} (2L)\left[\frac{(h-1+p)}{h}\right]^L + 3(1-p)$$

$$= p + 3 - 3p + \left[\frac{(1-p)^2}{h}\right] \sum_{L=0}^{\infty} (2L)\left[\frac{(h-1+p)}{h}\right]^L$$

Letting $y = \dfrac{h-1+p}{h}$ gives

$$= 3 - 2p + \left[\frac{2(1-p)^2}{h}\right] \sum_{L=0}^{\infty} L(y)^L$$

$$= 3 - 2p + \left[\frac{2(1-p)^2}{h}\right] (y + 2y^2 + 3y^3 + \dots)$$

$$= 3 - 2p + \left[\frac{2(1-p)^2}{h}\right] (y)(1 + 2y + 3y^2 + \dots)$$

$$= 3 - 2p + \left[\frac{2(1-p)^2}{h}\right] (y)\frac{d}{dy}(y + y^2 + y^3 + \dots)$$

$$= 3 - 2p + \left[\frac{2(1-p)^2}{h}\right] (y)\frac{d}{dy}\left[\frac{1}{1-y}\right]$$

$$= 3 - 2p + \left[\frac{2(1-p)^2}{h}\right] (y)\left[\frac{1}{(1-y)^2}\right]$$

$$= 3 - 2p + \left[\frac{2(1-p)^2}{h}\right] \left[\frac{y}{(1-y)^2}\right]$$

$$= 3 - 2p + \left[\frac{2(1-p)^2}{h}\right] \left[\frac{\left\{\dfrac{h-1+p}{h}\right\}}{\left\{1 - \left[\dfrac{h-1+p}{h}\right]\right\}^2}\right]$$

$$= 3 - 2p + \left[\frac{2(1-p)^2}{h}\right] \left\{\frac{h-1+p}{h}\right\} \left[\frac{1}{\left\{\dfrac{h-h+1-p}{h}\right\}^2}\right]$$

$$= 3 - 2p + \left[\frac{2(1-p)^2}{h}\right] \left\{\frac{h-1+p}{h}\right\} \left[\frac{1}{\left\{\dfrac{1-p}{h}\right\}^2}\right]$$

$$= 3 - 2p + \left[\frac{2(1-p)^2}{h}\right] \left\{\frac{h-1+p}{h}\right\} \left\{\frac{h}{1-p}\right\}^2$$

$$= 3 - 2p + \left[\frac{2(1-p)^2}{h^2}\right]\left[\frac{h^2}{(1-p)^2}\right] (h - 1 + p)$$

$$= 3 - 2p + 2(h - 1 + p)$$

$= 3 - 2 - 2p + 2p + 2h$

$= 1 + 2h$

Since the probability of using x uniform deviates is dependent uoon p and h it is independent of the function being used for $w_U(u)$. The expected number of uniform deviates used to generate a random variate is $1 + 2h$ for any $w_U(u)$. For (4.2) the expected number of [0, 1) uniform deviates used to generate one random variate is

$$1 + 2h = 1.0596628772999.$$

To test the first approximation of the inverse cdf against another approximation, a second approximation for $2e^{-z}$ was derived. Let

$$w_U(u) = c_1u + c_2u^2 + c_3u^3 \qquad u \sim U(0, 1) \qquad (4.6)$$

where

$$w_U(0) = 0 \text{ and } w_U(1) = \ln(2)$$

since $H_Z^{-1}(0) = 0$ and $H_Z^{-1}(1) = \ln(2)$.

The conditions $w_U(0) = 0$ and $w_U(1) = \ln(2)$ results in

$$w_U(0) = c_1(0) + c_2(0)^2 + c_3(0)^3 = 0$$

$$w_U(1) = c_1(1) + c_2(1)^2 + c_3(1)^3 = \ln(2)$$

$$c_1 + c_2 + c_3 = \ln(2)$$

$$c_2 = \ln(2) - c_1 + c_3 \qquad (4.7)$$

Finding $w_U'(u)$ and $w_U''(u)$

$$w_U'(u) = c_1 + 2c_2u + 3c_3u^2$$

$$w_U''(u) = 2c_2 + 3c_3u$$

Solving $g_U(0)$ and $g_U(1)$ gives

$$g_U(0) = 2e^{-w_U(0)}(c_1 + 2c_2(0) + 3c_3(0)^2)$$

$$= 2e^0c_1$$

$$= 2c_1$$

$$c\left[\frac{2}{\ln(2)} - \frac{1}{\ln(2)}\right] = 1$$

$$c\left[\frac{1}{\ln(2)}\right] = 1$$

$$c = \ln(2)$$

The pdf that will be used for all three approximations to the inverse cdf is

$$h_Z(z) = \ln(2)\ 2^z , \qquad 0 \le z \le 1.$$

For the first approximation let

$$w_U(u) = c_1 u + c_2 u^2,$$

then

$$w_U'(u) = c_1 + 2c_2 u,$$

$$w_U''(u) = 2c_2,$$

and

$$g_U(u) = \ln(2)\ 2^{w_U(u)}\ w_U'(u).$$

The values that $w_U(u)$ takes on at $u = 0$ and $u = 1$ are

$$w_U(0) = 0 \quad \text{and} \quad w_U(1) = 1$$

since $\quad H_Z(0) = 0 \quad$ and $\quad H_Z(1) = 1$.

Solving $w_U(1) = 1$ results in

$$w_U(1) = 1 = c_1(1) + c_2(1)^2$$

$$1 = c_1 + c_2$$

$$c_2 = 1 - c_1.$$

therefore

$$g_U(0) = \ln(2)\ 2^{w_U(0)}(\ c_1 + 2c_2(0))$$

$$= \ln(2)\ 2^0\ (c_1)$$

$$= \ln(2)\ c_1 \qquad (4.8)$$

and

$$g_U(1) = \ln(2)\ 2^{w_U(1)}(\ c_1 + 2c_2(1))$$

$$g_U(1) = 2e^{-W_U(1)}(c_1 + 2c_2(1) + 3c_3(1)^2)$$

$$= 2e^{-\ln(2)}(c_1 + 2(\ln(2) - c_1 - c_3) + 3c_3)$$

$$= 2\left[\frac{1}{2}\right](c_1 - 2c_1 + 3c_3 - 2c_2 - 2\ln(2))$$

$$= (-c_1 + c_3 - 2\ln(2))$$

Imposing an equal wave effect results in

$$g_U(0) + g_U(1) = 2$$

$$2c_1 + (-c_1 + c_3 - 2\ln(2)) = 2$$

$$c_1 + c_3 + 2\ln(2) = 2$$

$$c_1 = 2 - 2\ln(2) - c_3.$$

To find the value of c_3 that would give the maximium value of p set the derivative $g_U'(u) = 0$ and solve

$$g_U'(u) = 0 = 2\frac{d}{du}\left[e^{-W_U(u)}W_U'(u)\right]$$

$$= 2\left[e^{-W_U(u)}\{(W_U'(u))\cdot(-W_U'(u))\} + e^{-W_U(u)}W_U''(u)\right]$$

$$= 2e^{-W_U(u)}\left[\{(W_U'(u))\cdot(-W_U'(u))\} + W_U''(u)\right]$$

Since $2e^{-W_U(u)} > 0$ for all u divide both sides by $2e^{-W_U(u)}$ giving

$$0 = \{(W_U'(u))\cdot(-W_U'(u))\} + W_U''(u)$$

$$= (c_1u + c_2u^2 + c_3u^3)(-c_1u - c_2u^2 - c_3u^3) + (2c_2 + 3c_3u)$$

$$= -9c_3^2u^4 - 12c_3c_2u^3 - (6c_3c_1 + 4c_2^2)u^2 + (-4c_2c_1 + 6c_3)u + c_1^2 + 2c_2$$

Rather then trying to solve this equation, a direct search was used to find the values

$c_1 = 0.5000000000,$

$c_2 = 0.0794415416,$

$c_3 = 0.1137056389.$

Using these values in (5) results in

$p = 0.9829974429,$

$r = 1.016422347,$

$h = 0.0334249041.$

To use the exact approximation method generate a [0, 1) uniform deviate U, if U < p then use U/p as a new [0, 1) uniform deviate and calculate $w_U(u)$ as follows:

$$w_U(u) = c_1u + c_2u^2 + c_3u^3$$

$$w_U(u) = C_1u + C_2u^2 + C_3u^3$$

where

$$C_1 = \frac{c_1}{p} \quad \text{and} \quad C_2 = \frac{c_2}{p^2} \quad \text{and} \quad C_3 = \frac{c_3}{p^3}.$$

When sampling from the region above p as before two [0, 1) uniform deviates will be required for each test for rejection. Simplifing $U'h + p \leq g_U(u)$ to reduce computations gives

$$\frac{U'h + p}{2} \leq e^{-w_U(u)} w_U(u)$$

$$U'H + P \leq e^{-w_U(u)} w_U(u)$$

where

$$H = \frac{h}{2} \text{ and } P = \frac{p}{2}.$$

Note, that although the p value for this function is larger than the p value for the first approximate cdf resulting in the acceptance rejection method being needed with a probability of 1 in 58.81 = 1/(1 − p). The expected number of [0, 1) uniform deviates used per random variate is

$$1 + 2h = 1.066849808.$$

Another disadvantage for this function is the time of computation. The function is more computation intensive than the first approximate cdf which will slow it down.

A table of the different methods for generating random variates is given in Table 4.1. The numbers in the columns refer to time taken to generate 10,000 random variates. For example under the IBM PC column 16.62 is 16 seconds and 62 hundredths of a second.

Table 4.1

Method of generation	IBM PC[1]	Zenith[2]	AT&T 6300[3]
Inversion	16.62	114.24	37.79
Acceptance Rejection	> 1 hour	> 4 hours	> 2 hours
Exact Approximation 1[4]	22.33	144.63	54.45
Exact Approximation 2[5]	24.65	191.57	71.73

The second probability density function was solved using the exact approximation method for 3 different approximations of $H_Z^{-1}(z)$. The pdf $h_Z(z) = c\ 2^z \quad 0 \le z \le 1$ had to be solved for c first.

$$H_Z(z) = \int_0^1 c\ 2^z dz = 1$$

$$c \left[\frac{2^z}{\ln(2)} \right] \Bigg|_0^1 = 1$$

[1] IBM PC running at 4.7 Mhz with a 8088 processor and a 8087 coprocessor.

[2] Zenith running at 6 Mhz with a 8088 processor

[3] AT&T 6300 running at 10 Mhz with a 80286 processor.

[4] The exact approximation method using $\frac{a}{b - u} + c$ as the approximation to the inverse cdf.

[5] The exact approximation method using $c_1u^1 + c_2u^2 + c_3u^3$ as the approximation to the inverse cdf.

$$= \ln(2)\ 2^{1}\ (c_1 + 2c_2)$$

$$= \ln(2)\ 2\ (c_1 + 2(1 - c_1))$$

$$= 2\ \ln(2)\ (c_1 + 2 - 2c_1)$$

$$= 2\ \ln(2)\ (2 - c_1) \qquad (4.9)$$

Since the function $g_U(u)$ is a parabola to find the values for c_1 and c_2 that will give the maximium p value set (4.8) and (4.9) equal and solve for c_1.

$$\ln(2)\ c_1 = 2\ \ln(2)\ (2 - c_1)$$

$$c_1 = 4 - 2c_1$$

$$c_1 + 2c_1 = 4$$

$$c_1 = 4/3$$

Since $c_2 = 1 - c_1$

$$c_2 = 1 - 4/3$$

$$c_2 = -\ 1/3$$

Using the c_1 and c_2 to find

$$p = g_U(0) = \ln(2)\ c_1$$

$$= 0.9241962407$$

To find r take the derivative of $g_U(u)$, set it equal to 0 and solve for u.

$$\frac{d}{du}\left[\ g_U(u)\right] = 0 = \ln(2)\left[\ 2^{w_U(u)}\ \{\ln(2)\}\{\ w_U'(u)\}^2 + 2^{w_U(u)} w_U''(u)\right]$$

$$= \ln(2)\ 2^{w_U(u)}\left[\ \{\ln(2)\}\{\ w_U'(u)\}^2 + w_U''(u)\right]$$

Since $\ln(2)\ 2^{w_U(u)} > 0$ for all u divide by $\ln(2)\ 2^{w_U(u)}$ to get

$$0 = \ln(2)\ \{\ w_U'(u)\}^2 + w_U''(u)$$

$$= \ln(2)\ \{c_1 + 2c_2u\}^2 + 2c_2$$

$$= \ln(2)\ (c_1 + 2c_2u)(c_1 + 2c_2u) + 2c_2$$

$$= \ln(2)\ (c_1^2 + 4c_1c_2u + 4c_2^2u^2) + 2c_2$$

$$= c_1^2 + \frac{2c_2}{\ln(2)} + 4c_1c_2u + 4c_2^2u^2$$

Using the quadratic formula to solve for u gives

$$u = \frac{-4c_1c_2 \pm \sqrt{(4c_1c_2)^2 - 4(4c_2^2)(2c_2/\ln(2) + c_1^2)}}{2(4c_2^2)}$$

$$u = \frac{-4c_1c_2 \pm \sqrt{(4c_1c_2)^2 - (16c_2^2)(2c_2/\ln(2) + c_1^2)}}{8c_2^2}$$

$$u = \frac{-4c_1c_2 \pm \sqrt{16c_1^2c_2^2 - 32c_2^3/\ln(2) - 16c_2^2\,c_1^2}}{8c_2^2}$$

$$u = \frac{-4c_1c_2 \pm \sqrt{(16)(-2c_2^3/\ln(2))}}{8c_2^2}$$

$$u = \frac{-4c_1c_2 \pm 4\sqrt{-2c_2^3/\ln(2)}}{8c_2^2}$$

$$u = \frac{-c_1c_2 \pm c_2\sqrt{-2c_2^1/\ln(2)}}{2c_2^2}$$

$$u = \frac{-c_1 \pm \sqrt{-2c_2/\ln(2)}}{2c_2}$$

Using the value

$$u = \frac{-c_1 - \sqrt{-2c_2/\ln(2)}}{2c_2}$$

in $w_U(u)$ gives negative values so it is discarded, and the following value of u is kept

$$u = \frac{-c_1 + \sqrt{-2c_2/\ln(2)}}{2c_2},$$

$$u = 0.52893149.$$

The value of r is

$$r = g_U(0.52893149) = 1.038946508.$$

Therefore the value of h is

$$h = 0.1147502673.$$

To generate random variates from $h_Z(z)$ can be done as follows. Generate a [0, 1) uniform deviate U, if $U < p$ then use U/p as a new [0, 1) uniform deviate and calculate $w_U(u)$ as follows:

$$w_U(u) = c_1u + c_2u^2$$
$$= C_1u + C_2u^2$$

where

$$C_1 = \frac{c_1}{p} \quad \text{and} \quad C_2 = \frac{c_2}{p^2}$$

The acceptance rejection method will be used approximately 1 in 8.71 = $1/(1 - p)$ trials. When sampling from the region above p as before two [0, 1) uniform deviates will be required for each test for rejection. Simplify the test $U'h + p \le g_U(U)$ by

$$\frac{U'h + p}{\ln(2)} \le 2^{w_U(u)} w_U'(u)$$

$$U'H + P \le 2^{w_U(u)} w_U'(u)$$

where

$$H = \frac{h}{\ln(2)} \text{ and } P = \frac{p}{\ln(2)}.$$

The expected number of [0, 1) uniform deviates used per random variate generated is

$$1 + 2h = 1.229500535.$$

Since $1 + 2h$ is relatively large this function might be expected to run relatively slow. Figure 4.3 shows this is the case when the computer has a math coprocesser. When the computer doesn't have a math coprocesser the function may run faster than a function that has a larger minimum, if it's harder to compute.

The second approximation to the inverse cdf is

$$w_U(u) = c_1u + c_2u^2 + c_3u^3.$$

The first and second derivative of $w_U(u)$ are

$$w_U'(u) = c_1 + 2c_2u + 3c_3u^2,$$

$$w_U''(u) = 2c_2 + 6c_3u,$$

with

$$g_U(u) = \ln(2)\ 2^{w_U(u)}\ w_U'(u).$$

The values that $w_U(u)$ takes on at $u = 0$ and $u = 1$ are

$$w_U(0) = 0 \quad \text{and} \quad w_U(1) = 1$$

since $\quad H_Z(0) = 0 \quad$ and $\quad H_Z(1) = 1$.

Solving $w_U(1) = 1$ results in

$$w_U(1) = 1 = c_1(1) + c_2(1)^2 + c_3(1)^3$$

$$1 = c_1 + c_2 + c_3$$

$$c_2 = 1 - c_1 - c_3.$$

Solving $g_U(u)$ at 0 and 1 gives

$$g_U(0) = \ln(2)\ 2^{w_U(0)}(\ c_1 + 2c_2(0) + c_3(0)^3)$$

$$= \ln(2)\ 2^0\ (c_1)$$

$$= \ln(2)\ c_1$$

and

$$g_U(1) = \ln(2)\ 2^{w_U(1)}(\ c_1 + 2c_2(1) + 3c_3(1)\ )$$

$$= \ln(2)\ 2^1\ (c_1 + 2c_2 + 3c_3)$$

$$= \ln(2)\ 2\ (c_1 + 2(1 - c_1 - c_3) + 3c_3)$$

$$= 2\ \ln(2)\ (c_1 + 2 - 2c_1 - 2c_3 + 3c_3)$$

$$= 2\ \ln(2)\ (2 - c_1 + c_3)$$

Imposing an equal wave effect results in

$$g_U(0) + g_U(1) = 2$$

$$2 = \ln(2)\ c_1 + 2\ \ln(2)\ (\ 2 - c_1 + c_3)$$

$$2 = \ln(2)\left[c_1 + 4 - 2c_1 + 2c_3\ \right]$$

$$\frac{2}{\ln(2)} = 4 - c_1 + 2c_3$$

$$c_1 = 4 - \frac{2}{\ln(2)} + 2c_3$$

$$c_1 = \frac{4\ln(2) + 2c_3\ln(2) - 2}{\ln(2)}$$

$$c_2 = 1 - c_1 - c_3$$

Setting the derivative $g_U'(u) = 0$ and solving gives

$$g_U'(u) = 0 = \ln(2)\frac{d}{du}\left[2^{w_U(u)}\, w_U'(u)\right]$$

$$= \ln(2)\left[2^{w_U(u)}\{\ln(2)\}\{(w_U'(u))(w_U'(u))\} + 2^{w_U(u)} w_U''(u)\right]$$

$$= \ln(2)\, 2^{w_U(u)}\left[\{\ln(2)\}\{(w_U'(u)(w_U'(u))\} + w_U''(u)\right]$$

Since $\ln(2)\, 2^{w_U(u)} > 0$ for all u divide both sides by $\ln(2)\, 2^{w_U(u)}$ giving

$$0 = \{\ln(2)\}\{(w_U'(u))(w_U'(u))\} + w_U''(u)$$

$$= (\ln(2))\,(c_1u + c_2u^2 + c_3u^3)(-c_1u - c_2u^2 - c_3u^3) + (2c_2 + 3c_3u)$$

Then by using a direct search the values

$p = 0.983009554$

$r = 1.01699045$

$h = 0.033980892$

$c_1 = 1.442680$

$c_2 = -0.606715$

$c_3 = 0.164035$

were found. The exact approximation method for sampling from $h_Z(z) = \ln(2)\, 2^{-z}$ can now procceed as follows. Generate a [0, 1) uniform deviate U if $U < p$ then use U/p as a new [0, 1) uniform deviate and calculate $w_U(u)$ as follows:

$$w_U(u) = c_1u + c_2u^2 + c_3u^3$$

$$w_U(u) = C_1u + C_2u^2 + C_3u^3$$

where

$$C_1 = \frac{c_1}{p} \quad \text{and} \quad C_2 = \frac{c_2}{p^2} \quad \text{and} \quad C_3 = \frac{c_3}{p^3}.$$

The acceptance rejection method will be used approximately 1 in 58.86 = 1/(1 − p) trials. To use the acceptance rejection method simplify $U'h + p \leq g_U(u)$ to

$$\frac{U'h + p}{\ln(2)} \leq 2^{w_U(u)} w_U'(u)$$

$$U'H + P \leq 2^{w_U(u)} w_U'(u)$$

where

$$H = \frac{h}{\ln(2)} \text{ and } P = \frac{p}{\ln(2)}.$$

The expected number of [0, 1) uniform deviates used to generate one random variate is

$1 + 2h = 1.067961784.$

The third and last function approximating the inverse cdf is

$$w_U(u) = \frac{a}{b - u} + c \qquad u \sim U(0, 1) \quad (4.10)$$

where

$$w_U(0) = 0 \quad \text{and} \quad w_U(1) = 1$$

since $H_Z^{-1}(0) = 0$ and $H_Z^{-1}(1) = 1.$

The conditions $w_U(0) = 0$ and $w_U(1) = 1$ results in

$$w_U(0) = \frac{a}{b - 0} + c = 0$$

$$\frac{a}{b} + c = 0$$

$$a = -bc$$

$$w_u(1) = \frac{a}{b-1} + c = 1$$

$$\frac{a}{b-1} = 1 - c$$

$$a = (b-1)(1-c)$$

Setting $a = -bc$ and $a = (b-1)(1-c)$ equal to each other and solving for c yields

$$-bc = (b-1)(1-c)$$

$$-bc = b - 1 - bc + c$$

$$0 = b - 1 + c$$

$$c = -(b-1) .$$

Solving $a = -bc$ in terms of b gives

$$a = -b(-(b-1))$$

$$a = b(b-1)$$

The last free parameter b must be fixed such that the minimum of

$$g_u(u) = h_z\left[\ w_u(u)\ \right]\cdot\left[w_u'(u)\right]$$

$$= \ln(2)\ 2^{w_u(u)} \cdot w_u'(u)$$

is close to one for $0 \le u < 1$. Where $g_u(u)$ is the modified uniform distribution.

Solving $w_u(u)$ in terms of b gives:

$$w_u(u) = \frac{b(b-1)}{b-u} + \left[(-(b-1)\ \right]$$

$$= \frac{b(b-1)}{b-u} + \frac{\left[(-(b-1)(b-u)\right]}{b-u}$$

$$= \frac{\left[\ b(b-1) - (b-1)(b-u)\right]}{b-u}$$

$$= \frac{\left[b^2 - b - (b^2 - bu - b + u)\right]}{b - u}$$

$$= \frac{\left[b^2 - b^2 - b + b + bu - u\right]}{b - u}$$

$$= \frac{\left[u\ (b - 1)\right]}{b - u}$$

Solving $w_U'(u)$ in terms of b gives

$$w_U'(u) = (b - 1)\ \frac{d}{du}\left[\ u(b - u)^{-1}\ \right]$$

$$= (b - 1)\left[\ (b - u)^{-1} + (-1)(-1)\ u\ (b - u)^{-2}\right]$$

$$= (b - 1)\left[(b - u)^{-1} + u\ (b - u)^{-2}\right]$$

$$= (b - 1)\left[\frac{1}{b - u} + \frac{u}{(b - u)^2}\ \right]$$

$$= (b - 1)\left[\ \frac{b - u + u}{(b - u)^2}\ \right]$$

$$= (b - 1)\left[\ \frac{b}{(b - u)^2}\ \right]$$

$$= \frac{b(b - 1)}{(b - u)^2}$$

Thus the form of $g_U(u)$ in terms of b is

$$g_U(u) = \ln(2)\ 2^{\left[\ \frac{\left[u\ (b - 1)\right]}{b - u}\ \right]}\left[\frac{b(b - 1)}{(b - u)^2}\right]$$

The value of b must be found such that p will be maximumized and r will be minimumized. To determine the conditions that will give a maximumized p the function $g_U(u)$ was graphed with several artbitary values of b. The resulting curve was a parabola opening downward. This implies that the maximum p will occur when $g_U(0) = g_U(1)$.

$$g_U(0) = \ln(2)\, 2^{\left[\frac{(0)(b-1)}{b-0}\right]} \left[\frac{b(b-1)}{(b-0)^2}\right]$$

$$= \ln(2)\, 2^0 \left[\frac{(b-1)}{b}\right]$$

$$= \frac{\ln(2)(b-1)}{b} \qquad (4.11)$$

$$g_U(1) = \ln(2)\, 2^{\left[\frac{(1)(b-1)}{b-1}\right]} \left[\frac{b(b-1)}{(b-1)^2}\right]$$

$$= \ln(2)\, 2^1 \left[\frac{b}{(b-1)}\right]$$

$$= 2\ln(2) \left[\frac{b}{(b-1)}\right]$$

$$= \frac{2\ln(2)\, b}{(b-1)} \qquad (4.12)$$

Setting Equations 4.11 and 4.12 equal to each other and solving for b gives

$$\frac{(b-1)\ln(2)}{b} = \frac{2\ln(2)\, b}{(b-1)}$$

$$(b-1)(b-1) = 2b^2$$

$$b^2 - 2b + 1 - 2b^2 = 0$$

$$b^2 - 2b^2 - 2b + 1 = 0$$

$$b^2 + 2b - 1 = 0$$

Using the quadratic formula to solve for b gives

$$b = \frac{-2 \pm \sqrt{(2)^2 - 4(-1)(1)}}{2}$$

$$= \frac{-2 \pm \sqrt{4 + 4}}{2}$$

$$= \frac{-2 \pm \sqrt{8}}{2}$$

$$= \frac{-2 \pm 2\sqrt{2}}{2} = -1 \pm \sqrt{2}$$

The value $b = -1 + \sqrt{2}$ must be discarded since $g_U(u)$ will generate negative values. Thus the values of the three parameters are

$$a = b(b - 1)$$
$$= (-1 - \sqrt{2})(-1 - \sqrt{2} - 1)$$
$$= (-1 - \sqrt{2})(-2 - \sqrt{2})$$
$$= (2 + 3\sqrt{2} + 2)$$
$$= (4 + 3\sqrt{2})$$

$$b = -1 - \sqrt{2}$$

$$c = -(b - 1)$$
$$= -(-1 - \sqrt{2} - 1)$$
$$= -(-2 - \sqrt{2})$$
$$= 2 + \sqrt{2}$$

The p value for $g_U(u)$ can be found by solving $g_U(0)$ since the curve is a parabola opening downward and $g_U(0) = g_U(1)$. From Equation (4.11)

$$g_U(0) = \frac{(b - 1)\ln(2)}{b}$$

$$= \frac{(-1 - \sqrt{2} - 1)\ln(2)}{(-1 - \sqrt{2})}$$

$$= \frac{(-2 - \sqrt{2}\,)\ln(2)}{(-1 - \sqrt{2}\,)}$$

$$= 0.9802581432$$

The value of r can be found by setting $g_U'(u) = 0$ and solving for u.

$$g_U'(u) = 0 = \frac{d}{du}\left[\ln(2)\ 2^{\left[\frac{a}{b-u}+c\right]}\left[\frac{a}{(b-u)^2}\right]\right]$$

$$= \ln(2)\left[2^{\left[\frac{a}{b-u}+c\right]}\{\ln(2)\}\left\{\frac{a}{(b-u)^2}\right\}^2 + 2^{\left[\frac{a}{b-u}+c\right]}\left\{\frac{2a}{(b-u)^3}\right\}\right]$$

$$= \left[\ln(2)\ 2^{\left[\frac{a}{b-u}+c\right]}\right]\left[\{\ln(2)\}\left\{\frac{a}{(b-u)^2}\right\}^2 + \left\{\frac{2a}{(b-u)^3}\right\}\right]$$

since $\ln(2)\ 2^{[\,W_U(u)\,]} > 0$ for all u it can be canceled out leaving

$$0 = \left[\{\ln(2)\}\left\{\frac{a}{(b-u)^2}\right\}^2 + \left\{\frac{2a}{(b-u)^3}\right\}\right]$$

$$= \frac{\ln(2)\ a^2}{(b-u)^4} + \frac{2a}{(b-u)^3}$$

$$= \frac{\ln(2)\ a^2}{(b-u)^4} + \frac{2a(b-u)}{(b-u)^4}$$

Multipling both sides by $(b - u)^4$

$$0 = \ln(2)\ a^2 + 2a(b-u)$$

$$= \ln(2)\ a^2 + 2ab - 2au$$

$$2au = \ln(2)\ a^2 + 2ab$$

$$u = \frac{\ln(2)\ a}{2} + b$$

So the maximum value r of $g_U(u)$ occurs when $u = b + \frac{\ln(2)\ a}{2}$ and that value is

$$r = g_U(b + \tfrac{\ln(2)\ a}{2}) = 1.010089592$$

The last constant $h = r - p$ is

$$h = 0.0298314385$$

The exact approximation method can now proceed using

$$w_U(u) = \frac{a}{b - U/p} + c = \frac{A}{B - U} + c$$

where

$$A = ap \quad \text{and} \quad B = bp.$$

The acceptance rejectance method will only be needed if $p > U$ or a probability of 1 in 50.65 = $1/(1 - p)$. To use the acceptance rejectance method generate two [0, 1) uniform deviates U and U' and accept $w_U(u)$ if $U' \leq (g_U(u) - p)/h$ or $U'h + p \leq g_U(u)$. The time needed to make each comparison of values can be shortened by simplifing $U'h + p \leq g_U(u)$ in the following manner.

$$g_U(u) = \ln(2)\, 2^{w_U(u)} \cdot w_U'(u)$$

$$\ln(2)\, 2^{w_U(u)} \cdot w_U'(u) = U'h + p$$

$$2^{w_U(u)} \geq \frac{U'h + p}{\ln(2)\, w_U'(u)}$$

$$\geq \frac{U'h + p}{\ln(2) \left[\frac{a}{(b - U)^2}\right]}$$

$$\geq (U'h + p) \left[\frac{(b - u)^2}{\ln(2)\, a}\right]$$

$$\geq \left[U' \frac{h}{\ln(2)\, a} + \frac{p}{\ln(2)\, a}\right] (b - u)^2$$

$$\geq (U'H + P)(b - u)^2$$

where

$$H = \frac{h}{\ln(2)\ a} \quad \text{and } P = \frac{p}{\ln(2)\ a}.$$

The expected number of uniform deviates used to generate one random variate is

$$1 + 2h = 1.059662877.$$

Table 4.2 gives the various run times for the different methods and different approximations to the inverse cdf. The different methods ran as expected with the exact approximation method running the fastest. Note that this distribution is over a closed interval where the exponential distribution is over a open interval.

Table 4.2

Method of generation	IBM PC	Zenith	AT&T 6300
Inversion	17.57	131.40	43.74
Acceptance Rejection	47.76	296.96	95.34
Exact Approximation 1[1]	20.18	109.54	40.11
Exact Approximation 2[2]	18.95	122.80	44.34
Exact Approximation 3[3]	17.33	81.71	31.43

[1] The exact approximation method using $c_1u^1 + c_2u^2$ as the approximation to the inverse cdf.

[2] The exact approximation method using $c_1u^1 + c_2u^2 + c_3u^3$ as the approximation to the inverse cdf.

[3] The exact approximation method using $\frac{a}{b-u} + c$ as the approximation to the inverse cdf.

Chapter V

CONCLUSIONS

The results of Ahrens and Dieter(1988) showed that the exact approximation method generates random variates at a considerablly faster rate than the inversion method. Using FORTRAN on a Seimens 7760 the exact approximation method is approximately 34% fasted than the inversion method. Using (360/370) Assembler on a Seimens 7760, the results were nearly the same as algorithm SA[1972] found by Ahrens and Dieter. Algorithm SA is also faster than the inversion method. This paper shows that the exact approximation method is not always more efficient then the inversion method. Several possibilities that could explain the discrepencies are, a different language was used, different uniform generator, and a different type of machine.

The algorithms were implemented using Borland Turbo C version 2.0. The uniform generator used in this paper is a linear congruential generator, while Ahrens and Dieter(1988) used a multiplicative congruential generator. The linear congruential generator will generate uniform deviates slower than the multiplicative congruential generator. This will cause the exact approximation method to run slower compared to the inversion method. The machines used were a IBM PC 8088 with an 8087 coprocessor, a Zenith with an 8088 processor, and a AT&T PC6310 without a 80287 coprocessor.

The ratio of uniform deviates used per random variate generated is one indicator of how efficiently a program will generate random variates. However, it isn't the only important measure of efficiency. For example, the function $c_1u + c_2u^2 + c_3u^3$ has a much smaller ratio of uniform deviates used to random variates generated than $c_1u + c_2u^2$. This is due to how closely $c_1u + c_2u^2 + c_3u^3$ approsimates the inverse cdf. The function $c_1u + c_2u^2$

doesn't approximate the inverse cdf as closely as $c_1u + c_2u^2 + c_3u^3$, but it is more efficient on the Zenith due to the ease of computation. This clearly indicates a need to limit the complexicity of the function used to approximate the inverse cdf.

In conclusion the exact approximation method is a method that can be more efficient than the inversion method but factors such as the language, hardware, and the complexicity of the function used to approximate the inverse cdf must all be considered.

BIBLIOGRAPHY

AHRENS, J. H., and DIETER, U., (Oct. 1972), "Computer Methods for Sampling from the Exponential and Normal Distributions," Communications of the ACM 15, 10, 872–882.

AHRENS, J. H., and DIETER, U., (Nov. 1988), "Efficient Table-Free Sampling Methods for the Exponential, Cauchy, and Normal Distributions," Communications of the ACM 31, 11, 1330–1337.

DEGROOT, M. H., (1975), Probability and Statistics, Addison-Wesley Publishing Company, Reading, Massachusetts.

GAUGHAN, E. D., (1975), Introduction to Analysis, Brooks/Cole Publishing Company, Monterey, California.

HOGG, R. V., and CRAIG A. T., (1965), Introduction to Mathematical Statistics, The Macmilllan Company, New York, New York.

MARSAGLIA, G., (1961), "Expressing a Random Variable in Terms of Uniform Random Variables," Annals of Mathematical Statistics 32, 894–898.

VON NEUMANN, J., (1951), "Various Techniques Used in Connection With Random Digits," U. S. National Bureau of Standards Applied Mathematics Series No. 12, 36–38.

APPENDIX A

Program listing of the exact approximation method for the exponential distribution function using $\frac{a}{b - u} + c$ as the approximation to the inverse cdf.

```
/*******************************************************************/
/*                                                                 */
/*      This is a program to test the speed of generation of random      */
/*      variates.  The pdf is exponential with lambda = 1.  The exact     */
/*      approximation method is being used with the approximate inverse  */
/*      cdf being  a/(b - u) + c.                                   */
/*                                                                 */
/*******************************************************************/

#include <stdio.h>;
#include <math.h>;
#include <dos.h>;
#include <conio.h>;

typedef struct
  {
  unsigned char hours;
  unsigned char minutes;
  unsigned char seconds;
  unsigned char hundred;
  } time_diff;

void gettime(struct time *timep);
void elapsed_time(struct time *timerec1,struct time *timerec2,time_diff
                                                      *time_elapsed);
double acceptance_rejection( long *seed);
float linear_congruential_generator(long *seed);
double approximate_cdf_inverse(double u);

/*******************************************************************/
/*                                                                 */
/*      Description of main function.                              */
/*                                                                 */
/*      The function main will control the number of random variates      */
/*      generated and determine whether the uniform deviate U is in the   */
/*      region [0,p).  If so, then accept the uniform deviate and use     */
/*      U/p to generate a random variate from the approximate inverse     */
/*      cdf w(u).  If U>p than the acceptance rejection method must be    */
/*      used to generate a random variate which will be exponentially     */
/*      distributed.                                               */
/*                                                                 */
/*      Description of constants and variables.                   */
/*                                                                 */
/*      repts        : the number of random variates to be generated.    */
/*      seed         : the initial seed value used in the linear         */
/*                        congruential generator.                  */
/*      P            : upper bound on the value of the uniform deviate.  */
/*      u            : [0, 1) uniform deviate.                     */
/*      z            : random variate from a truncated exponential       */
/*                     distribution.                               */
/*      naturallog   : the value of the natural log of 2.          */
/*      timerec1     : the time when started to generate random variates. */
/*      timerec2     : the time when finished generating random variates. */
/*      time_elapsed : the time used to generate the random variates.    */
/*      c                : constant for the approximate inverse cdf.         */
/*      g              : g = k*ln(2) + c  where k is the number of leading    */
/*                       bit zeros of a [0, 1) uniform deviate.              */
/*                                                                           */
/****************************************************************************/

main()
  {
  struct time timerec1;
  struct time timerec2;
  time_diff time_elapsed;
  int repts;
  double naturallog = log(2);
  double P = 0.9802581434;
  double c =-1.6734053240;
  double g,u,z;
  long *seed;
  seed = (long *) malloc(sizeof(long));
  clrscr();
  printf("Input the value of seed ==> ");
  scanf("%d",seed);
  gettime(&timerec1);
  for(repts = 1; repts<=10000; repts++)         /* random variate generation */
    {                                               /* loop                  */
    g = c;
    u = linear_congruential_generator(seed);
    if(!(u = 0.0))
      {
      u = u + u;
      while(1.0 >= u)                                /* loop to find k*ln(2) */
        {
        u = u + u;
        g = g + naturallog;
        }
      u --;
      if (u<P)                                          /* generate z using */
        z = approximate_cdf_inverse(u);             /* uniform distribution */
      else
        z = acceptance_rejection(seed);   /* or acceptance rejection method */
      z = z + g;                                   /* wfunct = z + k*ln(2) */
      }
   else
     {
     repts--;
     count--;
     }
  }
  gettime(&timerec2);
  elapsed_time(&timerec1,&timerec2,&time_elapsed);
  printf("\n\n elapsed time for %d trials:",repts-1);
  printf("\n hours minutes seconds hundredths");
  printf("\n %3d ", time_elapsed.hours);
  printf("%7d ", time_elapsed.minutes);
  printf("%7d ", time_elapsed.seconds);
```

```
 printf("%8d ", time_elapsed.hundred);
 }

/*****************************************************************/
/*                                                              */
/*      Purpose of the function.                                */
/*                                                              */
/*      approximate_cdf_inverse is a function that will approximate the  */
/*      inverse function of the desired cdf.                    */
/*                                                              */
/*      Description of constants and variables.                 */
/*                                                              */
/*      A : A = a*p      :constant used to determine the values of w(u1)  */
/*                          and w'(u1)                          */
/*      B : B = b*p      :constant used to determine the values of w(u1)  */
/*                          and (w'(u1)                         */
/*      wfunct        : The value of the approximate inverse cdf function.  */
/*                                                              */
/*****************************************************************/

double approximate_cdf_inverse(double u)
 {
 double A = 5.6005707570;
 double B = 3.3468106481;
 double zfunct;
 zfunct = A/(B - u);
 return(zfunct);
 }

/*****************************************************************/
/*                                                              */
/*      Purpose of the function.                                */
/*                                                              */
/*      accceptance_rejection is a function that will use the acceptance  */
/*      rejection method to determine whether a uniform deviate is  */
/*      below the curve of a function.  In this case the funotion is  */
/*      defined to be:                                          */
/*                                                              */
/*     g(u1) = f[w(u1)]*w'(u1).                                 */
/*                                                              */
/*      Description of constants and variables.                 */
/*                                                              */
/*      done              : loop control variable used to determine when  */
/*                          (u2*H + P)(b - u1)^2 <= exp(-w(u1)).  */
/*      a,b,c            : constants to determine the values of w(u1)  */
/*                          and w'(u1).                         */
/*      P : P = p/(2*a) : upper bound of the [0,1)-uniform deviates  */
/*                              divided by 2*a to reduce computations.  */
/*      H : H = h/(2*a) : height of the rectangle that the acceptance  */
/*                               rejection method will be performed in divided */
/*                               by 2*a to reduce the number of computations. */
/*      u1,u2            : [0,1)-uniform deviates.              */
/*      wfunct            : value of the approximate inverse cdf.  */
/*      wprimefunct       : value of the derivative of the approximate  */
/*                              inverse cdf.                                */
/*      ffunct          : the pdf of the random variates that are          */
/*                            generated.                                    */
/*      gfunct          : function that will adjusts the approximate       */
/*                          inverse cdf so it will give random variates    */
/*                          that have the desired distribution.             */
/*                                                                          */
/***************************************************************************/

double acceptance_rejection(long int *seed)
  {
  int done = 0;
  double a = 5.7133631526454228;
  double b = 3.4142135623730950;
  double c =-1.6734053240284925;
  double H = 0.0026106723602095;
  double P = 0.0857864376299050;
  double u1,u2;
  double wfunct,wprimefunct,ffunct,y;
  while (!(done))
    {
    u1 = linear_congruential_generator(seed);
    u2 = linear_congruential_generator(seed);
    wfunct = a/(b - u1);
    wprimefunct = (b - u1)*(b - u1);
    y = (u2*H + P)*wprimefunct;
    ffunct = exp(-(wfunct + c));
    if (y <= ffunct)
      done = 1;                     /* random variate is below g(u) */
    }
  return(wfunct);
  }

/***************************************************************************/
/*                                                                          */
/*      Purpose of function.                                                */
/*                                                                          */
/*      linear_congruential_generator is a function that will generate     */
/*      a [0,1)-uniform deviate.                                            */
/*                                                                          */
/*      Description of constants and variables.                             */
/*      A           : multiplier                                            */
/*      C           : increment                                             */
/*      M           : modulus                                               */
/*                                                                          */
/***************************************************************************/

float linear_congruential_generator(long int *seed)
  {
  long int A = 25173;
  long int C = 13849;
  long int M = 65536;
  *seed = (A*(*seed) + C) % M;
  return((float)*seed/(float)M);
```

```
 }

/**************************************************************************/
/*                                                                      */
/*      Purpose of function.                                            */
/*                                                                      */
/*      elapsed_time is a function that will determine the time it takes    */
/*      for the program to generate a known number of random variables. */
/*                                                                      */
/*      Description of constants and variables.                         */
/*                                                                      */
/*      timerec1     : Record that contains the time when random variate  */
/*                        generation begin.                             */
/*      timerec2     : Record that contains the time when random variate  */
/*                        generation ended.                             */
/*      time_elapsed : Record that contains the time taken to generate    */
/*                          the random variates.                        */
/*                                                                      */
/*      fields of the records.                                          */
/*      hour          : Time in hours of random variate generation.       */
/*      min           : Time in minutes of random variate generation.     */
/*      sec           : Time in seconds of random variate generation.     */
/*      hund          : Time in hundredths of a second of                 */
/*                        random variate generation.                    */
/*                                                                      */
/**************************************************************************/

void elapsed_time(struct time *timerec1,struct time *timerec2,time_dif
                                                              *time_elapsed)
  {
  if (timerec2->ti_hund >= timerec1->ti_hund)
    time_elapsed->hundred = timerec2->ti_hund - timerec1->ti_hund;
  else
    {
    timerec2->ti_sec --;
    time_elapsed->hundred = 100 + timerec2->ti_hund - timerec1->ti_hund;
    }
  if (timerec2->ti_sec >= timerec1->ti_sec)
    time_elapsed->seconds = timerec2->ti_sec - timerec1->ti_sec;
  else
    {
    timerec2->ti_min --;
    time_elapsed->seconds = 60 + timerec2->ti_sec - timerec1->ti_sec;
    }
  if (timerec2->ti_min >= timerec1->ti_min)
    time_elapsed->minutes = timerec2->ti_min - timerec1->ti_min;
  else
    {
    timerec2->ti_hour --;
    time_elapsed->minutes = 60 + timerec2->ti_min - timerec1->ti_min;
    }
  if (timerec2->ti_hour >= timerec1->ti_hour)
    time_elapsed->hours = timerec2->ti_hour - timerec1->ti_hour;
  else
  time_elapsed->hours = 24 + timerec2->ti_hour - timerec1->ti_hour;
}
```

APPENDIX B

Program listing of the exact approximation method for the exponential distribution function using $c_1u + c_2u^2 + c_3u^3$ as the approximation to the inverse cdf.

```
/*************************************************************************/
/*                                                                       */
/*      This is a program to test the speed of generation of random      */
/*      variates. The pdf is exponential with lambda = 1. The exact      */
/*      approximation method is being used with the approximate inverse  */
/*      cdf being c1u + c2u ² + c3u³.                                     */
/*                                                                       */
/*************************************************************************/

#include <stdio.h>;
#include <math.h>;
#include <dos.h>;
#include <conio.h>;

typedef struct
  {
  unsigned char hours;
  unsigned char minutes;
  unsigned char seconds;
  unsigned char hundred;
  } time_diff;

void gettime(struct time *timep);
void elapsed_time(struct time *timerec1,struct time *timerec2,time_diff
                                                          *time_elapsed);
double acceptance_rejection( long *seed);
float linear_congruential_generator(long *seed);
double approximate_cdf_inverse(double u);

/*************************************************************************/
/*                                                                       */
/*      Description of main function.                                    */
/*                                                                       */
/*      The function main will control the number of random variates     */
/*      generated and determine whether the uniform deviate U is in the  */
/*      region [0,p). If so, then accept the uniform deviate and use     */
/*      U/p to generate a random variate from the approximate inverse    */
/*      cdf w(u). If U>p than the acceptance rejection method must be    */
/*      used to generate a random variate which will be exponentially    */
/*      distributed.                                                     */
/*                                                                       */
/*      Description of constants and variables.                          */
/*                                                                       */
/*      repts        : the number of random variates to be generated.    */
/*      seed         : the initial seed value used in the linear         */
/*                        congruential generator.                        */
/*      P            : upper bound on the value of the uniform deviate.  */
/*      u            : [0, 1) uniform deviate.                           */
/*      z            : random variate from a truncated exponential       */
/*                     distribution.                                     */
/*      naturallog   : the value of the natural log of 2.                */
/*      timerec1     : the time when started to generate random variates. */
/*      timerec2     : the time when finished generating random variates. */
/*      time_elapsed : the time used to generate the random variates.    */
/*       c                  : constant for the approximate inverse cdf.          */
/*       g             : g = k*ln(2) + c  where k is the number of leading       */
/*                      bit zeros of a [0, 1) uniform deviate.                   */
/*                                                                               */
/*******************************************************************************/
main()
  {
  struct time timerec1;
  struct time timerec2;
  time_diff time_elapsed;
  int repts;
  double naturallog = log(2);
  double P = 0.9829974429;
  double g,u,z;
  long *seed;
  seed = (long *) malloc(sizeof(long));
  clrscr();
  printf("Input the value of seed ==> ");
  scanf("%d",seed);
  gettime(&timerec1);
  for(repts = 1; repts<=10000; repts++)          /* random variate generation */
    {                                                /* loop                     */
    g = 0.0;
    u = linear_congruential_generator(seed);
    if(u >= 0.0)
      {
      u = u + u;
      while(1.0 >= u)                                   /* loop to find k*ln(2) */
        {
        u = u + u;
        g = g + naturallog;
        }
      u --;
      if (u<P)                                           /* generate z using */
        z = approximate_cdf_inverse(u);              /* uniform distribution */
      else
        z = acceptance_rejection(seed);   /* or acceptance rejection method */
      z = z + g;                                      /* wfunct = z + k*ln(2) */
      }
     else
      {
      repts--;
      count--;
      }
  }
  gettime(&timerec2);
  elapsed_time(&timerec1,&timerec2,&time_elapsed);
  printf("\n\n elapsed time for %d trials.",repts-1);
  printf("\n hours minutes seconds hundredths");
  printf("\n %3d ", time_elapsed.hours);
  printf("%7d ", time_elapsed.minutes);
  printf("%7d ", time_elapsed.seconds);
  printf("%8d ", time_elapsed.hundred);
  }
```

```
/****************************************************************/
/*                                                              */
/*      Purpose of the function.                                */
/*                                                              */
/*      approximate_cdf_inverse is a function that will approximate the  */
/*      inverse function of the desired cdf.                    */
/*                                                              */
/*      Description of constants and variables.                 */
/*                                                              */
/*      C1 : C1 = c1/p      :constant used to determine the values of w(u1) */
/*                               and w'(u1)                     */
/*      C2 : C2 = c2/(p^2) :constant used to determine the values of w(u1)  */
/*                               and (w'(u1)                    */
/*      C3 : C3 = c3/(p^3) :constant used to determinet the values of w(u1) */
/*                               and (w'(u1))                   */
/*      wfunct      : The value of the approximate inverse cdf function.   */
/*                                                              */
/****************************************************************/

double approximate_cdf_inverse(double u)
  {
  double C1 = 0.5086483221;
  double C2 = 0.0822134526;
  double C3 = 0.1197084585;
  double zfunct;
  zfunct = C1*u + C2*u*u + C3*u*u*u;
  return(zfunct);
  }

/****************************************************************/
/*                                                              */
/*      Purpose of the function.                                */
/*                                                              */
/*      accceptance_rejection is a function that will use the acceptance  */
/*      rejection method to determine whether a uniform deviate is        */
/*      below the curve of a function.  In this case the function is      */
/*      defined to be:                                          */
/*                                                              */
/*    g(u1) = f[w(u1)]*w'(u1).                                  */
/*                                                              */
/*      Description of constants and variables.                 */
/*                                                              */
/*      done            : loop control variable used to determine when    */
/*                        (u2*H + P)(b - u1)^2 <= exp(-w(u1)).  */
/*      c1,c2,c3        : constants to determine the values of w(u1)      */
/*                        and w'(u1).                           */
/*      P : P = p/(ln(2)) : upper bound of the [0,1)-uniform deviates     */
/*                            divided by 2*a to reduce computations.      */
/*      H : H = h/(ln(2)) : height of the rectangle that the acceptance   */
/*                            rejection method will be performed in divided */
/*                            by 2*a to reduce the number of computations. */
/*      u1,u2           : [0,1)-uniform deviates.               */
/*      wfunct          : value of the approximate inverse cdf. */
/*      wprimefunct     : value of the derivative of the approximate        */
/*                          inverse cdf.                                     */
/*      y : y = u2*H + P:value used to test for acceptance reuection        */
/*                        generated.                                         */
/*      gfunct          : function that will adjusts the approximate        */
/*                        inverse cdf so it will give random variates       */
/*                        that have the desired distribution.               */
/*                                                                           */
/*************************************************************************/

double acceptance_rejection(long int *seed)
  {
  int done = 0;
  double c1= 0.5;
  double c2= 0.0794415416;
  double c3 = 0.1137056389;
  double H = 0.0167124520;
  double P = 0.4914987215;
  double u1,u2;
  double wfunct,wprimefunct,ffunct,y;
  while (!(done))
    {
    u1 = linear_congruential_generator(seed);
    u2 = linear_congruential_generator(seed);
    wfunct = c1*u1 + c2*u1*u1 + c3*u1*u1*u1;
    wprimefunct = c1 + 2*c2*u1 + 3*c3*u1*u1;
    y = (u2*H + P);
    gfunct = exp(-(wfunct ))*wprimefunct;
    if (y <= gfunct)
      done = 1;                    /* random variate is below g(u) */
    }
  return(wfunct);
  }

/*************************************************************************/
/*                                                                           */
/*      Purpose of function.                                                 */
/*                                                                           */
/*      linear_congruential_generator is a function that will generate      */
/*      a [0,1)-uniform deviate.                                             */
/*                                                                           */
/*      Description of constants and variables.                             */
/*      A           : multiplier                                             */
/*      C           : increment                                              */
/*      M           : modulus                                                */
/*                                                                           */
/*************************************************************************/

float linear_congruential_generator(long int *seed)
  {
  long int A = 25173;
  long int C = 13849;
  long int M = 65536;
  *seed = (A*(*seed) + C) % M;
  return((float)*seed/(float)M);
  }

/*****************************************************************/
/*                                                                         */
/*      Purpose of function.                                               */
/*                                                                         */
/*      elapsed_time is a function that will determine the time it takes    */
/*      for the program to generate a known number of random variables.  */
/*                                                                         */
/*      Description of constants and variables.                            */
/*                                                                         */
/*      timerec1     : Record that contains the time when random variate   */
/*                        generation begin.                                */
/*      timerec2     : Record that contains the time when random variate   */
/*                        generation ended.                                */
/*      time_elapsed : Record that contains the time taken to generate     */
/*                          the random variates.                           */
/*                                                                         */
/*      fields of the records.                                             */
/*      hour          : Time in hours of random variate generation.        */
/*      min          : Time in minutes of random variate generation.       */
/*      sec          : Time in seconds of random variate generation.       */
/*      hund          : Time in hundredths of a second of                   */
/*                        random variate generation.                       */
/*                                                                         */
/*****************************************************************/

void elapsed_time(struct time *timerec1,struct time *timerec2,time_dif
                                                        *time_elapsed)
  {
  if (timerec2->ti_hund >= timerec1->ti_hund)
    time_elapsed->hundred = timerec2->ti_hund - timerec1->ti_hund;
  else
    {
    timerec2->ti_sec --;
    time_elapsed->hundred = 100 + timerec2->ti_hund - timerec1->ti_hund;
    }
  if (timerec2->ti_sec >= timerec1->ti_sec)
    time_elapsed->seconds = timerec2->ti_sec - timerec1->ti_sec;
  else
    {
    timerec2->ti_min --;
    time_elapsed->seconds = 60 + timerec2->ti_sec - timerec1->ti_sec;
    }
  if (timerec2->ti_min >= timerec1->ti_min)
    time_elapsed->minutes = timerec2->ti_min - timerec1->ti_min;
  else
    {
    timerec2->ti_hour --;
    time_elapsed->minutes = 60 + timerec2->ti_min - timerec1->ti_min;
    }
  if (timerec2->ti_hour >= timerec1->ti_hour)
    time_elapsed->hours = timerec2->ti_hour - timerec1->ti_hour;
```

```
else
  time_elapsed->hours = 24 + timerec2->ti_hour - timerec1->ti_hour;
}
```

APPENDIX C

Program listing of a program that generates random variates using the inversion method. The function to invert is the exponential function.

```
/*****************************************************************/
/*                                                               */
/*      This is a program to test the speed of generation of random    */
/*      variates.  The pdf is e^(-y).  This program uses inversion.     */
/*                                                               */
/*****************************************************************/

#include <stdio.h>;
#include <math.h>;
#include <dos.h>;
#include <conio.h>;

typedef struct
  {
  unsigned char hours;
  unsigned char minutes;
  unsigned char seconds;
  unsigned char hundred;
  } time_diff;

void gettime(struct time *timep);
void elapsed_time(struct time *timerec1,struct time *timerec2,time_diff
                                                          *time_elapsed);
double acceptance_rejection( long *seed);
float linear_congruential_generator(long *seed);
double approximate_cdf_inverse(double u);

/*****************************************************************/
/*                                                               */
/*      Description of main function.                            */
/*                                                               */
/*      The function main will control the number of random variates  */
/*      generated.  It uses inversion to generate a random variate z   */
/*      from an exponential function.                            */
/*                                                               */
/*      Description of constants and variables.                  */
/*                                                               */
/*      repts        : the number of random variates to be generated.  */
/*      seed         : the initial seed value used in the linear       */
/*                       congruential generator.                 */
/*      u           : [0, 1) uniform deviate.                     */
/*      z           : random variate from an exponential distri-  */
/*                    bution.                                     */
/*      timerec1     : the time when started to generate random variates. */
/*      timerec2     : the time when finished generating random variates. */
/*      time_elapsed : the time used to generate the random variates.   */
/*                                                               */
/*****************************************************************/

main()
  {
  struct time timerec1;
  struct time timerec2;
  time_diff time_elapsed;
  int repts;
  double u,z;
  long *seed;
  seed = (long *) malloc(sizeof(long));
  clrscr();
  printf("Input the value of seed ==> ");
  scanf("%d",seed);
  gettime(&timerec1);
  for(repts = 1; repts<=10000; repts++)        /* random variate generation */
    {                                              /* loop                    */
    u = linear_congruential_generator(seed);
    z = -log(u);
     }
  }
  gettime(&timerec2);
  elapsed_time(&timerec1,&timerec2,&time_elapsed);
  printf("\n\n elapsed time for %d trials.",repts-1);
  printf("\n hours minutes seconds hundredths");
  printf("\n %3d ", time_elapsed.hours);
  printf("%7d ", time_elapsed.minutes);
  printf("%7d ", time_elapsed.seconds);
  printf("%8d ", time_elapsed.hundred);
  }

/*******************************************************************/
/*                                                                 */
/*      Purpose of function.                                       */
/*                                                                 */
/*      linear_congruential_generator is a function that will generate  */
/*      a [0,1)-uniform deviate.                                   */
/*                                                                 */
/*      Description of constants and variables.                   */
/*      A           : multiplier                                   */
/*      C           : increment                                    */
/*      M           : modulus                                      */
/*                                                                 */
/*******************************************************************/

float linear_congruential_generator(long int *seed)
  {
  long int A = 25173;
  long int C = 13849;
  long int M = 65536;
  *seed = (A*(*seed) + C) % M;
  return((float)*seed/(float)M);
  }

/*******************************************************************/
/*                                                                 */
/*      Purpose of function.                                       */
/*                                                                 */
/*      elapsed_time is a function that will determine the time it takes  */
/*      for the program to generate a known number of random variables.  */
/*                                                                 */
/*      Description of constants and variables.                            */
/*                                                                         */
/*      timerec1     : Record that contains the time when random variate   */
/*                        generation begin.                                */
/*      timerec2     : Record that contains the time when random variate   */
/*                        generation ended.                                */
/*      time_elapsed : Record that contains the time taken to generate     */
/*                        the random variates.                             */
/*                                                                         */
/*      fields of the records.                                             */
/*      hour         : Time in hours of random variate generation.         */
/*      min          : Time in minutes of random variate generation.       */
/*      sec          : Time in seconds of random variate generation.       */
/*      hund         : Time in hundredths of a second of                   */
/*                      random variate generation.                         */
/*                                                                         */
/**************************************************************************/

void elapsed_time(struct time *timerec1,struct time *timerec2,time_dif
                                                           *time_elapsed)
  {
  if (timerec2->ti_hund >= timerec1->ti_hund)
    time_elapsed->hundred = timerec2->ti_hund - timerec1->ti_hund;
  else
    {
    timerec2->ti_sec --;
    time_elapsed->hundred = 100 + timerec2->ti_hund - timerec1->ti_hund;
    }
  if (timerec2->ti_sec >= timerec1->ti_sec)
    time_elapsed->seconds = timerec2->ti_sec - timerec1->ti_sec;
  else
    {
    timerec2->ti_min --;
    time_elapsed->seconds = 60 + timerec2->ti_sec - timerec1->ti_sec;
    }
  if (timerec2->ti_min >= timerec1->ti_min)
    time_elapsed->minutes = timerec2->ti_min - timerec1->ti_min;
  else
    {
    timerec2->ti_hour --;
    time_elapsed->minutes = 60 + timerec2->ti_min - timerec1->ti_min;
    }
  if (timerec2->ti_hour >= timerec1->ti_hour)
    time_elapsed->hours = timerec2->ti_hour - timerec1->ti_hour;
  else
    time_elapsed->hours = 24 + timerec2->ti_hour - timerec1->ti_hour;
  }
```

APPENDIX D

Program listing of a program that generates random variates from the exponential distribution using the acceptance rejection method.

```
/*****************************************************************/
/*                                                               */
/*      This is a program to test the speed of generation of random      */
/*      variates.  The pdf is exponential and the acceptance            */
/*      rejection method is being used.                                 */
/*                                                               */
/*****************************************************************/

#include <stdio.h>;
#include <math.h>;
#include <dos.h>;
#include <conio.h>;

typedef struct
  {
  unsigned char hours;
  unsigned char minutes;
  unsigned char seconds;
  unsigned char hundred;
  } time_diff;

void gettime(struct time *timep);
void elapsed_time(struct time *timerec1,struct time *timerec2,time_diff
                                                        *time_elapsed);
double acceptance_rejection( long *seed);
float linear_congruential_generator(long *seed);
double approximate_cdf_inverse(double u);

/*****************************************************************/
/*                                                               */
/*      Description of main function.                                   */
/*                                                               */
/*      The function main will control the number of random variates      */
/*      generated. It a will generate a [0, 500) uniform deviate u and use */
/*     and set z = g(u).  Main will also generate a [0, 1) uniform deviate  */
/*     y and use it to compare y <= z.  If y <= z then accept it.   Else   */
/*      reject u.                                                       */
/*                                                               */
/*      Description of constants and variables.                         */
/*                                                               */
/*      repts       : the number of random variates to be generated.    */
/*      seed        : the initial seed value used in the linear        */
/*                     congruential generator.                          */
/*      u             :  [0, 50001) uniform deviate.                    */
/*      z             :  g(u)                                           */
/*      y             :  value between 0 and max g(u).                  */
/*      timerec1     : the time when started to generate random variates. */
/*      timerec2     : the time when finished generating random variates. */
/*      time_elapsed : the time used to generate the random variates.    */
/*                                                               */
/*****************************************************************/

main()
  {
```

```
  struct time timerec1;
  struct time timerec2;
  time_diff time_elapsed;
  int repts;
  double u,y,z;
  long *seed;
  seed = (long *) malloc(sizeof(long));
  clrscr();
  printf("Input the value of seed ==> ");
  scanf("%d",seed);
  gettime(&timerec1);
  for(repts = 1; repts<=10000; repts++)        /* random variate generation */
    {                                              /* loop                  */
    int done = 0;
    while(!(done))
      {
      u = linear_congruential_generator(seed);
      u = u*500;
      y = linear_congruential_generator(seed);
      z = exp(-u);
      if (y <= z)
        done=1;
      }
  }
  gettime(&timerec2);
  elapsed_time(&timerec1,&timerec2,&time_elapsed);
  printf("\n\n elapsed time for %d trials.",repts-1);
  printf("\n hours minutes seconds hundredths");
  printf("\n %3d ", time_elapsed.hours);
  printf("%7d ", time_elapsed.minutes);
  printf("%7d ", time_elapsed.seconds);
  printf("%8d ", time_elapsed.hundred);
  }

/*******************************************************************/
/*                                                                 */
/*       Purpose of function.                                      */
/*                                                                 */
/*       linear_congruential_generator is a function that will generate */
/*       a [0,1)-uniform deviate.                                  */
/*                                                                 */
/*       Description of constants and variables.                   */
/*       A          : multiplier                                   */
/*       C          : increment                                    */
/*       M          : modulus                                      */
/*                                                                 */
/*******************************************************************/

float linear_congruential_generator(long int *seed)
  {
  long int A = 25173;
  long int C = 13849;
  long int M = 65536;
  *seed = (A*(*seed) + C) % M;
  return((float)*seed/(float)M);
  }

/*************************************************************/
/*                                                           */
/*      Purpose of function.                                 */
/*                                                           */
/*      elapsed_time is a function that will determine the time it takes   */
/*      for the program to generate a known number of random variables.  */
/*                                                           */
/*      Description of constants and variables.              */
/*                                                           */
/*      timerec1     : Record that contains the time when random variate  */
/*                        generation begin.                  */
/*      timerec2     : Record that contains the time when random variate  */
/*                        generation ended.                  */
/*      time_elapsed : Record that contains the time taken to generate    */
/*                          the random variates.             */
/*                                                           */
/*      fields of the records.                               */
/*      hour        : Time in hours of random variate generation.        */
/*      min         : Time in minutes of random variate generation.      */
/*      sec         : Time in seconds of random variate generation.      */
/*      hund        : Time in hundredths of a second of                  */
/*                      random variate generation.           */
/*                                                           */
/*************************************************************/

void elapsed_time(struct time *timerec1,struct time *timerec2,time_dif
                                                          *time_elapsed)
  {
  if (timerec2->ti_hund >= timerec1->ti_hund)
    time_elapsed->hundred = timerec2->ti_hund - timerec1->ti_hund;
  else
    {
    timerec2->ti_sec --;
    time_elapsed->hundred = 100 + timerec2->ti_hund - timerec1->ti_hund;
    }
  if (timerec2->ti_sec >= timerec1->ti_sec)
    time_elapsed->seconds = timerec2->ti_sec - timerec1->ti_sec;
  else
    {
    timerec2->ti_min --;
    time_elapsed->seconds = 60 + timerec2->ti_sec - timerec1->ti_sec;
    }
  if (timerec2->ti_min >= timerec1->ti_min)
    time_elapsed->minutes = timerec2->ti_min - timerec1->ti_min;
  else
    {
    timerec2->ti_hour --;
    time_elapsed->minutes = 60 + timerec2->ti_min - timerec1->ti_min;
    }
  if (timerec2->ti_hour >= timerec1->ti_hour)
    time_elapsed->hours = timerec2->ti_hour - timerec1->ti_hour;
```

```
else
  time_elapsed->hours = 24 + timerec2->ti_hour - timerec1->ti_hour;
}
```