An Efficient Method of Generating Infinite-Precision
Exponential Random Variables

by

Herman Rubin¹
Purdue University

Technical Report #86-39

Department of Statistics
Purdue University

1986

¹ Research supported by the National Science Foundation under Grant DMS-8401996.
An Efficient Method of Generating Infinite-Precision
Exponential Random Variables

by
Herman Rubin

ABSTRACT

We give a method of generating random variables which, assuming we have perfectly
random bits available, are exactly exponential with mean 1. The procedure produces the
integer part and zeros out some of the bits in the fractional part, all other bits in the
fractional part being independent random bits. The procedure uses an expected number
of 4.383154 bits to obtain the integer part and the mask, whereas the information in these
quantities is 3.647347 bits. Unfortunately, the architecture of computers makes it difficult
to carry out the operations efficiently because of bookkeeping. The procedures are as easy
to implement on microcomputers as on large computers, although certain instructions not
found in high level languages may speed up the implementation on some machines. Vector
machines which have good merge instructions (combining vectors by using a bit vector to
decide from which vector to take the next element of that vector, not the element in the
corresponding place) and other cute instructions, such as the CYBER 205, can vectorize
the procedure.

Introduction. The procedure is based on the fact that the waiting times of a Poisson
process with mean 1 are exponential with mean 1. Thus if we let $N_i$ be independent
Poisson random variables with mean 1, $K$ is the number of initial 0’s in the $N$ sequence,
and $J = N_{K+1}$, the exponential random variable is $E = K + \min(U_1, \ldots, U_J)$. The way
that the minimum is formed is to find the first place where the uniform random variables
differ, put a 0 in that place, and proceed using only the uniform random variables which
are 0 there, until we get to one random variable only. Of course, we need only get the right
distribution of the locations of the 0’s, and this is what we do. We use the fact that, in
generating Poisson random variables with mean 1, we find it convenient to also generate
geometric random variables $g_4$ with parameter $1/4$ if $J \geq 3, J \neq 5$, and geometric random
variables $g_{16}$ with parameter $1/16$ if $J \geq 5$. Very little of this extra information is not used
in the process. Now let us give the procedures for obtaining the minimum of $j$ uniform
random variables by masking out bits in one uniform random variable. Initially we assume
the mask bit $b$ is in the place to the left of the binary point. We shall use $G$ for a new
geometric (1/2) random variable each time, and $B$ for a new 50-50 test each time.

If $j = 1$, exit.

If $j = 2$, shift $b$ right $G$ places; mask out $b$; exit.

If $j = 3$, shift $b$ right $g_4$ places; mask out $b$; $B; j = 1$, $j = 2$. (The last step can also
be done as shift $b$ right $G - 1$ places; mask out $b$; exit. This is what we have programmed,
as it uses less tests, but which procedure to use is optional.)

If $j = 4$, we use both $g_4$ and $G$. The probability that $g_4 = i, G = k$ is $\frac{3}{4} \times 2^{-2i-k}$,
so if $i < k$, this is just the probability that the first mask bit is $i$, the new $j$ is 2, and we
have \( k - i \) as an independent geometric \((1/2)\) random variable to continue masking, while if \( i \geq k \), the first mask bit is \( k \), the new \( j \) is equally likely to be 1 or 3, and if it is 3 we use \( i - k + 1 \) as our new \( g4 \).

If \( j = 5 \), we may use \( g16 \) as the first mask bit. The probabilities for the new \( j \) are as 1:2:2:1, which can be obtained by the use of \( B \) and a 2:1 test. The most efficient use of bits to obtain one 2:1 test yields a \( g4 \) as a free byproduct.

If \( j = 6 \), we may use \( \min(G, g16) \) for the first mask bit. If they are unequal, the difference can again be used as a random variable of the same type, and we shall use that notation. If \( g16 > G \), the new \( j \) is 5. If they are equal, \( B : j = 2, j = 4 \). If \( G > g16 \), let the new \( G \) be \( 4k - h, h = 0,1,2,3 \). Now if we set the new \( j \) to be 1 if \( h \leq 1, 3 \) if \( h = 3 \), and if \( h = 2 \), choose 3 or 5 with equal probabilities, and if the new \( j \) is 5, set \( g4 = k \). This gives the correct probabilities.

If \( j = 7 \), let \( G = 6k - h, 0 \leq h < 6 \). Then \( k \) is the first mask bit, if \( h \in \{0,1,5\} \) we set the new \( j \) to be 3 or 4, and otherwise if \( m = g4 - 1 \) is 0 we set the new \( j \) to be 2 or 5 (neither of which uses \( g4 \)), and otherwise the new \( j \) is 1 or 6, and if \( j = 6, g4 = m \).

For \( j \geq 8 \), we have just used brute force.

We do not claim that these procedures are optimal, but the amount of improvement cannot be great unless random information is recycled. For example, if \( j = 4 \) with probability \( 2/7 \), for \( j = 5 \) with probability \( 1/2 \), for \( j = 6 \) with probability \( 27/82 \), and for \( j = 7 \) with probability \( 1/16 \), there is an unused \( g4 \), and for \( j = 6 \) with probability \( 25/83 \), and for \( j = 7 \) with probability \( 7/5 \), \( g16 \) is never used. Certainly for \( j \geq 7 \), better procedures can be used both for generating \( j \) and for finding the minimum of \( j \) uniform random variables. However, the wastage in work is only about 0.02 bits per exponential random variable generated, so that it does not seem worthwhile to do anything about it.

The other place where a better procedure could be generated is to use a more efficient means of generating Poisson random variables. The improvement possible here is about 0.25 bits per Poisson generated, or 0.39 per exponential random variable generated. Most of this inefficiency is in the fact that we only use the two least significant bits of the first geometric random variable in the process. The unused information is a \( g16 \), which has nearly 0.36 bits of information. These procedures are also difficult to implement. The rest of the inefficiency is in the fact that there are different paths to a given mask, and these are almost impossible to do much about.

In the procedure the Poisson random variables are not formed unless they are at least 8, but only become internal states of the process.

What we do is in each cycle is either to exit without generating anything or to generate independent random variables \( X \) and \( Y \) in \( \{U, 0, 1, \ldots\} \), where \( U \) means undefined. If the random variable is at least 3 and not 5, an geometric random variable \( g4 \) with \( p = 1/4 \) is also generated, and if the random variable is at least 5, a \( g16 \) with \( p = 1/16 \) is generated; these are essentially free, and may be regenerated in the process as needed. We never separate
0 and 1 until we finish the use of that particular Poisson variable. As a preliminary step we generate random variables \( S \) and \( T \) in the set \( \{-1,1,3,4,5,\ldots\} \), \( T \neq 5 \), with 
\[ P(S = s, T = t) = 2^{-s-t}/7.5, \]
and just abort the cycle with probability \( 1/320 \). It might pay in the case of this abort to generate \( S \) only with probability 11/12 in this case. That this can be useful is due to the fact that there is already a 2:1 division. We will call this the alternative method in what follows. We then use \( S(T) \) to generate \( X(Y) \), with 
\[ P(X = x, T = t) = 2^{-t}/(7.5x!), \]
and similarly in the other case. Thus the probability that \( X = m \) is \( \frac{29}{80m!} \) and the probability that \( Y = m \) is \( \frac{11}{30m!} \). The "t" blocks in "work" get the values of \( X \) and/or \( Y \) if they exceed 3, although a value less than 8 does not appear except as a state. Most of the inefficiency of the procedure is in the generation of random variables with probabilities proportional to \( 2^{-i}, i = -1,1,3,4,6,7,\ldots \), but the author does not know of any ways to do this better which are not extremely complicated.

**Procedure.** We now give the detailed procedure for a sequential computer. On some computers, one may be able to guarantee that a certain number of geometric random variables with \( p = 1/2 \) are available and that enough bits are available for tests, assuming that the Poisson random variable process terminates by 7. If it does not, it may be necessary to check in the cases where we consider setting a variable to a value above 7, or in processing a value over 7, whether enough random information of the appropriate kind exists. On certain computers, such as the VAX and the CYBER 205, for which a right shift is a left shift by a negative amount, all geometric random variables should be negated in their construction. Of course, conditional transfers should be done differently on different machines. For each instruction, we give the expected number of times that instruction is taken per cycle, multiplied by 23040 to eliminate most fractions. The expected number of exponential random variables produced is \( \frac{35}{48}(e - 1) \), or 1.25291383, or 28867.1347 when multiplied. If we use the alternative method, the expected number of exponential random variables produced and the expected number of bits used for output purposes increase by \( 1/700 \), while the expected number of bits used in the preliminary process increases from 66000 to 66048. This decreases the expected number of bits used per exponential random variable generated by 0.001601; however, the amount of additional computation is less than would be expected, and this might pay. The expected number of times each operation in work is done is increased by \( 1/700 \) except the integers ending in "36," for which we must round up to the next integer. In the preliminary part, we indicate the additional computations.

We now produce some abbreviations to simplify the description.
done if enough exponential random variables have been obtained, finish up
chg check if there are enough geometric random variables available
     for the next cycle and refill if necessary
chb check if enough random bits are available for the next cycle
     and refill if necessary

OUT store the exponential random variable and initialize for the next
P1 add 1 to the integer part
O1 OUT with probability 1/2 and P1 with probability 1/2
G at each use, a new geometric random variable with p = 1/2
     (may be obtained as the distance to the next 1 in a random bit stream)
OUT2 clear the G'th bit in the fraction, store and initialize
B with probability 1/2 do the first branch; otherwise do the second
     branch (if the second branch is omitted, fall through)
work a subroutine to be transferred to (not called) to process values
     greater than 2
R return to the start of the cycle
q a mask of 1 in the bit to the left of the binary point

start of cycle

done
chg
chb

by = G&3
aa: O1;O1;R
ab: B
    O1;OUT2;R
    OUT2;O1;R
bc: u = G; if(u=1)
    OUT2;OUT2;R
else if(u=2)
    u=G+2; work;OUT2;R
else if(u=5)
    else OUT2;work;R
ac: u=G+2; B
    work;O1;R
    if(u=5)goto cc
    O1;work;R
cc: v=G+2; if(v=5)R
    u=G+2; work; u=v; work; R

** get S and T and set up **
** get the last 2 bits of G **
    23040
    23040
    12288
    6144
    3072
    3072
    1536
    768
    768
    384
    384
    336
    3072
    1536
    1536
    1344
    192
    168
If we use the alternative method, if \( u = 5 \) under bc we go to alta instead of aborting, and if \( v = 5 \) under cc we go to altb. The additional code is

alta:  
\[
O1;R
\]
   ** S = -1 **  
altb:  
\[
\begin{align*}
u & = G; \text{if}(u = 1) \\
O2;R & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \qa...
binomial coefficients to get the next step. For $k=4$ and $k=6$ we use a more complicated procedure. For $k>7$ we just look at $k$ random bits until a number other than 0 or $k$ occurs. The expected numbers are irrational because we are taking into account the fact that a given value of $k$ can also arise due to the reduction of $k$ until 1 is reached. The detailed instructions for shifting and masking are:

\[
\begin{align*}
f4: & \quad u=G; c=b\gg u; mask\ c; & \quad 730.5858 \\
\ & \text{if} (u\gg4)\text{goto f42; } & \\
\ & \quad B & \quad 417.4776 \\
\ & \quad \text{** next k is 1 or 3 **} & \quad 208.7388 \\
\ & \quad \text{** next k = 1 **} & \quad 208.7388 \\
\ & \quad B & \quad 3065.0225 \\
f3: & \quad b\gg g4; mask\ b; b\ll 1; & \quad 3118.9221 \\
f2: & \quad b\ll G; mask\ b; & \quad 3667.1347 \\
f1: & \quad OUT; return; & \quad 313.1082 \\
f42: & \quad b\gg g4; mask\ b; \text{goto f1; } & \quad 142.9446 \\
f5: & \quad b\gg g16; mask\ b; u=G; B & \quad 142.9446 \\
f512: & \quad \text{if(u odd)goto f2} & \quad 71.4723 \\
\ & \quad \text{else goto f1} & \quad 23.8241 \\
f534: & \quad g4=(u+1)/2; \text{if(u odd)goto f3; } & \quad 71.4723 \\
\ & \quad \text{else goto f4} & \quad 23.8241 \\
f6: & \quad u=G; \text{if} (g16 > u) \text{goto f65;} & \quad 23.5742 \\
\ & \quad \text{else} b\gg g16; mask\ b; \text{if(g16 = u)} & \quad 22.8137 \\
\ & \quad \{B & \quad 11.4069 \\
\ & \quad \text{goto} f4; & \quad 5.7034 \\
\ & \quad \text{goto} f2; \} & \quad 5.7034 \\
\ & \quad \text{else} u=u-g16; & \quad 11.4069 \\
\ & \quad \text{go to (f3,f6a,f1,f1) if (u mod4)=(1,2,3,0) } & \quad 3.0418 \\
f6a & \quad B & \quad 1.5209 \\
\ & \quad \text{goto} f3; & \quad 1.5209 \\
\ & \quad g16=(u+2)/4; \text{goto f5} & \quad .7605 \\
f65: & \quad \{c=b\gg u; \text{mask\ c; goto\ f5}\} &
\end{align*}
\]
f7:  \[ u = G; b > (u + 5)/6; u = (u + 4) \mod 6; \]
\[ \text{if } (u < 3) \text{ goto f7a} \]
\[ B \]
\[ \text{goto f3} \]
\[ \text{goto f4} \]

f7a:  \[ g_4 = g_4 - 1; \text{if}(g_4 \neq 0) \text{ goto f716}; \]
\[ B \]
\[ \text{goto f2}; \]
\[ \text{goto f5}; \]

f716:  \[ B \]
\[ \text{goto f1}; \]
\[ \text{goto f6}; \]

fh:  \[ b > > 1; \text{chb}; i = \# \text{of 1's in } k \text{ bits}; \]
\[ \text{if } (i = 0 \text{ or } i = k) \text{ goto fh}; \]
\[ ** \text{number of bits looked at} ** 3.8388 \]

k = i; mask b; \text{ goto } (f1, f2, f3, f4, f5, f6, f7, fh) \text{ for }
\[ i = (1, 2, 3, 4, 5, 6, 7, >7) \]