Statistics 580

Random Number Generation

1 Introduction

"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." This famous statement concerning the use of sequences of numbers generated using recursive formulas as random sequences is attributed to John von Neumann (1951). To generate a random number u from the Uniform (0,1) distribution, one must randomly select an integer $x \in (0, m)$ and set

$$u = \frac{x}{m}.$$

Machine-generated sequences of real numbers depend on recursions of the form:

$$x_j = f(x_{j-1}, x_{j-2}, \ldots, x_{j-k}),$$

or more expeditiously,

$$x_j = f(x_{j-1}).$$

Such sequences are obviously *deterministic*. We want to be able to generate sequences whose properties closely resemble those of random sequences in important ways. In addition, since such sequences are also necessarily *periodic*, we want such generators to have *long* periods, as well.

2 Linear Congruential Generators

`Linear Congruential Generators` are defined by the recursion

$$x_{i+1} \equiv (a\, x_i + c) \bmod m$$

where $x_0, x_1, \ldots$ is a sequence of integers and depends upon

$$\begin{aligned} x_0 &: \text{a seed} \\ a &: \text{a multiplier} \\ c &: \text{a shift} \\ m &: \text{a modulus} \end{aligned}$$

all of which are also integers "$\equiv$" defines an equivalence relation. Two numbers a and b are said to be *congruent modulo* m or $a \equiv b \bmod m$ where m is an integer, if their difference is exactly divisible by m. If $0 \leq a < m$ and $a \equiv b \bmod m$, then a is said to be a *residue* of b modulo m. a can be easily calculated using $a = b - \lfloor b/m \rfloor \times m$, where the *floor* function $\lfloor x \rfloor$ computes the greatest integer less than x.

The pseudo-random sequence $\{u_i\}$ is obtained by setting $u_i = x_i/m$ for i = 1, 2, ... If $c = 0$, the above defines a *multiplicative congruential generator* (MCG). Some theoretical

discussions on MCG's are given in Ripley(1987), Gentle(1998, 2003) and Fishman(2006). A typical choice for m on binary computers is 2^k where k is the typical machine length for integer type storage. In early generators this was thought to be a good choice because then the *modulo reduction* can be achieved simply via standard fixed-point overflow in the integer multiplication operation $(a\,x_i)$. Recall that if fixed-point overflow is not trapped, the result of an integer arithmetic multiplication is modulo reduced by 2^k. This is because if the product results in a number greater than $2^{31}-1$, only the lower order 32 bits are retained to represent the result. However, since the first bit in fixed-point representation is the sign bit, instead of integers in the range $(0, 2^{32}-1]$, integers in the range $[-2^{31}+1, 2^{31}-1]$ are produced by such multiplications.

The maximal period for a generator of this type is $2^k/4$; but it can only be realized if "a" is chosen such that $a\ mod\ \equiv\ 3$ or 5. On computers using 32-bits to store integers, a popular early choices were $m = 2^{32}$ and $a = 65539$. Since the speeds of modern-day computers allow the modulo reduction to be done efficiently many implementations of random number generators use extended-precision floating point computations. Other values of a such as 1099087573, 2396548189, 2824527309, 3934873077, and 2304580733 have been shown to produce sequences that perform well on some tests.

Another choice for m is a large prime p, whence the period is $p-1$ if the multiplier is chosen to be its primitive root. Primes of the form 2^p-1, called *Mersenne primes*, are of particular interest. On 32-bit machines, popular choices are $m = 2^{31}-1$ and its primitive root $a = 7^5 = 16807$. The IMSL routines `rnun( )` and `rnunf( )` use $m = 2^{31}-1$ and gives choices for a = 16807, 397204094, 950706376 to be selected by the user. The Unix C library contains the function `rand()` which implements the mixed generator

$$x_{i+1} \equiv (1103515245x_i + 12345) \bmod 2^{31}$$

This is discredited in the literature. The Unix `drand48` generator is defined on 48-bit unsigned integers with $a = 25214903917$, $c = 11$ and $m = 2^{48}$. It is implemented as the function `gsl_rng_uniform` in the GNU Scientific Library. The function `g05caf` in the NAG library uses the MCG:

$$x_{i+1} \equiv 13^{13}x_i \bmod 2^{59}$$

with the result divided by 2^{59} to convert to be in the range (0,1). The period of this generator is 2^{57}.

3 Linear Feedback Shift-Register Generators

Another type of random number generator employs the so-called *linear feedback shift-register* (FSR) technique, based on random sequences of bits. The i^{th} bit is computed from the previous p bits using a recurrence relation; thus for example: $b_i \equiv (a_1\,b_{i-1} + \cdots + a_d\,b_{i-p})\ mod\ 2$, where $a_1, a_2, \ldots, a_d$ are 0 or 1.

Suppose that $p = 5$ and $a_1 = a_2 = a_4 = 0$, $a_3 = a_5 = 1$. Then the successive bits (represented by b_6 in the table) are computed as shown in the table below:

a_i	0	0	1	0	1		
b_6	b_5	b_4	b_3	b_2	b_1		
1	1	0	0	1	1	$b_3 + b_1$	$(\text{mod } 2) = 1$
1	1	1	0	0	1	$b_4 + b_2$	$(\text{mod } 2) = 1$
1	1	1	1	0	0	$b_5 + b_3$	$(\text{mod } 2) = 1$
1	1	1	1	1	0	$b_6 + b_4$	$(\text{mod } 2) = 1$
0	1	1	1	1	1	$b_7 + b_5$	$(\text{mod } 2) = 0$
0	0	1	1	1	1	$b_8 + b_6$	$(\text{mod } 2) = 0$
.			.			.	
.			.			.	
.			.			.	

The above recurrence relation can easily be performed by hardware circuitry using shift-registers. A sequence of bits is shifted to the right (or left) one place at a time and the leftmost (or rightmost) bit i.e., b_i is computed using an *XOR* operation with the coefficients $a_1, \ldots, a_p$. The pseudo-random number is formed by taking a bit-string of length L from this sequence (a process called *L-wise decimation*):

$$u_i = \left(b_{(i-1)L+1}\, b_{(i-1)L+2} \ldots b_{iL}\right)_2 / 2^L$$

for i = 1,2,...

The above generator attains its maximum period of $2^p - 1$ if and only if the polynomial

$$f(x) = x^p + a_{p-1}\, x^{p-1} + \cdots + a_1 x_1 + 1$$

is irreducible on GF(2). Polynomials usually considered are trinomials of the form

$$1 + x^q + x^p$$

with $1 \leq q < p$, i.e., the generator is

$$b_i \equiv (b_{i-p} + b_{i-q}) \bmod 2$$

Addition of 0's and 1's modulo 2 is the binary *exclusive OR* operation denoted by $\oplus$. Using this operator the above can be expressed as

$$b_i = b_{i-p} \oplus b_{i-q}.$$

The recurrence expressed in bits can be translated to the numbers x_i formed by the L-tuples and expressed in binary, Thus

$$x_i = x_{i-p} \oplus x_{i-q}$$

holds, where $\oplus$ is understood to be a bitwise operation.

In our example above, uses the polynomial

$$x^5 + x^3 + 1$$

and thus beginning with the bit sequence 10011 will produce the sequence

$$11110\ 00110\ 11101\ 00001\ 00101\ \ldots$$

5-wise decimation gives the sequence $30, 6, 29, 1, 5, \ldots$ which are divided by 32 to give the uniforms needed. The above polynomial is irreducible; thus the period is $2^5 - 1 = 31$. In a variation, called the generelized feedback shift register (GFSR) method, instead of taking succesive bits from the sequence to form the uniforms, bits are taken with a *delay*. For example, instead of the bits $(b_i, b_{i+1}, \ldots, b_{i+L})$ the bits accessed to form u_i is of the form $(b_i, b_{i+j1}, b_{i+j2}, \ldots, b_{i+j(L-1)})$ where $j1, j2, \ldots$ may all be the same delay.

The `random()` function on Unix (C or Fortran) implements an FSR with

$$x_i^{31} + x_i^3 + 1 \quad i.e. \quad p = 31 \quad q = 3$$

`random()` family of functions is a set of linear feedback shift register generators originally used in BSD Unix, and a version is available on GNU/Linux systems. The uniform generator in Splus is an example of a coupled generator: combination of 2 or more types of generators. The Splus routine `runif()` uses a combination of an MCG and FSR. There are several GFSR and mixed type generators available in the GNU Scientific Library (GSL) one of which is the generator `taus2` which is one of the fastest simulation quality generators available.

4 Portable Generators

The Fortran routine uni() from CMLIB is an example of `portable` random number generator. It uses an FSR and uses a table of machine constants to adapt the generator to different computers. The general idea is to make generators machine independent. That is, a generator should be able to produce the same sequence of numbers on different machines. Schrage (1979) proposed a Fortran implementation of the MCG with $m = 2^{31} - 1$ and $c = 16807$. (The IMSL routines use the same generator but implements it using double precision arithmetic.)

Wichmann and Hill (1982) uses 3 simple MCG's each with a prime for modulus and a primitive root for multiplier. If the results from each are added, the fractional part is $U(0, 1)$. This follows from the fact that if X and U are independent and $U \sim U(0, 1)$, then the fractional part of $X + U$ is $U(0, 1)$ irrespective of the distribution of X. The generator is defined by the relations:

$$\begin{aligned} x_{i+1} &\equiv 171\, x_i \ mod\ 30269 \\ y_{i+1} &\equiv 172\, y_i \ mod\ 30307 \\ z_{i+1} &\equiv 170\, z_i \ mod\ 30323 \end{aligned}$$

and returns u_i where

$$\begin{aligned} u_i &= (x_{i+1}/30269 + y_{i+1}/30307 + z_{i+1}/30323) \\ u_i &= u_i \ mod\ 1.0 \end{aligned}$$

This generator requires three seeds x_0, y_0, and $z)0$ and has a period in the order of 10^{12}. Studies have shown that higher-order autocorrelations in sequences from the Wichman-Hill generator compare favorably with those from other good generators. L'Ecuyer (1988) proposes an efficient portable generator with a very long period, defined by:

$$\begin{aligned} x_{i+1} &\equiv 40014\, x_i \ mod\ 2147483563 \\ y_{i+1} &\equiv 40692\, y_i \ mod\ 2147483399 \end{aligned}$$

and returns

$$v_i \equiv (x_i + y_i) \; mod \; 2147483563$$

and gives code for fast and portable implementation. A C implementation of the Wichmann-Hill generator is given in Figure 1.

```
/*****************************************************************/
/* Function  WICHHILL                                            */
/* This function generates and returns one Uniform(0,1) random   */
/* variate.  It uses the Wichmann-Hill algorithm.  Three seeds   */
/* addresses from 1-30,000 are passed to the subroutine and used */
/* to generate the variate.                                      */
/*****************************************************************/

#include <stdlib.h>
#include <math.h>
double fmod(double, double);
double wichhill(int *ix, int *iy, int *iz)
{
  double rand;

  *ix = (171**ix)%177 - (2*(*ix/177));
  *iy = (172**iy)%176 - (35*(*iy/176));
  *iz = (170**iz)%178 - (63*(*iz/178));

  if (*ix < 0)
    *ix += 30269;
  if (*iy < 0)
    *iy +=  30307;
  if (*iz < 0)
    *iz +=  30323;

  rand = fmod(((*ix)/30269.0f + (*iy)/30307.0f + (*iz)/30323.0f),1.0f);
  return rand;
}
```

Figure 1: A C Language Implementation of the Wichmann-Hill Generator

5 Recent Innovations

Marsaglia and Zaman(1991) introduced the *add-with-carry* (AWC) generator of the form

$$x_i \equiv (x_{i-s} + x_{i-r} + c_i) \; mod \; m$$

where r and s are lags and $c_i = 0$ and $c_{i+1} = 0$ if $x_{i-s}+x_{i-r}+c_i < m$ and $c_{i+1} = 1,$ otherwise. The c is the "carry." The *subtract-with-carry*(SWC) is given by

$$x_i \equiv (x_{i-s} - x_{i-r} - c_i) \; mod \; m.$$

Since there is no multiplication these can be implemented to be very fast and have large periods. Research has shown that the sequences resulting from these generators are similar to sequences obtained from linear congruential generators with very large prime moduli. However, other work show that the lattice structure of these may be very poor. Marsaglia also describes a *multiply-with-carry* (MWC) generator

$$x_i \equiv (ax_{i-} + c_i) \; mod \; m$$

and suggests $m = 2^{32}$ and an implementation in 64-bits.

The GFSR generators have been modified by "twisting" the bit pattern in x_{i-q} by pre=multiplying the "vector" of bits in x_{i-q} by the $L \times L$ matrix C. The recurrence then becomes

$$x_i = x_{i-p} \oplus Cx_{i-q}$$

. The famous *Mersenne twister* is a twisted GFSR proposed by Matsumoto and Nishmura (1998), where the matrix C has special properties leading to the generator to have a period of $2^{19937} - 1$ and 623-variate Uniformity. They gave a C program named `mt19937`. The initialization procedure for this generator is rather complicated and described on the website `http://www.math.keio.ac.jp/~matumoto/emt.html` by the main author Matsumoto.

Several choices of basic unform generators are available in R. The R function `RNGkind()` may be used to select the generator. The default is the Mersenne twister. Wichmann-Hill and several Marsaglia generators, including the MWC, are available.

6 More Reading

You may add to the general ideas introduced here by perusing some additional material on uniform random number generators in the references such as Ripley(1987), L'Ecuyer(1998), Monahan(2001), Gentle(2003), Fishman(2006), and the paper by Atkinson (1980). Specifically, research that has looked at the structure of generated random number sequences using graphical methods and the use of standard empirical and theoretical tests specifically constructed for checking the randomness and goodness of fit of these sequences such as the runs test, lattice test, etc. are informative. The source code for the random number generator battery of tests known as `DIEHARD` is available online at `http://stat.fsu.edu/pub/diehard/` at Florida State University. Also read about the use of *shuffling* of generated sequences to break up lattice structure and increase period. An example is the Bays-Durham Shuffler.

`Bays-Durham Shuffler (as described in Gentle,1998 and 2003)):`

Initialize table T with ℓ $U(0,1)$ numbers.

Set $i = 0$.

Do until $i = n$

1. generate random integer k in $[1, \ell]$
2. set $i = i + 1$
3. set $u_i = T(k)$
4. replace $T(k)$ with a new $U(0,1)$

7 Generating Non-uniform Random Numbers

Special generating methods exist for various individual distributions: for e.g.,

Normal: Box-Müller Algorithm, Polar Method
Gamma: Sum of independent exponentials
χ^2: Sum of squares of independent N(0,1)'s
Binomials: Sum of independent Bernoulli's
Poisson: based on the definition of a Poisson Process.

among many others. These are described in several of the references, especially in Kennedy and Gentle(1980), Devroye(1986), Gentle(1998, 2003), and Fishman(2006). Here we look at a few interesting methods that are applicable to many different distributions. General methods discussed below are

- Table look-up Methods for discrete distributions.
- Transformation Methods e.g., inverse cdf method.
- Rejection (or Acceptance/Rejection) Methods.

7.1 Table Look-up (or Table Sampling) Methods

The most elementary application of *inverse cdf method* to discrete distributions is sampling from the *Bernoulli distribution*. Let X be a random variable with pmf

$$p(x) = \pi^x(1-\pi)^{1-x}, \quad \text{for } x = 0, 1.$$

To sample a random variate from this distribution:

1. Generate $u \sim U(0,1)$.
2. If $u < \pi$ set $i = 1$; Else set $i = 0$
3. Return $X = i$.

If the cumulative probability distribution of a discrete random variable is denoted by $p_i = Pr(X \leq i)$, $i = 0, 1, 2, \ldots$, then given u from $U(0,1)$, then the value of i that satisfies

$$p_{i-1} < u \leq p_i \quad for \quad i = 0, 1, \ldots$$

is a random variate from the corresponding discrete distribution. For example, to sample from the *geometric distribution* with pmf

$$p(x) = \pi(1-\pi)^x, \quad 0 < \pi < 1, \text{ for } x = 0, 1, \ldots,$$

consider its cdf

$$F(x) = 1 - \pi^{x+1}, \quad \text{for } x = 0, 1, \ldots,$$

it is needed to find x that satisfies

$$1 - (1-\pi)^x < u \leq 1 - (1-\pi)^{x-1}$$

which can be shown to be equivalent to

$$x < \frac{log(1-u)}{log(1-\pi)} \leq x + 1.$$

Thus to generate a random variate from the geometric distribution

1. Generate $u \sim U(0,1)$.
2. Return $X = \lfloor \frac{log(u)}{log(1-\pi)} \rfloor$.

7.2 Direct Search Method

If the cumulative probability distribution of a discrete random variable is denoted by $Pr(X \leq i), \quad i = 0, 1, 2, \ldots$, then given u from $U(0,1)$, the smallest i that satisfies

$$u < Pr(X \leq i) \quad for \quad i = 0, 1, \ldots$$

is a random variate from the corresponding discrete distribution. If the cumulative probability distribution is computed and stored in a table in advance, computing i is equivalent to locating the smallest i for which $Pr(X \leq i)$ exceeds or equals u. Repeating this procedure with independently generated u's for generates a random sample from the discrete distribution.

Algorithm:
Set-up a table of values $p_i = Pr(X \leq i)$ for $i = 0, 1, \ldots$.

1. Generate $u \sim U(0,1)$ and set $i = 0$.
2. While $u > p_i$ do $i = i + 1$.
3. Return $X = i$.

Remarks:

1. Random variables with infinite ranges need to have the range truncated, i.e., ignore values m for which
$$1 - Pr(X \leq m) \leq \epsilon$$
for a sufficiently small ϵ.
2. Efficient algorithms are needed to do the search on the table, since a direct search of an array may require too many comparisons, especially for large tables. Examples of those available are the binary search and indexed search algorithms described below.

Example:

We consider the Poisson distribution as an example of generating random deviates from a discrete distribution. The pmf of the Poisson distribution with mean λ is given by

$$P(X = x) = \frac{e^{-\lambda}\lambda^x}{x!}, \quad x = 0, 1, 2, \ldots$$

Consider the random variable X which has a Poisson distribution with mean 2. In order to set-up the table look-up method, first the cumulative distribution needs to be tabulated. Since the range of values that x can take is infinite, we will truncate (cut-off) the tabulation at some suitably large value of x (i.e., here we shall take ϵ to be .0001). The Poisson(2) cdf is tabulated below correct to four digits:

x	0	1	2	3	4	5	6	7	8	9
$P(X \leq x)$	.1353	.4060	.6767	.8571	.9473	.9834	.9955	.9989	.9998	1.000

To generate a Poisson random variate, generate a $U(0, 1)$ variate u and locate the smallest x for which a cumulative probability in the above table exceeds u. For example, if $u = 0.8124$, then $P(X \leq x)$ exceeds u for all $x \geq 3$; thus the required Poisson variate is $x = 3$.

Notice that we truncated the cdf of Poisson distribution at $x = 9$, because only four digits were used to represent the probabilities in the above table. In practice, when such tables are stored in computer memory, the truncation can be done at a larger x value since the probabilities can be represented more accurately in floating point. That is, ϵ can be chosen to be suitably small, say 0.1×10^{-10}, for example. Note also that there is a constant overhead associated with the computation of this table. Thus table look-up methods will not be efficient unless a large number of random variates are to be repeatedly generated using the same cdf table.

7.3 Binary Search Algorithm

The previous algorithm is a very naive approach for searching a table. It does not take advantage of the fact that the values in the table are already ordered and monotonely increasing. This leads to the possibilty that methods used in zero-finding algorithms could be used to exploit this fact.

Algorithm:
Set-up a table of values $p_i = Pr(X \leq i)$ for $i = 0, 1, \ldots, m$.

1. Set $L = 0, R = m$.
2. Generate $u \sim U(0, 1)$.
3. Repeat until $L \geq R - 1$:
   (a) Set $i = \lfloor (L + R)/2 \rfloor$.
   (b) If $u < p_i$ then $R = i$,
   else $L = i$
4. Return $X = i$.

7.4 Indexed search or cut-point method.

The number of comparisons needed in a table search can be reduced if the search is begun at a point nearer to the index i that satisfies the above criterion. In the algorithm of Chen and Asau (1974), the search point is made to depend on u in such a way that a starting value is selected which is as close to index i we need yet will not exceed that value:

Indexed Search Algorithm:
Set up tables of values

$$p_i = Pr(X \leq i) \text{ for } i = 0, 1, 2, \ldots$$

and

$$q_j = min\{i | p_i \geq j/m\} \text{ for } j = 0, 1, \ldots, m-1$$

where m is an integer selected beforehand. The set of values q_j are called the cut-points.

1. Generate $u \sim U(0,1)$. Set $k = \lfloor mu \rfloor$ and $i = q_k$.
2. While $u \geq p_i$ do $i = i + 1$.
3. Return $x = i$.

Method	μ			
	5	10	20	50
Algorithm 3.3				
Set up (msec)	14	22	37	85
Per call (msec)	33	59	109	260
Straight search				
Set up (msec)	180	280	470	910
Per call (msec)	15	26	49	114
Per call – 3.10A (msec)	26	30	34	40
Indexed search ($m = 2\mu$)				
Set up (msec)	280	470	820	1780
Per call (msec)	8.2	8.1	8.1	7.7
Mean comparisons	1.56	1.51	1.42	1.61
Binary search				
Set up (sec)	1.4	2.7	6.2	22.4
Per call (msec)	15	18	20	22
Alias				
Per call (msec)	6.8	6.8	6.8	6.8
Set up – 3.13A (sec)	2.08	4.50	10.4	38.0
Set up – 3.13B (sec)	0.58	0.89	1.47	2.79
Algorithm 3.15				
Set up (msec)	–	56	57	57
Per call (msec)	–	132	125	169

Table 1: Timings on BBC Microcomputer of Methods for the Poisson Distribution with mean μ from Ripley(1986)

The idea behind the algorithm is that since we want to find the smallest i for which $p_i > u$ (say i_1), we start at the smallest i, (say i_2) for which $p_i \geq k/m$. Since $k/m \leq u$ (obviously, since $k = \lfloor mu \rfloor$) we have $i_2 \leq i_1$. Thus it is guaranteed that i_2 is an integer that is smaller than the index of the required element.
Remarks:

1. Gentle(1998, 2003) discuss another probability based table-sampling method due to Marsaglia (1963).

2. The Alias method of Walker (1974, 1977) discussed in Ripley(1986)and Gentle(1998,2003) is a table-based method using composition.

3. Several algorithms for generating from the Poisson distribution were compared using a simulation study by Ripley(1986). The results appear in Table 1.

7.5 Transformation Methods

The Box-Müller method for generating Normal random variates is an example of a transforming Uniform random numbers to variates from other distributions. If U_1, U_2 are independently distributed U(0,1) random variables then it can be proved that

$$\begin{aligned} U_1 &= \sqrt{-2\log(U_1)}\,\sin(2\pi U_2) \\ U_2 &= \sqrt{-2\log(U_1)}\,\cos(2\pi U_2) \end{aligned}$$

are independent *Standard Normal* random variables. Thus it is easy to transform two get a sample of two standard normals from two uniforms. A variation of the same method is based on expressing the transformation in terms of polar co-ordinates. The resulting *Polar Method* avoids the use of the *computationally expensive* functions `cos()`, `sin()`, and `log()`. It is given by

1. Generate u_1 and u_2 from $U(0,1)$ and set $v_1 = 2u_1 - 1$ and $v_2 = 2u_2 - 1$.

2. Set $s = v_1^2 + v_2^2$. If $s > 1$, go to Step 1.

3. Set $c = (-2\log(s)/s)^{1/2}$.

4. Return $x_1 = cv_1$ and $x_2 = cv_2$.

Many other efficient and theoretically interesting algorithms have been proposed for the generation of Standard Normal deviates. The general method, known as inversion and discussed below, can also be applied to the Normal distribution. This method requires an accurate algorithm for the numerical inversion of the cdf of the Standard Normal distribution. This is because a closed form expression for the inverse of the normal cdf does not exist.

The `method of inversion` (or `the inverse cdf method` as it is sometimes called) can be used to obtain transformations for many distributions. This method is based on the following well-known result. Consider the problem of generating a random variable X from a distribution F and suppose that F is continuous and strictly increasing and $F^{-1}(u)$ is

well-defined for $0 \leq u \leq 1$. If U is a random variable from $U(0,1)$, then it can be shown that $X = F^{-1}(U)$ is a random variable from the distribution F.

This result can be used to obtain closed form transformations only if F can be inverted analytically i.e., the equation $u = F(x)$ can be solved for x in closed form. However, if F can be accurately and efficiently inverted numerically, this method can be very useful. For e.g., numerical inversion of the Standard Normal cdf is used as the basis for several functions available in libraries for generating from the Standard Normal distribution. The IMSL routine RNNOR uses the inverse cdf method for Standard Normals with numerical inversion of the cdf.

`Example`

Consider the *Exponential distribution* with mean θ. The density and the cdf are, respectively,

$$\begin{aligned} f(x) &= \frac{1}{\theta} e^{-x/\theta} I_{(0,\infty)}(x) \\ F(x) &= 1 - e^{-x/\theta} \end{aligned}$$

Setting $u = 1 - e^{-x/\theta}$ and solving for x gives

$$x = -\theta \ell n(1-u)$$

So to sample x from Exp(θ) distribution, generate u from $U(0,1)$ and plug it in the above transformation. Thus, an algorithm for generating Exponential variates with parameter θ is:

1. Generate u from $U(0,1)$.
2. Set $x = -\theta \log(1-u)$.
3. Return x.

As a further example of the inversion method, we consider generating random variables from the *Triangular distribution* with the pdf given by

$$\begin{aligned} f(x) &= 2(1-x) \quad , \quad 0 \leq x \leq 1 \\ &= 0 \quad , \quad \text{otherwise} . \end{aligned}$$

We obtain the cdf to be

$$F(x) = \int_0^x 2(1-x)dx = (2-x)x .$$

Setting $u = F(x)$ and solving for x we obtain $x = 1 + \sqrt{(1-u)}$. We discard the solution $x = 1 + \sqrt{(1-u)}$, since x then will be outside the range of possible values of x, i.e., [0,1]. Thus, we use the transformation $x = 1 - \sqrt{(1-u)}$ to generate variates from the above distribution.

The density function of *the Gamma distribution* with parameters α and β is given by

$$f(x) = \frac{1}{\Gamma(\alpha)\beta^\alpha} x^{\alpha-1} e^{-x/\beta} , \quad x > 0, \quad \alpha, \beta > 0 .$$

When $\beta = 1$, the distribution is called the Standard Gamma. A Standard Gamma variable Y can be easily converted to a general Gamma variable X by the transformation $X = \beta Y$. Thus, only the problem of generating a Standard Gamma needs to be considered.

Consider the case when α is an integer. In this case a Standard Gamma variable is easily generated by summing α independent Exponential variates each with parameter $\theta = 1$. This gives the following algorithm for generating Gamma variables with parameters α and β, where α is an integer:

1. Generate $u_1, u_2, \ldots, u_\alpha$ independent $U(0,1)$ variates.
2. Set $y_i = -\log(1 - u_i)$ for $i = 1, 2, \ldots, \alpha$.
3. Set $s = \sum_{i=1}^{\alpha} y_i$ and $x = \beta s$.
4. Return x.

A classical method known as the Wilson-Hilferty algorithm for generating approximate χ^2 random variates uses the normal approximation to the χ^2 distribution. An approximate χ^2 variable with k degrees of freedom is given by

$$X = k\left[\sqrt{2/9k}\, Z + (1 - 2/9k)\right]^3 ,$$

where Z is a Standard Normal variate. However, since the *Chi-squared distribution* is a special case of the Gamma distribution, eficient techniques available for the Gamma can be specialized for the Chi-square.

Using the notation introduced for the Gamma distribution, the χ^2 with k degrees of freedom (i.e., a $\chi^2(k)$) is the same as a Gamma variable with $\alpha = k/2$ and $\beta = 2$. Thus, a simple algorithm for generating χ^2 with integer degrees of freedom is as follows:

1. When $k = 2n$ (i.e., when k is an even integer), use the algorithm in Section 9.5 to generate a Gamma variates with $\alpha = n$ and $\beta = 2$. This is a χ^2 with $k = 2n$ degrees of freedom.
2. When $k = 2n+1$ (i.e., when k is an odd integer), use part (a) above for generating a χ^2 with $k = 2n$ degrees of freedom and denote this by Y. Then generate an independent $N(0,1)$ variate Z and set $X = Y + Z^2$. Then X is a χ^2 with $k = 2n + 1$ degrees of freedom.

The *Beta distribution* with parameters a and b has the density function given by

$$f(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\, x^{a-1}(1-x)^{b-1}, \quad 0 \le x \le 1$$

Various values of the parameters allow many different shapes of the density, which makes the Beta distribution useful for modelling a variety of input distributions in simulation. Also of interest is the fact that ratios of Beta random variables yield Chi-square and F variables. One of the first algorithms for Beta variables with arbitrary parameters was given by Johnk (1964).

1. Generate u_1 and u_2 independently from $U(0,1)$.
2. Set $x = u_1^{1/a}$, $y = u_2^{1/b}$ and $w = x + y$.
3. If $w \leq 1$, set $x = x/w$ and deliver x.
   Else, go to 1.

7.6 Rejection Methods

Suppose it is required to generate a random variate from a distribution with density $f(\cdot)$. Let $g(\cdot)$ be another density defined in the support of f such that $f(x) \leq c\, g(x)$, where $c > 1$ is a known constant, holds for all x in the support. A method for to generate random variates from g must be available. The function $q(x) = c\, g(x)$ is called an *envelope* or a *majorizing function*. Now we can sample a variate y from g with a probability that depends on y itself:

`Rejection Algorithm:`
Repeat
1. Generate y from $g(\cdot)$.
2. Generate u from $U(0,1)$

Until
3. $u \leq f(y)/c\, g(y)$

Return $X = y$

It can be shown that X has the required distribution. The closer the enveloping function to $f(\cdot)$ the better the *acceptance rate*. For this method to be efficient, the constant c must be selected so that the *rejection rate* is low. For a random variable U,

$$\begin{aligned} Pr\left(U \leq \frac{f(y)}{c\, g(y)}\right) &= Pr\left(u \leq \frac{f(Y)}{c\, g(Y)} \,\middle|\, Y = y\right) \\ &= \frac{f(y)}{c\, g(y)} \end{aligned}$$

Thus, the probability of acceptance, p, is calculated as

$$\begin{aligned} p = Pr\left(U \leq \frac{f(Y)}{c\, g(Y)}\right) &= \int_{-\infty}^{\infty} Pr\left(U \leq \frac{f(Y)}{c\, g(Y)} \,\middle|\, Y = y\right) g(y)dy \\ &= \int_{-\infty}^{\infty} \frac{f(y)}{c\, g(y)} \cdot g(y)dy = \frac{1}{c} \end{aligned}$$

If T denotes the number of trials *before* acceptance, then we see that T has a Geometric distribution with parameter p, and the density is:

$$f(t) = p(1-p)^t, \qquad t = 0, 1, \ldots$$

Thus the expected value of T is given by

$$E(T) = \frac{q}{p} = \frac{1 - 1/c}{1/c} = c - 1$$

Thus the expected number of trials for an acceptance is c and thus the smaller c is, the lower the rejection rate. However, for an accepted y, c still needs to satisfy

$$c \geq f(y)/g(y)$$

so that one way to choose an optimum c is by setting

$$c = \sup_x f(x)/g(x)$$

Even this choice of c however, may result in an acceptably large number of rejections, making the algorithm inefficient. In practice, the envelope is found by first considering another density with similar shape for which a method for generating variables is available. This density is then multiplied by a constant to obtain an envelope.

`Example 1:`

As an example consider the half-normal density

$$f(x) = \sqrt{\frac{2}{\pi}}\, \exp^{-x^2/2}\, I_{[0,\infty)}(x)$$

as shown in Figure 2. We will try $g(x) = e^{-x}$, i.e., the Exponential density with mean 1. The support of this density is $[0, \infty)$ as needed. The ratio

$$\frac{f(x)}{g(x)} = \frac{\sqrt{2/\pi}\exp^{-x^2/2}}{\exp^{-x}} = \sqrt{2/\pi}\exp^{-x^2/2+x}\,.$$

Thus, we find the optimum c to be

$$c = \sup_x f(x)/g(x) = \sqrt{2e/\pi}$$

giving us the envelope

$$q(x) = \sqrt{2/\pi}\exp^{-x+1/2}\,.$$

`Example 2:`

Consider the Beta density

$$f(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1} I_{[0,1]}(x) \quad a \geq 1,\ b \geq 1$$

and use the density

$$g(x) = ax^{a-1} I_{[0,1]}(x)$$

to construct an envelope. Taking

$$q(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}$$

we see that $q(x) \geq f(x)\ \forall\ x\epsilon[0,1]$, $a \geq 1$. We generate y from g easily using $y = u^{1/a}$ where u is a $U(0,1)$.

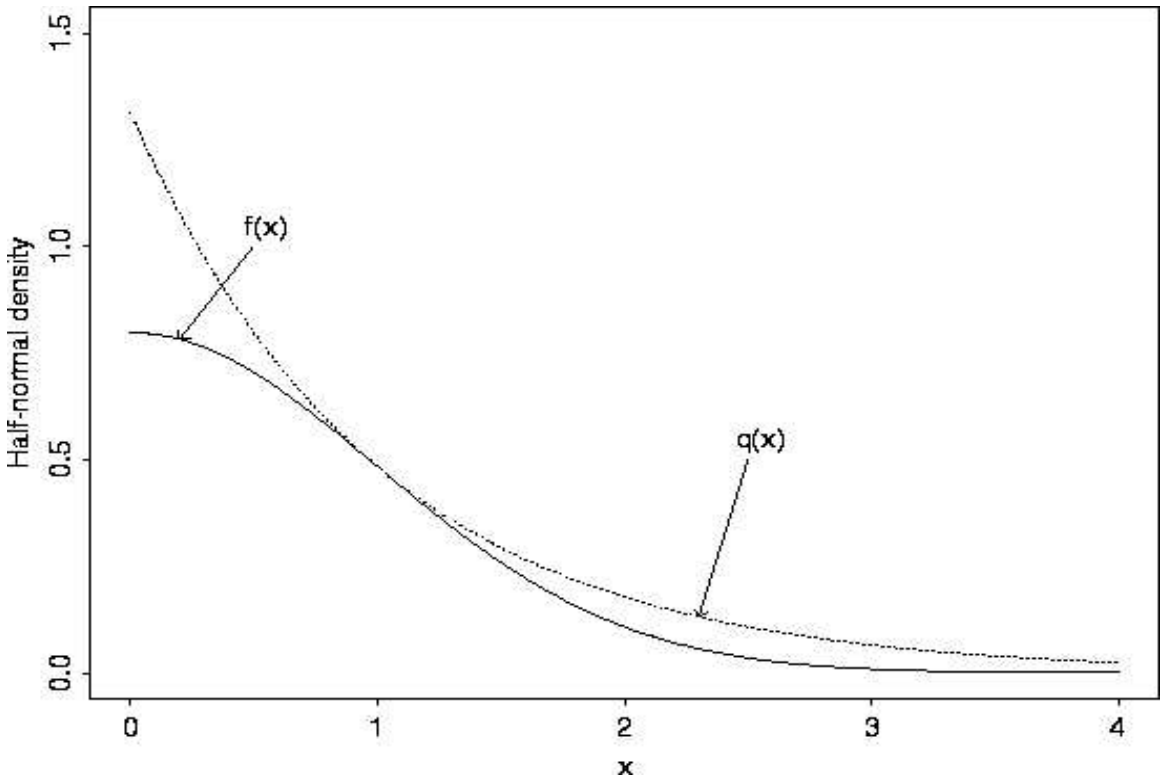

Figure 2: An illustration of the optimum choice of k. Here we illustrate the use of the exponential e^{-x} probability density function as the basis of an envelope for the half-normal density (solid line). The enveloping function is given by $\sqrt{\left(\frac{2e}{\pi}\right)}\, e^{-x}$ (dashed line), illustrated for $x \leq 4$, as is $f(x)$(solid line)

Consider the Beta density with shape parameters $\alpha = 3,\ \beta = 2$ being majorized using two line segments forming a triangular density (see Figure 3. It is quite easy to derive the majorizing function $q(x)$ and therefore $g(x)$ in this case.

It is important to note that, in general, $f()$ needs to be known only upto a constant i.e., $f()$ may be known to be proportional to a density but the constant of integration is not known. This characteristic of the method is valuable when random samples are to be drawn from a distribution of which only the form of the density is known, as occurs quite often in sampling from posterior densities in Bayesian statistics.

As an example, consider the simple case where

$$f(x) = x^{a-1}(1-x)^{b-1} I_{[0,1]}(x) \quad a \geq 1,\ b \geq 1$$

i.e., $f(x)$ is proportional to a Beta density. Then taking

$$g(x) = a x^{a-1} I_{[0,1]}(x)$$

as before, we shall find the optimal c to be

$$c = \sup_x f(x)/g(x) = \sup_x \frac{(1-x)^{b-1}}{a} = \frac{1}{a}$$

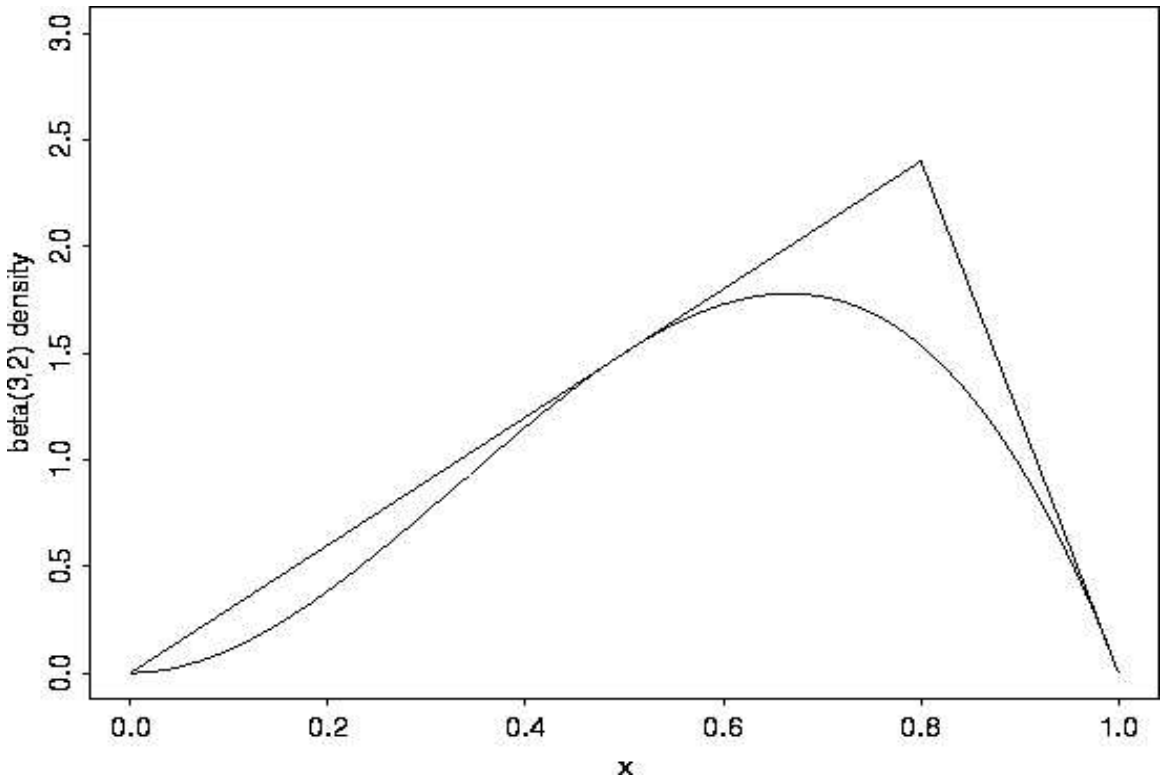

Figure 3: Envelope for the Beta(3,3) density

That is accept x when

$$u \leq \frac{f(x)}{c\, g(x)} = \frac{x^{a-1}(1-x)^{b-1}}{\frac{1}{a}ax^{a-1}} = (1-x)^{b-1}$$

where x is drawn from $g(x)$. This is the same result as obtained above.

`Example 3`: Consider sampling from the positive tail of the normal distribution. That is, we want a realization from the density:

$$f(x) = k \exp\{-\frac{x^2}{2}\} I_{[d,\infty)}; \tag{1}$$

Notice that the half-normal distribution is a special case of this density. In order to simulate from this density, we have to find a density g which is *close* to f that majorizes f. Consider the following choice for g:

$$g(x) = x \exp\{-\frac{(x^2 - d^2)}{2}\} I_{[d,\infty)}. \tag{2}$$

In order to use the rejection algorithm, we need to choose c appropriately. Since

$$\frac{f(x)}{g(x)} = \frac{k \exp\{-\frac{x^2}{2}\}}{x \exp\{-\frac{(x^2-d^2)}{2}\}} = \frac{k}{x \exp\{\frac{d^2}{2}\}}$$

The supremum of this over the support of g is easily seen and thus $c = \frac{k}{d}\exp\{-\frac{d^2}{2}\}$ and thus the envelope is

$$q(x) = cg(x) = \frac{kx}{d}\exp\{-\frac{x^2}{2}\}$$

It can be shown that $g()$ is the pdf of $\sqrt{X + d^2}$, where $X \sim \exp(2)$. Hence we get the following procedure for simulating a random variate from the right tail of the standard Gaussian distribution:

`Repeat`

1. Generate a variate with pdf g by the method of inversion to get $x = \sqrt{d^2 - 2\log u_1}$ where $u_1 \sim U(0,1)$.
2. Generate u_2 from $U(0,1)$

`Until` $u_2 \leq \frac{d}{x}$.
`Return` $X = x$

Note that for $d = 1$, X has an acceptance rate of 0.66 while for $d = 2$, X has an acceptance probability of 0.88. In particular, as $d \to \infty$, the efficiency of the sampling algorithm increases to 100%. Finally, note that the above again confirms that $f(x)$ need be known only upto a constant

7.7 Rejection with Squeezing

To increase the efficiency of the Rejection method when the calculation of $f(x)$ is expensive (such as when it involves algebraic or transcendental functions), the comparison step (step 3.) in the above algorithm can be broken down in to two steps:

`Squeeze Algorithm:`

Repeat
1. Generate y from $g()$.
2. Generate u from $U(0,1)$

Until
3a. $u \leq h(y)/c\,g(y)$
Else
3b. $u \leq f(y)/c\,g(y)$
Return $X = y$

where $h(x)$ is a *minorizing function* i.e.,$h(x) \leq f(x)$ in the support of f and $h(x)$ is simpler to evaluate. Thus a preliminary comparison is made with a lower bound of $f(y)/c\,g(y)$. This is called the *squeeze* method and the squeeze function is usually a piecewise linear function.

One very nice application of the acceptance/rejection algorithm and the squeeze method is the well known algorithm proposed by Kinderman, Monahan, Ramage(1977) for generating variates from the t-distribution. This method is very succinctly described in Monahan(2001) (p. 290) who also provides a Fortran funtion `gttir`.[See Stat580 webpage for alink to extracted pages from this description]

7.8 Adaptive Rejection Sampling

The generalization of the idea of using straight lines for majorizing the Beta density (see example above) is the basis for the *adaptive rejection sampling* or ARS method for distributions with log-concave densities. If a density $f(x)$ is log-concave, then any line tangent to $\ln f(x)$ will lie above $\ln f(x)$. Thus log-concave densities are ideally suited for the rejection method with piecewise linear functions as majorizing functions for the log-density or the density itself can be majorized using piecewise exponential functions. Log-concavity of a density $f(x)$ can be easily verified by checking whether $\frac{d^2}{dx^2} \ln f(x) \leq 0$ on the support of $f(x)$. For example, it is easily seen that Normal density is log-concave since the second derivative of $\ln f(x)$ is $-1/\sigma^2$. A short discussion of the ARS method follows.

Obviously the choice of suitable envelopes in a given situation determines the efficiency of an algorithm for generating random samples based on the acceptance/rejection or the squeeze method. Gilks(1992) and Gilks and Wild(1992) proposed a method for generating envelopes automatically for the squeeze method for a continuous, differentiable, log-concave density having a connected region as support. The method refines the *envelope* and the *squeezing function* as the iteration progresses and thus reduces the rejection rate and the number of function evaluations.

Let $\ell(x) = \log f(x)$ and thus $\ell(x)$ is concave i.e., $\ell(a) - 2\ell(b) + \ell(c) < 0$for any three points $a < b < c$ in the support of f. Also note that because of the assumption of differentiability and continuity of f, $\ell'(x)$ decreases monotonically with increasing x. This allows $\ell'(x)$ to be both straight-line segments as well as have discontinuities.

At the beginning a k is chosen and both ℓ and ℓ' are evaluated at the set S_k of points $x_1 < x_2 < \ldots < x_k$. (For the moment, assume that the support of f does not extend to $\pm\infty$.) The rejection envelope defined on S_k is the exponential of the piecewise linear upper hull of ℓ formed by the tangents to ℓ at each point in S_k. The concavity ensures that the tangents lie completely above ℓ; thus the envelope lies completely above f.

This envelope can be calculated by noting that the tangents intersect at the points

$$z_j = \frac{\ell(x_{j+1}) - \ell(x_j) - x_{j+1}\ell'(x_{j+1}) + x_j\ell'(x_j)}{\ell' x_j) - \ell'(x_{j+1})},$$

for $j = 1, \ldots, k-1$. Thus the equation of the tangent in the region $[z_{j-1}, z_j]$ is

$$g_1^*(x) = \ell(x_j) + (x - x_j)\ell'(x_j) \qquad \text{for } x \in [z_{j-1}, z_j],$$

for $j = 1, \ldots, k-1$, where z_0 and z_k are defined to be the bounds (possibly infinite) of the support region of f. Thus the rejection envelope for $f(x)$ based on tangents is thus $\exp g(x) = \exp\left(g_1^*(x)\right)$

Similarly, the squeeze function $h(x)$ in S_k, is the exponential of the piecewise linear lower hull of ℓ formed by the chords to ℓ between successive points in S_k. The equation of the chord in the region $[x_j, x_{j+1}]$ is

$$h^*(x) = \frac{(x_{j+1} - x)\ell(x_j) + (x - x_j)\ell(x_{j+1})}{x_{j+1} - x_j}, \qquad \text{for } x \in [x_j, x_{j+1}],$$

for $j = 1, \ldots, k-1$. For $x < x_1$ or $x > x_k$, $h^*(x)$ is set to $-\infty$. The squeeze function for $f(x)$ is thus $\exp h(x) = \exp\left(h^*(x)\right)$. Figure 7.8 shows the piecewise linear upper hull (in long dashes) and the piecewise linear lower hull (in short dashes).

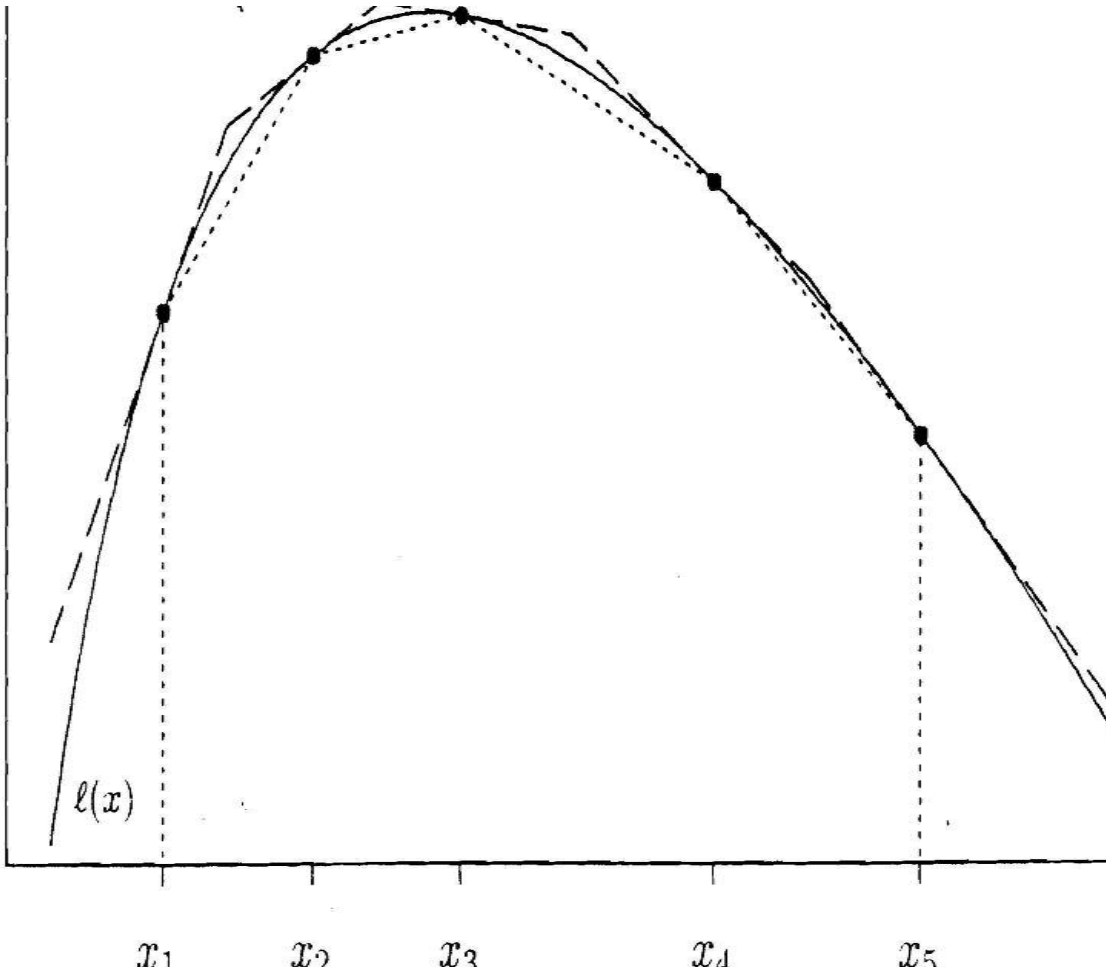

Figure 4: Piecewise linear outer and inner hulls for $\ell(x) = \log f(x)$ used in adaptive rejection sampling with k=5.

Although the hulls are linear, the majorizing function and the squeeze function are seen to be piecewise exponentials. In each step of the algorithm, the set S_k is augmented by the point generated from the distribution and k incremented by 1. Gilks and Wild(1992) describe a method where the evaluation of the derivatve ℓ' is not required. Thus the assumption of existence of continuous derivatives is not required for the *derivative-free method*. Instead of using tangents this method uses secants to obtain the majorizing function. Define $L_j(x)$ to be the straight line joining the points $(x_j, \ell(x_j))$ and $(x_{j+1}, \ell(x_{j+1}))$, for each $j = 1, \ldots, k-1$. Then define the piecewise linear function $g*(x)$ as

$$g_2^*(x) = \min\{L_{j-1}(x), L_{j+1}(x)\} \qquad \text{for } x \in [x_j, x_{j+1}],$$

with $g^*(x) = L_1(x)$ for $x < x_1$ and $g^*(x) = L_{k-1}(x)$ for $x > x_k$. Again concavity of ℓ ensures that $L_j(x)$ lies above $\ell(x)$ when $x < x_j$ or $x > xj + 1$ and below it otherwise. The rejection envelope based on chords is $\exp g(x) = \exp(g_2^*(x))$ This density is

The complete algorithm as in Gentle(2003) is given below. Note that the subscript k

in g_k and h_k refer to the number of point used in constructing squeeze function and the rejection envelope.

1. Initialize k and S_k.
2. Generate y from the normalized density g_k.
3. Generate u from $U(0,1)$ distribution.
4. If $u \leq \exp\left(h_k(y) - g_k(y)\right)$ then return X=y.

   Else

   If $u \leq \frac{f(y)}{\exp(g_k(y))}$ then return X=y.
5. Set $k = k+1$, add y to S_k and reconstruct g_k and h_k
6. Go to Step 1.

While the initial choice of the set S_k affects the computational efficiency of the algorithm, one would conjecture that these points placed more frequently in areas where $\ell()$ is more peaked. At any rate, the design of the algorithm will make sure that more points will be added in these regions in subsequent iterations.

`Example`:

`Adaptive Rejection Sampling from Beta(2,3) using the derivative-free envelope`

Three points $x = 0.2, 0.4$, and 0.7 were selected as the starting points of the set S_k and

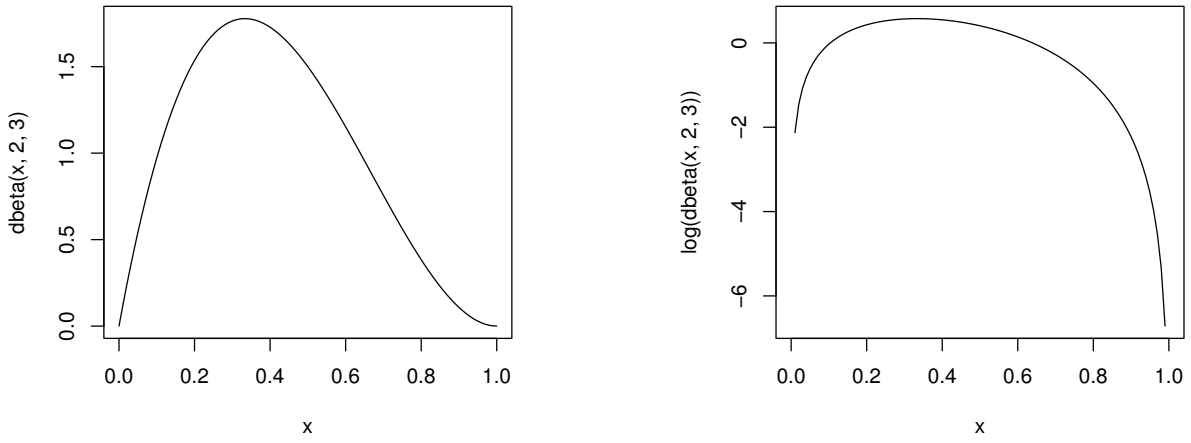

Figure 5: Density and the Log Density of Beta(2,3), respectively

$\log f(x)$ was evaluated at each point: $P_1 = (0.2, 0.4292)$, $P_2 = (0.4, 0.567)$ and $P_3 = (0.7, -0.28)$ The log-concavity of $f(x)$ was checked. Linear equations for the chords of

P_1P_2 and P_2P_3 are respectively, $y = 0.5889x + 0.3114$ and $y = 2.7556x + 1.6492$ Vertical lines are constructed at the extreme points,P_1 and P_3. The upper bound is created by extending the chords to their points of intersection with the vertical lines at 0.2 and 0.7. The piece-wise linear upper and lowerbounds were exponentiated. Thus the equation of the piece-wise envelope is given by

$$g(x) = \begin{cases} \exp(0.5889x + 0.3114) & 0 < x \leq 0.2, \\ \exp(-2.7556x + 1.6492) & 0.2 < x \leq 0.4, \\ \exp(0.5889x + 0.3114) & 0.4 < x \leq 0.7 \\ \exp(-2.7556x + 1.6492) & 0.7 < x \leq 1 \end{cases}$$

and the piece-wise squezing functions is given by

$$h(x) = \begin{cases} \exp(0.5889x + 0.3114) & 0.2 < x \leq 0.4 \\ \exp(-2.7556x + 1.6492) & 0.4 < x \leq 0.7 \end{cases}$$

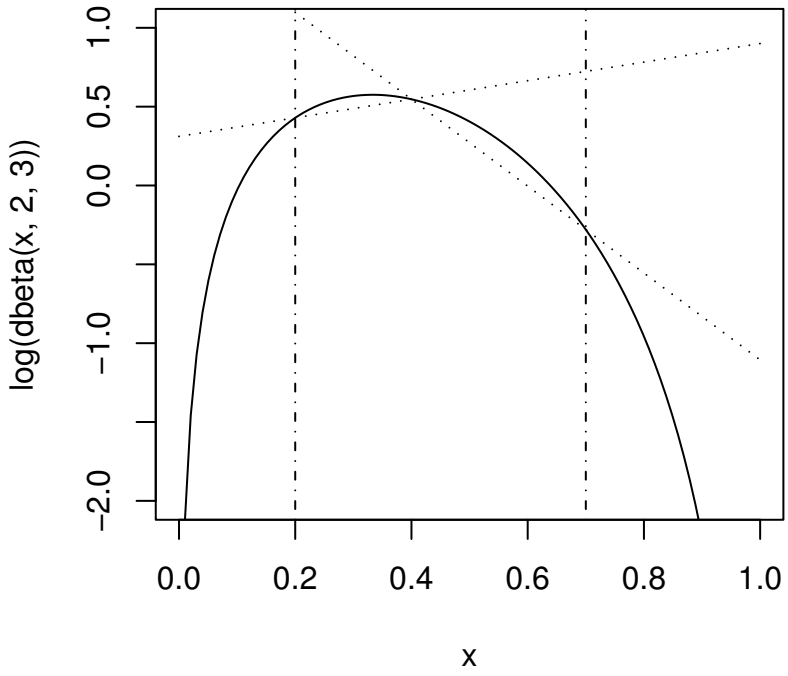

Figure 6: Upper and Lower Bounds for the Log Density

Samples were drawn from normalized density $g(x)$and $U(0, 1.$ If both the squeezing and rejection steps fails, the updating step is performed. The rejected point is included in the set S_k and relabeled in ascending order. Chords are then constructed to include the new point and the piece-wise linear upper and lower bounds are re-calculated.

8 Other Methods and Software

For generating standard normal random variates,a variety of rejection methods due to Marsaglia (1964), Marsaglia and Bray (1964) and Marsaglia, McLaren and Bray (1964)etc. are available. Some of these methods are described in Kennedy and Gentle(1980). In the NAG Library, the functions `g05fef` for Beta random variates and `g05fff` for Gamma random variates use rejection techniques. The ARS approach based on chords is available in WinBUGS software as a part of its options for carrying out MCMC algorithms for Bayesian analysis.

9 References

Atkinson, A. C. (1980) "Tests of pseudo-random numbers." *Applied Statistics*, 29, 164-171.

Devroye, Luc,(1986). *Non-Uniform Random Variate Generation*. Springer-Verlag: NY.

Fishman, G. S. (2006). *A Firts Course in Monte Carlo*. Thomson Brooks/Cole: Belmont, CA.

Gentle, J. E. (1998, 2003). *Random Number Generation and Monte Carlo Methods*. Springer-Verlag: NY.

Gilks, W.R.(1992) "Derivative-free adaptive rejectionsampling for Gibbs sampling". *Bayesian Statistics 4 Statistics*, Oxford University Press, Oxford, 641-649.

Gilks, W.R. and Wild (1992) "Adaptive rejectionsampling for Gibbs sampling". *Applied Statistics*, 41, 337-348.

Kennedy, W. J., Jr. and Gentle, J. E. (1980). *Statistical Computing*. Marcel Dekker: NY. (`Chapter 6`)

Kinderman, A. J., Monahan, J. F., and Ramage, J. G. (1977). "Computer Methods for Sampling from Student's t Distribution" *Mathematics of Computation*, 31, 1009-1018.

Knuth, D. E. (1981). *The Art of Computer Programming Volume 2: Seminumerical Algorithms (2nd Ed.)*. Addison-Wesley: Reading, MA. (`Chapter 3`)

Lange, K. (1998). *Numerical Analysis for Statisticians*. Springer-Verlag: NY. (`Chapter 20`)

L'Ecuyer, P. (1988) "Efficient and portable combined random number generators." *Comm. of the ACM*, 31, 742-749.

L'Ecuyer, P. (1998) Random Number Generation, Chapter 4 of the Handbook on Simulation, Jerry Banks Ed., Wiley, 1998, 93137.

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

Marsaglia, G. and Bray, T.A (1964) "A fast method for generating normal random variables." *Communications of the ACM*,7,4-10.

Marsaglia, G. MacLaren, M.D.,and Bray, T.A (1964) "A convenient method for generating normal variables." *SIAM Review*,6,260-264.

Marsaglia, G. and Zaman, A.(1991) "A new class of random number generators." *The Annals of Applied Probability*,1,462-480.

Matsumoto M. and Nishimura T., (1998) "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator." *ACM Transactions on Modeling and Computer Simulation*,8,1, 330.

Monahan, K. (2001). *Numerical Methods of Statistics*. Cambridge University Press: NY. (`Chapter 11`)

Ripley, B. D. (1987). *Stochastic Simulation*. Wiley: NY. (`Chapters 2 and 3`)

Schrage, L. E. (1979) "A more portable FORTRAN random number generator." *ACM Trans. Math. Software*, 5, 132-138.

Wichmann, B. A. and Hill, J. D. (1982) "An efficient and portable pseudo-random number generator." *Applied Statistics*, 31, 188-190. (`Algorithm AS183`)