R package `Ranlip` generating multivariate random variates from arbirtrary Lipschitz continuous distributions User Manual

Gleb Beliakov
`gleb@deakin.edu.au`

License agreement

Contents

Chapter 1

Introduction

This manual describes the programming library `ranlip`, which implements the method of acceptance/ rejection in the multivariate case, for Lipschitz continuous densities. It assumes that the Lipschitz constant of the density ρ is known, or can be approximated, and that computation of the values of ρ at distinct points is not expensive. The method builds a piecewise constant hat function, by subdividing the domain into hyperrectangles, and by using a large number of values of ρ. Lipschitz properties of ρ allow one to overestimate ρ at all other points, and thus to overestimate the absolute maxima of ρ on the elements of the partition.

The library `ranlip` implements computation of the hat function and generation of random variates, and makes this process transparent to the user. The user needs to provide a method of evaluation of ρ at a given point, and the number of elements in the subdivision of the domain, which is the parameter characterising the quality of the hat function and the number of computations at the preprocessing step.

The class of Lipschitz-continuous densities is very broad, and includes many multimodal densities, which are hard to deal with. No other properties beyond Lipschitz continuity are required, and the Lipschitz constant, if not provided, can be estimated automatically. The algoritm does not require ρ to be given analytically, to be differentiable, or to be normalised.

Section 1.1 describes theoretical background of this method of construction of the hat function. Chapter 2 describes the programming interfaceof `ranlip` library and provides a number of examples of its usage.

1.1 Theoretical background

1.1.1 Nonuniform random variate generation

Generation of nonuniform random variates is a common problem in such methods as Monte-Carlo simulation. While a large number of efficient algorithms exists for specific distributions [3, 4], frequently the distribution is unknown at the design stage. Universal (or black box) methods have recently gained popularity [5, 6, 8], as they do not require the distrubution to be given a priori, and essentially use the same programming code for very large classes of densities. Moreover, the densities need not to be given explicitly, only an algorithm for calculating the value of ρ at a given point has to be available.

A number of techniques for the univariate case are already available [4,6]. Inversion and acceptance/ rejection methods are the two main approaches used. However inversion does not generalise for multivariate distributions. Special properties, like convexity, concavity, or log-concavity help design efficient algorithms [5–8], but at the same time limit them to unimodal distributions.

In this manual we describe an approach to generate multivariate nonuniform random variates for a very general class of Lipschitz-continuous densities on a compact set. We will rely on the acceptance/ rejection technique, which generalises well for multivariate case.

Problem of random variate generation

Let ρ be the density of the required distribution, given on a compact set $D \subset R^n$. The goal is to generate a sequence of random variates with the density ρ.

We will assume that the density ρ is Lipschitz continuous, i.e., there exists a constant M, such that

$$|\rho(x) - \rho(y)| \leq M||x - y||,$$

for all x and $y \in D$, where $||\cdot||$ is any norm. We call the smallest such M the Lipschitz constant of ρ, and denote the class of such densities $Lip(M)$. We will use l_∞ norm, in which

$$||x - y||_\infty = \max_{i=1,\dots,n} |x_i - y_i|.$$

For simplicity, assume that D is a hyperrectangle, given by

$$a_i \leq x_i \leq b_i, \; i = 1, \dots, n.$$

Other compact domains are treated by embedding them into a hyperrectangle, and rejecting the random variates that fall outside D at the generation step.

1.1.2 Acceptance/rejection

Acceptance/ rejection is a classical approach to nonuniform random variate generation, based on approximation of the density ρ from above with a multiple of another density g, called the hat function $h(x) = cg(x)$. If generation of random variates with the density g is easy, then the approach is to generate random variates with the density g, and then either accept or reject them based on the value of an independent uniform random number. The better approximation with the hat function is, the higher are the chances of acceptance, and hence the efficiency of the generator.

Since ρ may take a variety of shapes, it is common to subdivide the domain into small parts (elements of the partition), and use a simple and accurate hat function on each element of the partition. In this case of piecewise continuous hat function, we first randomly choose an element of the partition (using a discrete random variate generator), and then generate a random variate on this element using acceptance/ rejection. Subdivision allows one to obtain much more accurate hat functions and hence higher acceptance ratio. The algorithm is outlined below.

Acceptance/rejection algorithm for a piecewise hat function

Given a partition of D, $D_k, k = 1, \dots, K$, and a piecewise hat function $h(x) = h_k(x)$, if $x \in D_k$, generate random variates with density $\rho(x) < h(x)$

Step 1 Generate a discrete random variate $k \in \{1, \dots, K\}$, where the probability of choosing k is proportional to the integral $\int_{D_k} h_k(x)dx$.

Step 2 Generate an independent random variate X on D_k with density proportional to h_k, and an independent uniform random number $Z \in (0, 1)$.

Step 3 If $Zh_k(X) \leq \rho(X)$ then return X, else go to Step 1.

1.1.3 Building the hat function

We will use a piecewise constant hat function $h(x)$, which takes constant values h_k on the elements of the partition of the domain D. We partition D into hyperrectangles, because generation of uniform random variates on a hyperrectangle is particularly efficient. The total number of the elements of the partition has to be sufficiently large for h to be an accurate approximation of ρ from above. However, too large numbers of elements translate into long preprocessing time, thus a right balance has to be struck between preprocessing time and the quality of approximation.

To build the hat function, we will find an overestimate of the absolute maximum of ρ on each hyperrectangle D_k, and take this value as h_k. An overestimate of the absolute maximum will be found by using a large number of values of ρ and its Lipchitz constant in l_∞ norm.

Consider an n-dimensional hyperrectangle R with the vertices $x^m, m = 1, \dots, 2^n$. Let us evaluate $\rho(x)$ at these vertices and denote the obtained values by ρ^m. Our goal is to find the absolute maximum of any $\rho \in Lip(M)$ on R.

From the Lipschitz condition it follows that any $\rho \in Lip(M)$ must satisfy

$$\forall x \in R : \rho(x) \leq \rho^m + M||x - x^m||,\ m = 1, \dots 2^n,$$

from which we deduce

$$\forall x \in R : \rho(x) \leq \min_{m=1,\dots,2^n} s^m(x) = \min_{m=1,\dots,2^n} (\rho^m + M||x - x^m||).$$

We call functions $s^m(x) = \rho^m + M||x - x^m||$ the support functions of ρ.

Evidently, the absolute maximum of $S(x) = \min_{m=1,\dots,2^n} s^m(x)$ will be a safe overestimate of the absolute maximum of $\rho(x)$, and we can take $\max_{x \in R} S(x)$ as the value of the hat function on R. Thus our strategy is to consider every hyperrectangle D_k of the subdivision of D, and compute $h_k = \max_{x \in D_k} S(x)$ by using the values of $\rho(x)$ at its vertices.

Since we need to process a very large number of hyperrectangles for an accurate hat function, let us simplify computation of h_k, in order to obtain an explicit approximate solution to the optimisation problem

$$maximise \min_{m=1,\dots,2^n} s^m(x).$$

First, let us consider the following subsets, which partition the hyperrectangle R,

$$S_i^m = \{x \in R : s^m(x) = \rho^m + M|x_i - x_i^m|\},\ i = 1, \dots, n.$$

On each such subset, the function $s^m(x)$ is linear.

Clearly, $\cup_{i=1,\dots,n} S_i^m$, and the interiors of these sets do not intersect. Now consider the pairwise intersections

$$S_i^{pq} = S_i^p \cap S_i^q.$$

The collection of the sets $S_i^{pq}, i = 1, \dots, n$, where pairs (p, q), $p, q \in \{1, \dots 2^n\}$, correspond to those vertices of R that share a common edge, forms an overlapping partition of R (i.e., $\cup S_i^{pq} = R$).

Since

$$\forall x \in R : \min_{m=1,\dots,2^n} s^m(x) \leq \min\{s^p(x), s^q(x)\}, \forall p, q \in \{1, \dots, 2^n\},$$

$$\max_{x \in S_i^{pq}} \min_{m=1,\dots,2^n} s^m(x) \leq \max_{x \in S_i^{pq}} \min\{s^p(x), s^q(x)\}.$$

Further,

$$\max_{x \in R} \min_{m=1,\dots,2^n} s^m(x) = \max_{\forall S_i^{pq}} \{ \max_{x \in S_i^{pq}} \min_{m=1,\dots,2^n} s^m(x)\}.$$

Hence we arrive to an overestimate

$$\max_{x \in R} \min_{m=1,\dots,2^n} s^m(x) \leq \max_{\forall S_i^{pq}} \{ \max_{x \in S_i^{pq}} \min\{s^p(x), s^q(x)\}\}.$$

The advantage of using expression on the right, is that $\max_{x \in S_i^{pq}} \min\{s^p(x), s^q(x)\}$ is easily found explicitly. Notice that the only pairs p, q that yield subsets S_i^{pq} from our collection, are the vertices of the hyperrectangle R that share the same edge. Then on the subset S_i^{pq} we have

$$\min\{s^p(x), s^q(x)\} = \min\{\rho^p + M|x_i - x_i^p|, \rho^q + M|x_i - x_i^q|\}.$$

Assume $x_i^p < x_i^q$. Because $\forall x \in S_i^{pq} : x_i^p \leq x_i \leq x_i^q$, we have

$$\min\{s^p(x), s^q(x)\} = \min\{\rho^p + M(x_i - x_i^p), \rho^q + M(-x_i + x_i^q)\}.$$

It is easy to show that the minimum is achieved at $x_i^* = \frac{x_i^p + x_i^q}{2} + \frac{\rho^q - \rho^p}{2M}$, and its value is $\frac{\rho^q + \rho^p}{2} + \frac{M(x_i^q - x_i^p)}{2}$. Thus we have

$$\max_{x \in R} \rho(x) \leq \max_{x \in R} \min_{m=1,\dots,2^n} s^m(x) \leq \max_{\forall S_i^{pq}} \{ \frac{\rho^q + \rho^p}{2} + M\frac{|x_i^q - x_i^p|}{2}\}. \qquad (1.1)$$

The right hand side of the above inequality is used in `ranlip` to overestimate the absolute maximum of $\rho(x)$ on each D_k.

Notice that an n-dimensional hyperrectangle has $n2^{n-1}$ edges, and this is how many sets S_i^{pq} are in the partition of D_k. Thus after we have computed 2^n values of ρ^m for each D_k, we need $n2^{n-1}$ comparisons to compute h_k.

In order to improve the quality of approximation on each D_k, we may further subdivide it into smaller hyperrectangles, apply Eq.(1.1) to each of these subsets, and then take the maximum as h_k. Of course, we could have simply increased the number of D_k, using the same number of computations. However from the practical point of view it may be counterproductive to have a very large partition of D, as the tables for the discrete random variate generator have limitations on their length. Thus it makes sense to have a partition of a reasonable size, but use a finer partition to improve the accuracy of the overestimate h_k. In `ranlip` the user has control over the size of both rough and fine partitions and may choose not to use the fine partition.

In the above formulae it is assumed that the Lipschitz constant of ρ in l_∞ norm, M, is known. This value is easily interpreted for differentiable densities as the largest value of the partial derivatives of ρ, but it also has a meaning for non-differentiable densities. The value of M can be safely overestimated by the user, but at a cost of less accurate hat function (and slower generation step).

It is possible to automatically estimate the Lipschitz constant by comparing the values ρ^m. Thus it makes sense to include this optional step into the computational algorithm. One has to be aware that automatic estimation of the value of M gives an *underestimate, not an overestimate* of M. There is a small chance that the actual value of M is larger then the estimate computed from a finite collection of function values. Hence it is desirable to use a priori information about the Lipschitz constant, if available.

Too low value of the chosen Lipschitz constant can be detected at the generation step (if $\rho(x) > h(x)$ for some x). This would mean, however, that the whole generation of the random sequence has to be repeated.

Note that for efficiency reasons, `ranlip` computes local estimates of the Lipschitz constant on the elements of the partition D_k, i.e., it uses different estimates of Lipschitz constants on different D_k. If the fine partition does not have enough elements, the estimate may not be accurate. The user can restrict local estimates to be no smaller than a given value.

Chapter 2

Description of the library

2.1 Installation

Installation of `Ranlip` package is standard: the user just needs to install the package from CRAN or from a local file

```
R CMD INSTALL ranlip.tar.gz
```

2.2 Description of the functions in package `Ranlip`

The method of building the `hat function`, and generation of random variates using acceptance/ rejection described in the previous section, have been implemented in a class library ranlip in c++ language, with an interface to R. All algorithms reside in the src folder in `ranlip.cpp` and `ranlip.h`.

The wrapper functions between `R` and `c++` are: `RcppExports.cpp` and `ranlipwrapper.cpp`

2.2.1 ranlip.Seed

Function of setting the seed

Function for setting the seed of the default uniform random number generator ranlux.

```
ranlip.Seed(seed)
```

2.2.2 ranlip.Init

Function for the initialization of the internal variables, it needs to be called before building the hat functions

`ranlip.Init(dim, left, right)`

Argument	Description
dim	The dimension
left	An array of size dim which determines the domain of ρ: $left_i \leq x_i \leq right_i$
right	An array of size dim which determines the domain of ρ: $left_i \leq x_i \leq right_i$

2.2.3 ranlip.RandomVec

Generates randoms variates with density ρ

Function for generating a random variate with density ρ. It should be called after ranlip.PrepareHatFunctionAuto() or ranlip.PrepareHatFunction().
`ranlip.RandomVec(Fn)`

Argument	Description
Fn	It is the function $\rho(x)$ where x is the array of size dim. It needs to be provided by the user coded in R, see examples.
output	Random vector of length *dim*.

2.2.4 ranlip.RandomVecN

Generates n randoms variates with density ρ

Function for generating n random variates with density ρ. It should be called after ranlip.PrepareHatFunctionAuto() or ranlip.PrepareHatFunction().
`ranlip.RandomVecN(100,Fn)`

Argument	Description
n	Number of random variates to generate
Fn	It is the function $\rho(x)$ where x is the array of size dim. It needs to be provided by the user coded in R, see examples.
output	Matrix of size $n \times dim$ of random vectors of length *dim*.

2.2.5 ranlip.PrepareHatFunction

Building the hat function

Function for building the hat function using Lipschitz constant and domain partition `ranlip.PrepareHatFunction(num, numfine, Lip, Fn)`

Argument	Description
num	The number of subdivisions in each variable to partition the Domain D into hyperrectangles D_k. On each D_k, the hat function will have a constant value h_k
numfine	The number of subdivisions in the finer partition in each variable. Each D_k is subdivided into $(numfine-1)^{dim}$ smaller hyperrectangles, in order to improve the quality of the overstimate h_k. nunmfine should be a power of 2 for numerical efficiency reason (if not, it will be automatically changed to a power of 2 larger than the supplied value) numdine can be 2, in which case the fine partition is not used
Lip	Lipschitz constant supplied
Fn	The density function $\rho(x)$ where x is the array of size dim.

2.2.6 ranlip.PrepareHatFunctionAuto

Building the hat function and estimates Lipschitz constant

Function for building the hat function and automatically computing an estimate to the Lipschitz constant.

`ranlip.PrepareHatFunctionAuto(num, numfine, minLip, Fn)`

Argument	Description
num	The number of subdivisions in each variable to partition the Domain D into hyperrectangles D_k. On each D_k, the hat function will have a constant value h_k
numfine	The number of subdivisions in the finer partition in each variable. Each D_k is subdivided into $(numfine-1)^{dim}$ smaller hyperrectangles, in order to improve the quality of the overstimate h_k. nunmfine should be a power of 2 for numerical efficiency reason (if not, it will be automatically changed to a power of 2 larger than the supplied value) numdine can be 2, in which case the fine partition is not used
minLip	Denotes the lower bound on the value of the computed Lipschitz constant, the default value is 0
Fn	The density function $\rho(x)$ where x is the array of size dim.
output	The estimate of the Lipschitz constant.

2.2.7 ranlip.SavePartition

Saves the computed hat function

Function for saving previously computed hat function to file name(string)
`ranlip.SavePartition(filename)`

Argument	Description
filename	The file name
output	0 if success, nonzero in case of error (1= hat function not computed, 2=file cannot be opned).

2.2.8 ranlip.LoadPartition

Load the computed hat function

Function for loading previously computed hat function from file name(string)
`ranlip.LoadPartition(filename)`

Argument	Description
filename	The file name
output	0 if success, nonzero in case of error (2=file cannot be opened, 3= corrupted file, 4=memory not allocated.).

2.2.9 Distribution function

The distribution function Fn needs to be provided by the user. This function takes two parameters, the input x and the dimension dim. The distribution needs not be normalised.

Example: trivariate normal distribution (not normalised)

```
Fn <- function(x,dim){
  out <- exp(-(x[1]^2+x[2]^2+x[3]^2))
  return(out)
}
```

2.3 Examples

```
library("ranlip")
# dimension 2

dim <- 2

Fn <- function(x,dim){
    r<-x[1]*x[1]+x[2]*x[2]
    out <- exp(-( (x[1]+0.2)^2+(x[2]+0.1)^2)/1.1 )*(1-exp(-sqrt(r)))
    return(out)
}
```

```
left <- c(-2,-2)
right <-  c(2,2)
num <- 20
numfine <- 4
MinLip <- 10

ranlip.Init(dim, left, right)

Lipconst<- ranlip.PrepareHatFunctionAuto(num, numfine, MinLip, Fn)
print(Lipconst)

r<-ranlip.RandomVec(Fn)
print(r)
rv<-ranlip.RandomVecN(1000, Fn)
plot(rv[,1],rv[,2],cex=0.5)

ranlip.FreeMem()

left <- c(-2,-2)
right <-  c(2,4)
ranlip.Init(dim, left, right)
Fn1 <- function(x,dim){
   out <- exp(-(x[2]-x[1]^2)^2 - (x[1]^2+x[2]^2)/2 )
   return(out)
}

ranlip.PrepareHatFunctionAuto(num, numfine, MinLip, Fn1)

rv<-ranlip.RandomVecN(10000, Fn1)
plot(rv[,1],rv[,2],cex=0.2)
ranlip.FreeMem()
```

2.4 Where to get help

The software library `Ranlip` and its components, are distributed by G.Beliakov AS IS, with no warranty, explicit or implied, of merchantability or fitness for a particular purpose. G.Beliakov, at his sole discretion, may provide advice to registered users on the proper use of `Ranlip` and its components.

Any queries regarding technical information, sales and licensing should be directed to `gleb@deakin.edu.au`. I am interested to learn about your experiences using `Ranlip` , bugs, suggestions, its usefulness, applying it in practice and so on.

If you want to cite `Ranlip` package, use references [1, 2].

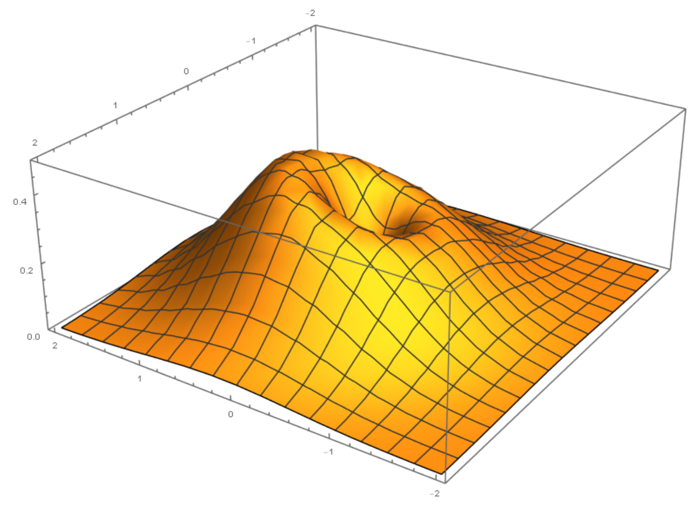

Figure 2.1: Distribution 1 used in the example $\rho(x,y) = \exp(-(r+a)^2/b)(1-\exp(-|r|))$, $r^2 = x^2 + y^2$, $a = (0.2, 0.1), b = 1.1$.

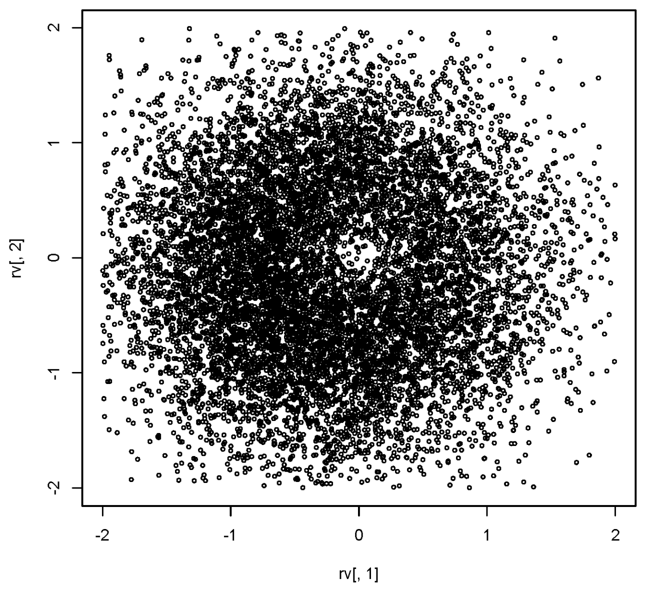

Figure 2.2: Random vectors generated with Distribution 1.

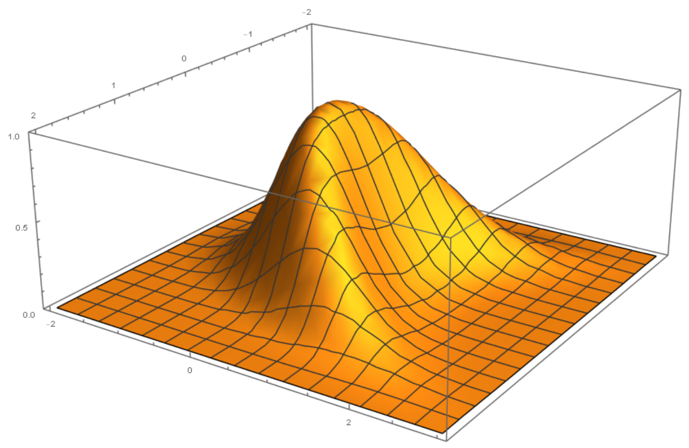

Figure 2.3: Distribution 2 used in the example $\rho(x, y) = \exp(-(y - x^2)^2 - \frac{x^2+y^2}{2})$

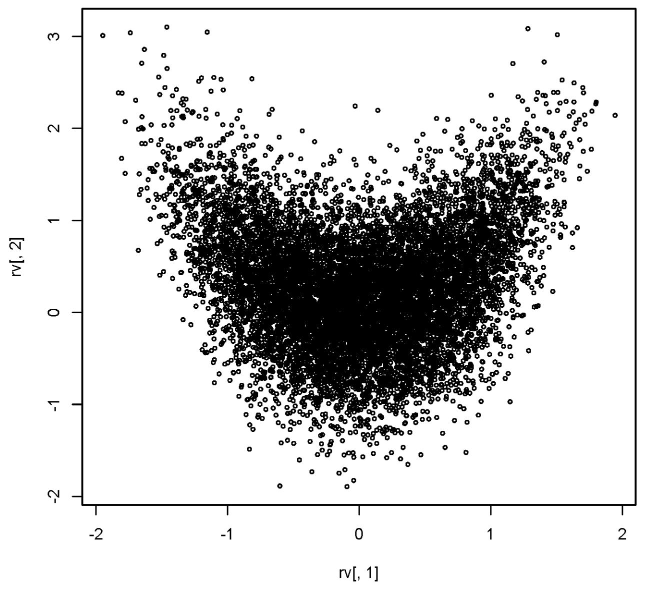

Figure 2.4: Random vectors generated with Distribution 2.

Bibliography

[1] G. Beliakov. Class library ranlip for multivariate nonuniform random variate generation. *Computer Physics Communications*, 170:93–108, 2005.

[2] G. Beliakov. Universal nonuniform random vector generator based on acceptance-rejection. *ACM Transactions on Modeling and Computer Simulation*, 15:205–232, 2005.

[3] J. Dagpunar. *Principles of Random Variate Generation.* Clarendon Press, Oxford, 1988.

[4] L. Devroye. *Non-uniform Random Variate Generation.* Springer Verlag, New York, 1986.

[5] W. Hörmann. A rejection technique for sampling from t-concave distributions. *ACM Transactions on Mathematical Software*, 21:182–193, 1995.

[6] W. Hörmann, J. Leydold, and G. Derflinger. *Automatic Nonuniform Random Variate Generation.* Springer, Berlin, 2004.

[7] J. Leydold and W. Hörmann. A sweep-plane algorithm for generating random tuples in simple polytopes. *Mathematics of Computation*, 67:1617–1635, 1998.

[8] J. Leydold and W. Hörmann. Universal algorithms as an alternative for generating non-uniform continuous random variates. In G.I. Schuëler and P.D. Spanos, editors, *Monte Carlo Simulation*, pages 177–183. A. A. Balkema, Rotterdam, 2001.