Maple toolbox RANLIP for multivariate nonuniform random variate generation. Version 1.0. User Manual

Gleb Beliakov
gleb@deakin.edu.au

Summary

This manual describes generation of nonuniform random variates from arbitrary Lipschitz-continuous distributions in Maple environment by using `RanLip` toolbox.

`RanLip` uses acceptance/ rejection approach, which is based on approximation of the probability density function from above with a "hat" function. It is assumed that the distribution has Lipschitz-continuous density, which is either given analytically or as a black box. The algorithm builds a piecewise constant hat function, using a large number of its values and subdivision of the domain into hyperrectangles.

`RanLip` provides very fast preprocessing and generation times, and yields small rejection constant, which is a measure of efficiency of the generation step. It exhibits good performance for up to five variables, and provides the user with a black box nonuniform random variate generator for a large class of distributions, in particular, multimodal distributions.

License agreement

Contents

1 Introduction **5**
1.1 Quick start . . . 8

2 Theoretical background **9**
2.1 Nonuniform random variate generation . . . 9
2.2 Acceptance/rejection . . . 10
2.3 Building the hat function . . . 10
2.4 Algorithms . . . 13
2.5 Random number generators used . . . 14

3 Maple interface **15**
3.1 Installation . . . 15
3.2 `RanLip` functions . . . 15
3.3 Examples . . . 18
3.4 Numerical performance . . . 19

4 More information **23**
4.1 Where to get help . . . 23
4.2 How to cite `RanLip` . . . 23

Bibliography **23**

Chapter 1

Introduction

Simulation in science relies on efficient methods of random variate generation. While generators of uniform random numbers are available in most programming languages/libraries, and methods of generation of nonuniform random variates for many specific distributions are thoroughly documented [3, 4, 5, 7], many practical situations require generation of random variates with unusual distributions, specific to the problem at hand. Universal random variate generators allow the user to specify the probability density function (as a formula, or even as a black box subroutine), and to use the same program code to generate random variates from this distribution. On the one hand, universal generators free the user from the need to devise a special generator for the required distribution, which is a challenging task in itself, and on the other hand are no less efficient than the specialized generators [7].

Acceptance/rejection is a standard technique used in the universal nonuniform random variate generators [4, 5, 7]. It relies on approximation of the probability density function from above using a "hat" function, and using the hat function to generate random variates, which are then either accepted or rejected. The method generalises well for multivariate distributions, however building an accurate hat function is very challenging in the multivariate case.

This manual describes a method of building accurate hat functions in the multivariate case, and an implementation of the method of acceptance/ rejection for multivariate distributions with Lipschitz-continuous densities. This method was described in [1, 2]. It assumes that the Lipschitz constant of the density ρ is known, or can be approximated from the data, 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 main purpose of `RanLip` toolbox is to provide Maple users with a sufficiently simple tool to rapidly generate non-uniform random variates from a non-standard distribution, for example the ones illustrated on Figs. 1.1-1.3.

While specialized methods are typically more efficient, development of such methods for non-standard distributions requires significant effort.

`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 characterizing the quality of the hat function and the computational complexity of 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 algorithm does not require ρ to be given analytically, to be differentiable, or to be normalized.

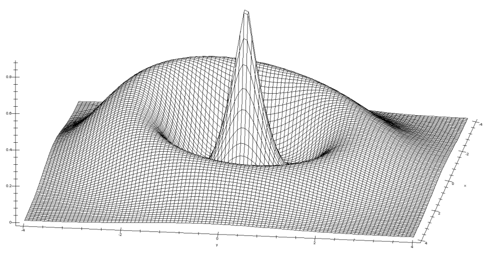

Figure 1.1: The graph of a polynomial-normal density. The density is given by $\rho(r) = (|r| - 1)^2 \exp(-\frac{|r+(0.2,0.2)|^2}{3})$, where $r = (x, y)$.

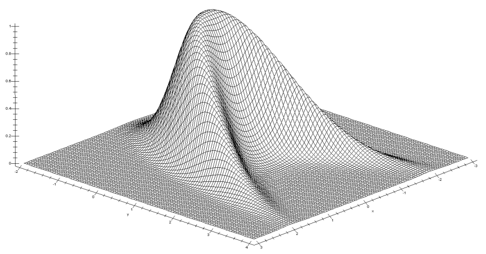

Figure 1.2: The graph of the density $\rho(x,y) = k\exp(-(y-x^2)^2 - \frac{x^2+y^2}{2})$.

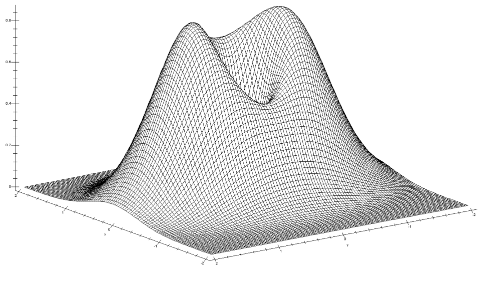

Figure 1.3: The graph of the density of a product of double exponential and normal distributions $\rho(r) = k\exp(-(r+a)^2/b) \times \exp(-|r|)$, $a = (0.2, 0.1)$, $b = 1.1$.

1.1 Quick start

Let us generate random variates from the following density

$$\rho(x,y) = k\exp(-(y-x^2)^2 - \frac{x^2+y^2}{2}),$$

k is the normalization constant (we do not need to supply it), see Fig. 1.2. We need to define the function f, set the number of variables and the boundaries of the domain, in this case $[-2,4]\times[-3,3]$, compute the hat function, and we are ready to use the generator.

Because preparation of the hat function can be expensive, we can save it in a file for later use, and then simply load this file.

Note that calls to different methods of RanLip have to be executed after an appropriate function has been declared in Maple.

```
f := proc( x, y)  # user's density function
        exp(-(y-x^2)^2 - (x^2+y^2)/2);
    end proc;

  RANLIPPATH:="mapleranlip.dll":
  PrepareHatFunction := define_external('MWRAP_PrepareHatFunction','MAPLE',
            LIB=RANLIPPATH):
  RandomVec := define_external('MWRAP_RandomVecRanLip','MAPLE', LIB=RANLIPPATH):
  Seed := define_external('MWRAP_SeedRanLip','MAPLE', LIB=RANLIPPATH):
  FreeMem := define_external('MWRAP_FreeMemRanLip','MAPLE', LIB=RANLIPPATH):

 Lo:=Vector(1..2,datatype=float[8],[-2,-3]):
 Up:=Vector(1..2,datatype=float[8],[4,3]):
 xrand:=Vector(1..2,datatype=float[8]);  # will receive generated random vector

 PrepareHatFunction(2,Lo, Up,f, 50,16,10);
     # will use (50*15)^2 evaluations of f
 Seed(10); # whatever integer
 # Generate a single random vectors
 RandomVec(xrand); xrand;

 # Generate many random vectors
 M:=3000:
 XR:=Matrix(1..M,1..2,datatype=float[8],order=C_order);
 RandomVec(XR,M);

#  We can plot them
 with(plots):
 pointplot({seq([XR[i,1],XR[i,2]],i=1..M)},symbol=point);

 FreeMem(); # destroys the hat function
```

Chapter 2

Theoretical background

2.1 Nonuniform random variate generation

Generation of nonuniform random variates is a common problem in such methods as Monte-Carlo simulation. A large number of efficient algorithms exists for specific distributions [3, 4, 5]. But frequently the distribution is unknown at the design stage. Universal (or black box) methods have recently gained popularity [6, 9, 7], 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, 7]. Inversion and acceptance/ rejection methods are the two main approaches used. However inversion does not generalize for multivariate distributions. Special properties, like convexity, concavity, or log-concavity help design efficient algorithms [7, 6, 9, 8], but at the same time limit them to a subset of 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 generalizes 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

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

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

$$a_i \le x_i \le b_i,\ i = 1, \ldots, n.$$

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

2.2 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, $D_k, k = 1, \ldots, K$, 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 $k \in \{1, \ldots, K\}$, 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 $Zh_k(X) \le \rho(X)$ then return X, else go to Step 1.

2.3 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 optimization 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

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

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

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

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

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 $\forall x \in S_i^{pq} : x_i^p \le x_i \le x_i^q$, 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

$$\max_{x\in R} \rho(x) \le \max_{x\in R} \min_{m=1,\dots,2^n} s^m(x) \le \max_{\forall S_i^{pq}}\{\frac{\rho^q + \rho^p}{2} + M\frac{|x_i^q - x_i^p|}{2}\}. \tag{2.1}$$

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.(2.1) 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. In other words, it treats ρ as a *locally Lipschitz* function. Of course Lipschitz functions are also locally Lipschitz. However, since the Lipschitz constant is a global parameter, it may happen that a sharp peak of ρ in one part of the domain makes this global parameter too large for other parts, which leads to computational inefficiency. Estimation of the Lipschitz constant locally makes the hat function a better approximation of ρ. The drawback is that if the fine partition does not have enough elements, this estimate may not be accurate. The user can restrict local estimates to be no smaller than a given value.

2.4 Algorithms

Random variate generation consists of two steps: preprocessing and generation. At the preprocessing step, the domain D is subdivided into hyperrectangles $D_k, k = 1, \ldots, K$, where $K = num^n$, each D_k is further subdivided into $(numfine - 1)^n$ smaller hyperrectangles, and then Eq. (2.1) is used to overestimate $\rho(x)$ on each element of the fine partition. The maximum of these values is taken as an overestimate h_k. Based on the volumes of D_k and the values h_k, the tables for the discrete random variate generator (we use the alias method) are created.

At the generation step, the values h_k and the tables for the alias method are used.

Preprocessing Algorithm

Purpose: Given the density $\rho(x)$ on $D = \{x \in R^n : a_i \leq x_i \leq b_i\}$, its Lipschitz constant M in l_∞-norm, build a piecewise constant hat function $h(x) = h_k$, if $x \in D_k$.

Input: The density $\rho(x)$, its domain D, Lipschitz constant M, and two parameters, num - the number of subdivisions of D with respect to each coordinate, and $numfine$ - the number of subdivisions of the fine partition.
Output: The hat function $h(x)$ and the tables for the alias method.

Step 1 Partition D into num^n hyperrectangles D_k.

```
Step 2 For each $D_k$ do:
    2.1 Partition $D_k$ into $(numfine - 1)^n$ hyperrectangles $R_j$
    2.2 Evaluate $\rho(x)$ at the vertices of each $R_j$
    2.3 Compute the overestimate $s_j$ using Eq.(2.1)
    2.4 Set the overestimate $h_k = \max_j s_j$
Step 3 Compute the volumes of $D_k$
Step 4 Build the tables for the alias method using $Volume(D_k) \times h_k$ as the vector
       of probabilities
```

Generation Algorithm

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

Input: The hat function $h(x)$ and the tables for the alias method, density $\rho(x)$.
Output: Random variate X.

```
Step 1 Generate a discrete random variate $k \in \{1, \dots, K\}$ using alias method.
       The probability of choosing $k$ is proportional to the integral $h_k \int_{D_k} dx$.
Step 2 Generate an independent random variate $X$ uniform on $D_k$, and an inde-
       pendent uniform random number $Z \in (0, 1)$.
Step 3 If $Zh_k \leq \rho(X)$ then return $X$, else go to Step 1.
```

2.5 Random number generators used

The code in `ranlip` includes a uniform random number generator `ranlux` by M.Luescher [10] (with the period 10^{171}), and the discrete random variate generator based on the *alias* method by A. Walker [11]. Both methods are taken from the `GSL` library `http://www.gnu.org/software/gsl/` in the public domain.

Chapter 3

Maple interface

3.1 Installation

Installation of `RanLip` is very simple. Just copy the file `mapleranlip.dll` into your working directory (or any directory on the PATH), and you are ready to use it. The examples in the subsequent sections illustrate the use of `RanLip`.

3.2 `RanLip` functions

Calling `RanLip` functions should be done after declaring interface to the methods in `RanLip` using the commands

```
RANLIPPATH:="mapleranlip.dll":
   # if you need to provide full path, use, e.g.,
   #     RANLIPPATH:="c:/work/maple/mapleranlip.dll":
 PrepareHatFunction :=define_external('MWRAP_PrepareHatFunction','MAPLE', LIB=RANLIPPATH):
 PrepareHatFunctionAuto:=define_external('MWRAP_PrepareHatFunctionAuto','MAPLE',LIB=RANLIPPATH):
 RandomVec := define_external('MWRAP_RandomVecRanLip','MAPLE',LIB=RANLIPPATH):
 SavePartition :=define_external('MWRAP_SavePartitionRanlip','MAPLE',LIB=RANLIPPATH):
 LoadPartition :=define_external('MWRAP_LoadPartitionRanLip','MAPLE', LIB=RANLIPPATH):
 SetDistFunction :=define_external('MWRAP_SetDistFunctionRanLip','MAPLE', LIB=RANLIPPATH):
 Seed := define_external('MWRAP_SeedRanLip','MAPLE', LIB=RANLIPPATH):
 GetSeed := define_external('MWRAP_GetSeedRanLip','MAPLE',LIB=RANLIPPATH):
 FreeMem :=define_external('MWRAP_FreeMemRanLip','MAPLE',LIB=RANLIPPATH):
 Count_total := define_external('MWRAP_Count_totalRanLip','MAPLE',LIB=RANLIPPATH):
 Count_error := define_external('MWRAP_Count_errorRanLip','MAPLE',LIB=RANLIPPATH):
 Lipschitz := define_external('MWRAP_LipschitzRanLip','MAPLE',LIB=RANLIPPATH):
```

Note that these commands have been set in the example Maple worksheet, and can be simply copied from there. The name on the left can be changed by the user.

The main parameters that need to be supplied to `RanLip` are the name of the user's density function, the number of variables, the coordinates of the left and right corners of the rectangular domain, the number of hyperrectangles to subdivide the domain, and an estimate of the Lipschitz constant. The latter can be computed automatically, but the user's guess will improve calculations.

`PrepareHatFunction(dim, Left, Right, Dist, num, numfine, Lip)`
Builds the hat function, using the Lipschitz constant supplied in `Lip`. The parameters are:

- integer *dim* – the number of variables
- array *Left* – array of size dim, the coordinates of the left corner of the domain
- array *Right* – array of size dim, the coordinates of the right corner of the domain
- Dist – the address of the user's distribution density function
- integer *num* – the number of subdivisions in each variable to partition the domain
- integer *numfine* – the number of subdivisions in the finer partition in each variable, should be > 1, and a power of 2 for numerical efficiency reasons (if not, it will be automatically changed to a power of 2 larger than the supplied value). There will be in total $(num * numfine)^{dim}$ evaluations of *Dist* (calls to user's function). *numfine* can be 2, in which case the fine partition is not used
- double *Lip* – an (over) estimate of the Lipschitz constant of the density.

`PrepareHatFunctionAuto( dim, Left, Right, Dist, num, numfine, minLip)`
Builds the hat function, and automatically computes an estimate to the Lipschitz constant. The parameters are:

- integer *dim* – the number of variables
- array *Left* – array of size dim, the coordinates of the left corner of the domain
- array *Right* – array of size dim, the coordinates of the right corner of the domain
- Dist – the address of the user's distribution density function
- integer *num* – the number of subdivisions in each variable to partition the domain
- integer *numfine* – the number of subdivisions in the finer partition in each variable, should be > 1, and a power of 2 for numerical efficiency reasons (if not, it will be automatically changed to a power of 2 larger than the supplied value). There will be in total $(num * numfine)^{dim}$ evaluations of *Dist* (calls to user's function). *numfine* can be 2, in which case the fine partition is not used

- double *minLip* – Optional: the specified lower bound on the estimate of the Lipschitz constant of the density. It can be 0, if not, the algorithm will ensure that the calculated Lipschitz constant is not smaller than *minLip*

`RandomVec(x)`
`RandomVec(x,m)`

Generates a random variate with the density *Dist*. **Should be called after `PrepareHatFunction` or `PrepareHatFunctionAuto` or `LoadPartition`;** `SetDistFunction`. This function returns an array of size *dim* in the output variable x. x has to be declared by the user as an array of size at least n, prior to this call as follows,

```
x:=Vector(1..2,datatype=float[8]);
```

m is an optional argument specifying how many random vectors to generate. Of course, in this case x has to be a two dimensional array as follows

```
XR :=Matrix (1..300,1..2,datatype=float[8],order=C_order);
 RandomVec(X,300)
```

`Seed(seed)`
Sets the seed of the default uniform random number generator `ranlux`.

`GetSeed()`
Returns the seed value used by the default generator `ranlux`

`SavePartition( fname)`
Saves the computed hat function into the file `fname`.

`LoadPartition(fname)`
Loads previously computed hat function from the file `fname`. `SetDistFunction(Dist)` has to be called before `RandomVec()`.

`SetDistFunction(Dist)`
Sets the address of the distribution function to *Dist*. Has to be called after `LoadPartition()`.

`FreeMem()`
Frees the memory occupied by the data structures, which can be very large. **It destroys the hat function, and `RandomVec` method cannot be called after `FreeMem`.**

`Lipschitz()`
Returns the Lipschitz constant computed in `PrepareHatFunctionAuto`.

`Count_total()`
Returns the total number of random variates generated by the algorithm (both accepted and rejected). It can be used for statistical purposes to estimate the acceptance ratio (*total_accepted/count_total*). It is reset to 0 in `PrepareHatFunction`, `PrepareHatFunctionAuto`, `Seed` and `LoadPartition` methods.

`Count_errors()`

Returns the number of points where $\rho(x) > h(x)$, i.e., where the hat function was computed incorrectly because of a too small Lipschitz constant. It indicates that the simulation using `ranlip` has to be repeated using a new hat function with a larger `Lip`. It is reset to 0 in `PrepareHatFunction`, `PrepareHatFunctionAuto`, `Seed` and `LoadPartition` methods.

3.3 Examples

A few examples have been provided in the Maple example worksheet. In the following example we will generate random variates with the density

$$\rho(r) = (|r| - 1)^2 \exp(-\frac{|r + (0.2, 0.2)|^2}{3}),$$

where $r = (x, y)$, on $[-4, 4]^2$, see the graph on Fig. 1.1.

We will first define the density $g = \rho$, then build the hat function and generate a few random vectors. Since building the hat function may be time consuming, we will save it into a file. Next time we need random variates with the same density, we load the hat function from the file, set the distribution function g, and we are ready to generate random vectors again.

```
g := proc( x, y)  # user's density function
       local r,r1:
       r:=(x^2+y^2)^(0.5):
       r1:= (x+0.2)^2+(y+0.2)^2:
      (r-1)^2*exp(-r1/3);
    end proc:

 #just define the boundaries and prepare arrays
 dim:=2:
 Lo:=Vector(1..dim,datatype=float[8],[-4,-4]):
 Up:=Vector(1..dim,datatype=float[8],[4,4]):
 xrand:=Vector(1..dim,datatype=float[8]):

 #generate the hat function
 PrepareHatFunctionAuto(dim,Lo, Up,g, 50,16,1);
 Seed(10);

 #generate random vectors
 RandomVec(xrand); xrand;

 M:=10000:
 XR:=Matrix(1..M,1..dim,datatype=float[8],order=C_order):
 RandomVec(XR,M);

 pointplot({seq([XR[i,1],XR[i,2]],i=1..M)},symbol=point,axes=boxed);
‘random numbers generated =‘;Count_total();
 ‘errors=‘;Count_error();
 SavePartition(‘hatfunction.txt‘);
 FreeMem();
 # destroys hat function

 # the Maple session may be closed and reopened
 LoadPartition(‘hatfunction.txt‘);
 SetDistFunction(g);  # do not forget set the distribution with this call
 #generate random vectors again
 RandomVec(xrand); xrand;
```

Of course, RanLip can also generate random numbers (uniform, or with any Lipschitz density). It turns out to be much faster than Maple's internal generator. The example below illustrates this.

```
f := proc( x)  # user's density function: normal distribution
        exp(-x^2/2);
    end proc;

    Lo:=Vector(1..1,datatype=float[8],[-100]):
    Up:=Vector(1..1,datatype=float[8],[100]):
    xrand:=Vector(1..1,datatype=float[8]);

 #generate the hat function
 PrepareHatFunction(1, Lo, Up,f, 10000,8,1);
 Seed(10);

 M:=100000:
 XR:=Vector(1..M,datatype=float[8],order=C_order):
 RandomVec(XR,M);
```

3.4 Numerical performance

We tested RanLip with Maple release 9.0.1, on a Pentium IV 2.1 GHz workstation. Computation of the hat function depends on the dimension, and parameters *num* and *numfine*. The tables below indicates average running times.

For comparison, Maple's internal generators are much slower: one generation of a uniform random number on $(0,1)$ took 8.8×10^{-6} sec using Maple's `rand()` function. It takes 150×10^{-6} using `stats[random, uniform[0,1]](1000000):` command. Generation of normally distributed random numbers also takes 150×10^{-6} per number using `stats[random,normald](1000000)`. For other distributions the performance is worse (e.g., for lognormal distribution generation time is 4700×10^{-6} per random number).

Using RanLip to generate normally distributed random numbers takes 0.05 sec to prepare the hat function, and then under 3.0×10^{-6} to generate one random number. For lognormal distribution, similarly, 0.17 preparation and 6.2×10^{-6} generation time.

Example	num	numfine	preprocessing time (s)	generation time (s $\times 10^{-6}$)
Example 2	20	8	0.015	10.5
	20	16	0.11	8.3
	20	32	0.39	5.7
	20	64	1.55	5.2
	80	8	0.04	5.2
	80	16	0.15	4.6
	80	32	6.20	4.0
	80	64	24.8	3.8

Table 1. Performance of the algorithms from `ranlip` on the example illustrated on Fig.2. The Lipschitz constant was automatically computed by the algorithm for each element of the partition D_k.

n	num	numfine	preprocessing time (s)	generation time (s $\times 10^{-6}$)
2	20	8	0.015	10.5
	20	16	0.11	8.3
	20	32	0.39	5.7
	20	64	1.55	5.2
	80	8	0.04	5.2
	80	16	0.15	4.6
	80	32	6.20	4.0
	80	64	24.8	3.8
3	10	8	0.56	35
	10	16	4.5	20
	10	32	35	13
	10	64	285	10
	20	4	0.62	39
	20	8	4.5	20
	20	16	35	12
	20	32	285	9
4	10	4	3.1	162
	10	8	49	75
	10	16	790	40

Table 2. Performance of the algorithm from `ranlip` as a function of the dimension and partition size. For comparison, one generation of a uniform random number on $(0, 1)$ took 8.8×10^{-6} sec using Maple's `rand()function` .

n	num	numfine	preprocessing time (s)	generation time (s $\times 10^{-6}$)
2	20	8	0.03	5.6
	20	16	0.10	4.6
	20	32	0.43	4.4
	20	64	1.7	4.4
	80	4	0.14	3.8
	80	8	0.44	3.8
	80	16	1.7	3.7
3	10	8	0.67	9.2
	10	16	5.1	8.5
	10	32	41	8.1
	10	64	330	7.7
	20	4	0.65	6.7
	20	8	5.0	6.1
	20	16	40	5.7
	20	32	310	5.2
4	10	4	3.5	17
	10	8	57	14
	10	16	908	12
	10	32	15510	11
5	10	2	2.3	100
	10	4	150	31
	10	8	4910	25
	20	2	83	25
	20	4	5601	16

Table 3. Performance of the algorithm `PrepareHatFunctionAuto()` from `ranlip` as a function of the dimension and partition size. The Lipschitz constant was automatically computed by the algorithm for each element of the partition D_k.

Chapter 4

More information

4.1 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`.

The web page(s) for `RanLip` are
`http://www.deakin.edu.au/~gleb/ranlip.html`
`http://www.it.deakin.edu.au/~gleb/ranlip.html`

4.2 How to cite `RanLip`

Please use the following references:

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

G. Beliakov. Universal nonuniform random vector generator based on acceptance-rejection. *ACM Trans. on Modelling and Computer Simulation*, 15:205–232, 2005.

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 Trans. on Modelling 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] J.E. Gentle. *Random number generation and Monte Carlo methods.* Springer, New York, 2003.

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

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

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

[9] 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.

[10] M. Luescher. A portable high-quality random number generator for lattice field theory calculations. *Computer Physics Communications*, 79:100–110, 1994.

[11] A.J. Walker. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electron. Lett.*, 10:127–128, 1974.