This is the authors final peer reviewed version of the item published as:


Copyright : 2005, Elsevier B.V.
Class library ranlip for multivariate nonuniform random variate generation

Gleb Beliakov

School of Information Technology, Deakin University, 221 Burwood Hwy, Burwood 3125, Australia

CPC: 4.13 Statistical methods

PACS: 02.50.-r

Abstract

This paper describes generation of nonuniform random variates from Lipschitz-continuous densities using acceptance/rejection, and the class library ranlip which implements this method. It is assumed that the required distribution has Lipschitz-continuous density, which is either given analytically or as a black box. The algorithm builds a piecewise constant upper approximation to the density (the hat function), using a large number of its values and subdivision of the domain into hyperrectangles.

The class library ranlip provides very competitive 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. It will be valuable for researchers who frequently face the task of sampling from unusual distributions, for which specialized random variate generators are not available.

Key words: Nonuniform random variate generation; Acceptance/rejection; Multimodal distributions; Multivariate Lipschitz densities.

PROGRAM SUMMARY

Title of program: Ranlip

Catalogue number: (supplied by Elsevier)

Program obtainable from: CPC Program Library, Queen’s University of Belfast, N.

Preprint submitted to Elsevier Science 28 February 2005
Ireland (see application form in this issue)

Licensing provisions: none

Computer for which the program is designed and others on which it has been tested:
Computers: IBM PC; DEC Alpha
Installations:

Operating systems under which the program has been tested: Windows XP, Linux, Unix

Programming language used: C++

Memory required to execute with typical data: 32M

No. of bits in a word: 32

No. of processors used: 1

Has the code been vectorised?: No

Peripherals used:

No. of lines in distributed program, including test data, etc.:

CPC Program Library subprograms used:

Keywords: Nonuniform random variate generation; Acceptance/rejection; Multi-modal distributions; Simulation

Nature of physical problem
This program allows one to generate nonuniform random vectors from a variety of distributions (especially multimodal), using acceptance/rejection approach. Suitable for non-standard distributions for up to five variables.

Method of solution
Assuming Lipschitz continuous density, a tight overestimate (the hat function) is computed. Then random variates are generated using acceptance/rejection approach.

Restrictions on the complexity of the problem
If the number of variables exceeds five, computation of the overestimate is inefficient.

Typical running time
Preprocessing 1-5000 s, generation 5-20 ×10⁻⁶ s

Unusual features of the program

References
1 Introduction

Simulation in physics and chemistry 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 [1,2,5,7], 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 [2,5,7]. 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 paper 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. The programming library ranlip implements this method. It assumes that the Lipschitz constant of the density $\rho$ is known, or can be approximated from the data, and that computation of the values of $\rho$ 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 $\rho$. Lipschitz properties of $\rho$ allow one to overestimate $\rho$ at all other points, and thus to overestimate the absolute maxima of $\rho$ on the elements of the partition.

The main purpose of ranlip library is to provide the user with a sufficiently simple tool to rapidly generate non-uniform random variates from a non-standard distribution. While specialized methods are typically more efficient, development of such methods for non-standard distributions requires significant effort.

The class 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 $\rho$ 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 $\rho$ to be given analytically, to be differentiable, or to be normalized.

The remainder of the paper describes theoretical background of the method, analyses the computational complexity of the algorithm, describes the application programming interface (API) of ranlip library, and provides a number of examples of its usage. It also provides the results of numerical experiments, that indicate applicability of this approach to various problems. The Appendix gives details of the algorithms.

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 [1,2,5,3]. Devroye describes efficient generators from some less-known distributions implemented in one line of computer code [3]. 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 distribution 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 $\rho$ at a given point has to be available.

A number of techniques for the univariate case are already available [2,7]. 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 paper 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 $\rho$ be the density of the required distribution, given on a compact set $D \subset \mathbb{R}^n$. The goal is to generate a sequence of random variates with the density $\rho$.

We will assume that the density $\rho$ 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 $\rho$, and denote the class of such densities $\text{Lip}(M)$. We will use $l_\infty$ 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

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

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 $\rho$ 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 $\rho$ 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 \( \frac{1}{2} < h(x) \)*

**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 \( Zh_k(X) \leq \rho(X) \) 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 \( \rho \) 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 \( \rho \) 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 \( \rho \) and its Lipschitz constant in \( l_\infty \) norm.

Consider an \( n \)-dimensional hyperrectangle \( R \) with the vertices \( x^m, m = 1, \ldots, 2^n \). Let us evaluate \( \rho(x) \) at these vertices and denote the obtained values by \( \rho^m \).

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

\[
\forall x \in R : \rho(x) \leq \rho^m + M||x - x^m||, \quad m = 1, \ldots, 2^n,
\]

from which we deduce

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

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

Evidently, the absolute maximum of \( S(x) = \min_{m=1,\ldots,2^n} s^m(x) \) 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

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

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

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

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

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

$$S^{pq}_i = S^p_i \cap S^q_i.$$

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

Since

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

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

Further,

$$\max_{x \in R} \min_{m=1,\ldots,2^n} s^m(x) = \max\{\max_{S^{pq}_i} \min_{x \in S^{pq}_i} s^m(x)\}. $$

Hence we arrive to an overestimate

$$\max_{x \in R} \min_{m=1,\ldots,2^n} s^m(x) \leq \max\{\max_{S^{pq}_i} \min_{x \in S^{pq}_i} \{s^p(x), s^q(x)\}\}. $$

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

$$\min\{s^p(x), s^q(x)\} = \min\{\rho^p + M|x_i - x^p_i|, \rho^q + M|x_i - x^q_i|\}. $$
Assume \( x_i^p < x_i^q \). Because \( \forall x \in S_{pq}^i : x_i^p \leq x_i \leq x_i^q \), 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
\[
\max_{x \in \mathbb{R}} \rho(x) \leq \max_{x \in \mathbb{R}} \min_{m = 1,...,2^n} s^m(x) \leq \max_{\forall S_{pq}^i} \left\{ \frac{\rho^p + \rho^q}{2} + \frac{M|x_i^q - x_i^p|}{2} \right\}. \tag{1}
\]
The right hand side of the above inequality is used in \texttt{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 \( S_{pq}^i \) are in the partition of \( D_k \). Thus after we have computed \( 2^n \) values of \( \rho^m \) 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) 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 \texttt{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 \( \rho \) in \( l_\infty \) norm, \( M \), is known. This value is easily interpreted for differentiable densities as the largest value of the partial derivatives of \( \rho \), 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 \( \rho^m \). 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 \textit{underestimate}, not an \textit{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 $\rho$ 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 $\rho$ 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 $\rho$. 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.

3 Description of the library ranlip

3.1 Programming interface

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. The main class which provides the interface to the preprocessing and random variate generation is called CRanLip. This is an abstract class, from which the user must derive his own class which overrides the method for computing $\rho(x)$, and declare an instance of that class, like in the example below

```c++
// declare a derived class MyRnumGen, with one method to override
class MyRnumGen:public CRanLip {
    public: virtual double Distribution(double* p) ;
};

double MyRnumGen::Distribution(double* p) { // example: multivariate normal distribution
    double r;
    for(int j=0;j<Dimension;j++) {
        r+=p[j]*p[j];
    }
    return exp(-r);
}
// declare an instance of this class
MyRnumGen MyGen;
```
Examples below illustrate this interface. There is an alternative procedural interface, declared in `ranlipproc.h`. Functions declared in this file simply pass execution to the relevant members of the class CRanLip. The names of the functions coincide with the names of the methods of CRanLip, but have "RanLip" suffix. The user must provide an implementation of the function which computes the desired distribution, using the following argument list `MyDist(double* p, int dim)`, and pass its address using `SetDistFunctionRanLip(&MyDist)`. Example 2 below illustrates the usage of the procedural interface.

The essential members of the class CRanLip are the following

```cpp
class CRanLip { public:
    // initialises the tables and arrays for the generator
    // should be the first method to call
    void Init(int dim, double* left, double* right);

    // sets the pointer to the uniform random number generator.
    // The default is ranlux generator
    void SetUniformGenerator(UFunction gen);

    // sets the seed for the uniform random number generator
    void Seed(int seed);

    // computes the hat function given the Lipschitz constant
    // num and numfine determine the rough and fine partitions
    void PrepareHatFunction(int num, int numfine, double Lip);

    // computes the hat function, and automatically computes
    // the Lipschitz constant
    void PrepareHatFunctionAuto(int num, int numfine, double minLip);

    // generates a random variate with the required density
    void RandomVec(double* p);

    // frees the memory occupied by the partition and various tables
    void FreeMem();
    ...
}
```
**Init(int dim, double* left, double* right)**

Initialises the internal variables of this class. `dim` is the dimension, `left` and `right` are arrays of size `dim` which determine the domain of \( \rho \): \( \text{left}_i \leq x_i \leq \text{right}_i \). **Init** must be called only once before any other method.

**SetUniformGenerator(UFunction gen)**

Sets a pointer to the uniform random number generator on \((0, 1)\). The default is M. Luescher’s ranlux generator, but the user can override it and use his own preferred generator. The prototype for this function is `double Generator()`.

**Seed(int seed)**

Sets the seed of the default uniform random number generator `ranlux`. If the user has supplied his own generator in `SetUniformGenerator`, that generator’s `seed()` function should be called instead.

**virtual double Distribution(double* x)**

This method must be declared and implemented by the user. It is supposed to return the value of \( \rho(x) \) for a given \( x \). The parameter \( x \) is an array of size `dim`. The member variable `Dimension` contains the value of `dim`.

**PrepareHatFunction(int num, int numfine, double Lip)**

Builds the hat function, using the Lipschitz constant supplied in `Lip`. Parameters `num` and `numfine` determine the rough and fine partitions. `num` is 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` (\( > 1 \)) is the number of subdivisions in the finer partition in each variable. Each set \( D_k \) is subdivided into \( (\text{numfine} - 1)^{\text{dim}} \) smaller hyperrectangles, in order to improve the quality of the overestimate \( h_k \). There will be in total \( (\text{num} * \text{numfine})^{\text{dim}} \) evaluations of \( \rho \) (calls to `Distribution()`). `numfine` should be 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). `numfine` can be 2, in which case the fine partition is not used.

**PrepareHatFunctionAuto(int num, int numfine, double minLip=0)**

Builds the hat function and automatically computes an estimate to the Lipschitz constant. Parameters `num` and `numfine` determine the rough and fine partitions, and are described in `PrepareHatFunction()` method. `minLip` denotes the lower bound on the value of the computed Lipschitz constant, the default value is 0.

**RandomVec(double* p)**
Generates a random variate with the density $\rho$. **Should be called after**
PrepareHatFunction() **or** PrepareHatFunctionAuto(). The parameter $p$ is
an array of size $dim$, it will contain the components of the computed random
vector.

**RandomVecUniform(double* p)**

Generates a random vector uniform on $(0, 1)^{dim}$.

double UniformRNumber()

Returns a uniform random number on $(0, 1)$.

int GetSeed()

Returns the seed value used by the default generator `ranlux`

**SavePartition(char * fname)**

Saves the computed hat function into a file `fname`.

**LoadPartition(char * fname)**

Loads previously computed hat function from the file `fname`.

**FreeMemory()**

Frees the memory occupied by the data structures, which can be very large. **It
destroy the hat function, and RandomVec() method cannot be called after**
FreeMemory(). Automatically called from the destructor. This method
is useful to deallocate memory while the object `CRanLip` still exists.

double Lipschitz

A public member variable that contains the Lipschitz constant computed in
PrepareHatFunctionAuto.

**int Dimension**

A public member variable that stores the dimension supplied in Init(). It is
useful in the user’s implementation of Distribution() method.

**int count_total**

A public member variable that counts 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 $(\text{total accepted}/\text{count total})$. 
It is reset to 0 in `PrepareHatFunction`, `PrepareHatFunctionAuto`, `Seed` and `LoadPartition` methods.

```c++
int count_errors
```

A public member variable that counts 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 \( \text{Lip} \). It is reset to 0 in `PrepareHatFunction`, `PrepareHatFunctionAuto`, `Seed` and `LoadPartition` methods.

### 3.2 Typical usage

There are several examples of the usage of `ranlip` provided in the distribution. There are three basic steps: to supply the required distribution, to build the hat function, and to generate random variates.

```c++
#include "ranlip.h"
#define dim 4 // the dimension

// declare a derived class MyRnumGen, with one method to override
class MyRnumGen:public CRanLip {
  public: virtual double Distribution(double* p) ;
};

double MyRnumGen::Distribution(double* p) {
  // example: multivariate normal distribution
  double r=0.0;
  for(int j=0;j<Dimension;j++) r+=p[j]*p[j];
  return exp(-r);
}

void main(int argc, char *argv[]) {
  double LipConst = 4.0;
  MyRnumGen MyGen;
  double left[dim], right[dim], p[dim];
  int i;
  // set the domain to be the unit hypercube
  for(i=0;i<dim;i++) {left[i]=0; right[i]=1;}

  MyGen.Init(dim,left,right);
  MyGen.PrepareHatFunction(10,8,LipConst);
  MyGen.Seed(10);
  for(i=0;i<1000;i++) {
```

13
MyGen.RandomVec(p);
// do something with p
}
MyGen.FreeMemory();
// now MyGen can be reused
MyGen.Init(dim,left,right);
MyGen.PrepareHatFunctionAuto(10,32);
for(i=0;i<1000;i++) {
    MyGen.RandomVec(p);
}

// Example 2 using procedural interface
#include "ranlipproc.h"
#define Dim 3     // the dimension

// implement calculation of the density in MyDist function
double MyDist(double* p, int dim) {
    // example: multivariate normal distribution
    double r=0.0;
    for(int j=0;j<dim;j++) r+=p[j]*p[j];
    return exp(-r);
}

double MyRand() // use my own random number generator
{ return (double)rand()/(RAND_MAX+0.001); }
//random numbers in [0,1)

void main(int argc, char *argv[]){
    double LipConst; int i;
    double left[Dim], right[Dim], p[Dim];
    // set the domain to be a hypercube
    for(i=0;i<Dim;i++) {left[i]=-2; right[i]=2;}

    InitRanLip(Dim,a,b);
    // pass the address of the density function (required)
    SetDistFunctionRanLip(&MyDist);
    // pass the address of my generator (optional)
    SetUniformGeneratorRanLip(&MyRand);
    srand(10);     // use srand, not SeedRanLip

    PrepareHatFunctionAutoRanLip(10,8);
    for(i=0;i<1000;i++) {
        RandomVecRanLip(p);
        // do something with p
    }
    cout<<"acceptance ratio is "<<1000.0/ Count_totalRanLip() <<endl;
    LipConst=LipschitzRanLip(); // computed Lipschitz constant
if(Count_errorRanLip() > 0) { // Lipschitz constant was too low
    LipConst *= 2;
    PrepareHatFunctionRanLip(10, 32, LipConst);
}
for (i = 0; i < 1000; i++)
    RandomVecRanLip(p);

4 Discussion and examples

Calculation of the hat function in the multivariate case is a very challenging
problem. While the formulae and the algorithms presented in this paper are
formulated for any number of variables $n$, the sheer number of elements in
the partition of the domain makes the problem computationally intractable.
The number of hyperrectangles in the partition grows as an exponent of $n$.
For more than five variables the algorithm becomes impractical. This is of
course also true for any other "universal" method based on subdivision of the
domain.

On the other hand, there are still many practical problems that require sam-
pling from unusual distributions in few variables, and ranlip addresses these
needs. This method can also be used as a building block in algorithms oriented
to a larger number of variables, for instance if the unusual distribution is with
respect to a few variables only.

We would like to emphasize that practically in all cases specialized random
variate generators are much more efficient than the universal generators, like
the one implemented in ranlip. The purpose of ranlip is not to compete
with more efficient standard methods, but complement them in the case of
unusual distributions. The examples below illustrate such cases.

The first example involves polynomial-normal densities, which were recently
discussed and used in [4]. The density is given by

$$
\rho(x) = kp\left(\frac{x - \tau_1}{\tau_2}\right)N\left(\frac{x - \tau_3}{\tau_4}\right),
$$

where $p$ is a non-negative polynomial, $N$ is the density of the standard nor-
mal distribution $N(0, 1)$, and $k$ is the normalization constant. This is a very
rich family of distributions. A simple extension to the multivariate case is by
replacing $x$ with $|r|$. One example is illustrated on Fig.1 in the bi-variate case.

The second example involves a density from the following class

$$
\rho(x) = k \exp(p(x)),
$$

15
where as earlier \( p \) is a polynomial. In this particular example, see Fig.2, we used the density
\[
\rho(x, y) = k \exp(-(y - x^2)^2 - \frac{x^2 + y^2}{2}).
\]

Note that even though this distribution is unimodal, it is not log-concave, and hence methods from [7] cannot be applied.

The third example on Fig.3 involves a product of a double exponential and a normal distributions
\[
\rho(r) = k \exp(-\frac{(r + a)^2}{b}) \times \exp(-\frac{|r|}{c}).
\]

Note that in all three examples, the densities are Lipschitz continuous, and the Lipschitz constant can be calculated analytically. The preprocessing and generation time are reported in Table 1.
<table>
<thead>
<tr>
<th>Example</th>
<th>num</th>
<th>numfine</th>
<th>preprocessing time (s)</th>
<th>generation time (s × 10⁻⁶)</th>
<th>acceptance ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example 1</td>
<td>20</td>
<td>8</td>
<td>0.04</td>
<td>5.5</td>
<td>0.62</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>16</td>
<td>0.12</td>
<td>5.3</td>
<td>0.63</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>32</td>
<td>0.44</td>
<td>5.2</td>
<td>0.65</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>64</td>
<td>1.75</td>
<td>4.9</td>
<td>0.68</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>2</td>
<td>0.05</td>
<td>4.6</td>
<td>0.70</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>4</td>
<td>0.12</td>
<td>4.2</td>
<td>0.78</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>8</td>
<td>0.45</td>
<td>4.0</td>
<td>0.84</td>
</tr>
<tr>
<td>Example 2</td>
<td>20</td>
<td>8</td>
<td>0.04</td>
<td>4.8</td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>16</td>
<td>0.11</td>
<td>4.7</td>
<td>0.74</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>32</td>
<td>0.44</td>
<td>4.4</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>64</td>
<td>1.8</td>
<td>4.5</td>
<td>0.76</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>2</td>
<td>0.06</td>
<td>4.0</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>4</td>
<td>0.12</td>
<td>3.7</td>
<td>0.90</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>8</td>
<td>0.46</td>
<td>3.3</td>
<td>0.91</td>
</tr>
<tr>
<td>Example 3</td>
<td>20</td>
<td>8</td>
<td>0.04</td>
<td>4.9</td>
<td>0.65</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>16</td>
<td>0.15</td>
<td>4.7</td>
<td>0.66</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>32</td>
<td>0.48</td>
<td>4.2</td>
<td>0.68</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>64</td>
<td>1.91</td>
<td>4.1</td>
<td>0.70</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>2</td>
<td>0.05</td>
<td>4.5</td>
<td>0.68</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>4</td>
<td>0.16</td>
<td>4.1</td>
<td>0.72</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>8</td>
<td>0.51</td>
<td>3.9</td>
<td>0.80</td>
</tr>
</tbody>
</table>

Table 1. Performance of the algorithms from ranlip on the examples illustrated on Figs.1-3. The Lipschitz constant was automatically computed by the algorithm for each element of the partition $D_k$. 
5 Computational complexity and performance of the algorithms

It is important to estimate the computing time and memory requirements when using ranlip, especially for the case of several variables. The quality of the hat function directly depends on the number of values of $\rho(x)$ used for its computation. The higher this number is, the longer is preprocessing step (building the hat function), but the more efficient is the generation step (less rejected variates). A good estimate of the Lipschitz constant is also important, as it improves the quality of the hat function.

The method PrepareHatFunction(num, numfine, LipConst) uses the first two parameters to establish the rough partition of the domain, sets $D_k$, on which the hat function will have a constant value $h_k$, and the fine partition, used to compute this value. In total, $(num * numfine)^{dim}$ evaluations of $\rho$ will be performed. The memory required by the algorithm is $numfine^{dim} + 2 * num^{dim}$ values of type double (8 bytes each).

For numerical efficiency, the second parameter numfine should be a power of 2. This facilitates computation of the neighbors in an $n$-dimensional mesh by using binary arithmetic.

The tables 2 and 3 illustrate the performance of the ranlip library for more than two variables on a test distribution $\rho(r) = \exp(-(r + a)^2) \times \exp(-|r|)$, with $a = (0.1, 0.1, \ldots, 0.1)$. This distribution was chosen solely for the sake of an example, which can be easily extended to any number of variables. At the same time, this is a nontrivial distribution (see Fig.3), to which standard algorithms are not applicable. Measurements were performed on a modest workstation with Pentium IV processor (1.2GHz) and 512 MB of RAM. The tables indicate the range of applicability of the algorithm.

The data in Table 2 correspond to the method PrepareHatFunction() with an overestimate of the Lipschitz constant of $\rho$. The data in Table 3 correspond to the method PrepareHatFunctionAuto(), in which the Lipschitz constant was automatically computed.

Estimation of the Lipschitz constant automatically creates a potential problem of underestimating it. If the partition is sufficiently fine, there is no much chance of this happening (the density must have sharp peaks right in the middle of rectangles $D_k$). However for a coarse partition, which would be the case for more than three variables, such possibility exists. The way ranlip handles this issue is to give the user an option not to use Lipschitz constant smaller than some value. On the other hand, ranlip will detect violation of
the condition $h \geq \rho$ at the generation stage, giving the user an option to repeat simulation with the corrected value of the Lipschitz constant.

<table>
<thead>
<tr>
<th>n</th>
<th>num</th>
<th>numfine</th>
<th>preprocessing time (s)</th>
<th>generation time (s x $10^{-6}$)</th>
<th>acceptance ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>8</td>
<td>0.01</td>
<td>16.0</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>16</td>
<td>0.02</td>
<td>10.5</td>
<td>0.31</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>32</td>
<td>0.1</td>
<td>7.9</td>
<td>0.41</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>64</td>
<td>0.39</td>
<td>6.6</td>
<td>0.48</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>8</td>
<td>0.03</td>
<td>9.7</td>
<td>0.32</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>16</td>
<td>0.1</td>
<td>6.8</td>
<td>0.47</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>32</td>
<td>0.39</td>
<td>5.1</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>64</td>
<td>1.46</td>
<td>5.0</td>
<td>0.66</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>2</td>
<td>0.03</td>
<td>12.9</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>4</td>
<td>0.1</td>
<td>6.2</td>
<td>0.49</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>8</td>
<td>0.4</td>
<td>5.0</td>
<td>0.65</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>4</td>
<td>0.07</td>
<td>138.2</td>
<td>0.021</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>8</td>
<td>0.5</td>
<td>61.1</td>
<td>0.055</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>16</td>
<td>3.9</td>
<td>33.0</td>
<td>0.101</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>32</td>
<td>32</td>
<td>21.0</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>64</td>
<td>255</td>
<td>14.2</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>4</td>
<td>0.51</td>
<td>68.1</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>8</td>
<td>4.0</td>
<td>31.5</td>
<td>0.11</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>16</td>
<td>32</td>
<td>18.2</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>32</td>
<td>250</td>
<td>11.0</td>
<td>0.30</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>4</td>
<td>2.3</td>
<td>650</td>
<td>0.006</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>8</td>
<td>41.2</td>
<td>290</td>
<td>0.011</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>16</td>
<td>660</td>
<td>150</td>
<td>0.029</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>32</td>
<td>10911</td>
<td>75</td>
<td>0.05</td>
</tr>
</tbody>
</table>

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 $0.271 \times 10^{-6}$ sec.
<table>
<thead>
<tr>
<th>n</th>
<th>num</th>
<th>numfine</th>
<th>preprocessing time (s)</th>
<th>generation time (s × 10^{-6})</th>
<th>acceptance ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>8</td>
<td>0.01</td>
<td>5.0</td>
<td>0.51</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>16</td>
<td>0.02</td>
<td>5.7</td>
<td>0.56</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>32</td>
<td>0.12</td>
<td>5.6</td>
<td>0.57</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>64</td>
<td>0.44</td>
<td>5.6</td>
<td>0.58</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>8</td>
<td>0.04</td>
<td>4.8</td>
<td>0.71</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>16</td>
<td>0.12</td>
<td>4.7</td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>32</td>
<td>0.43</td>
<td>4.4</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>64</td>
<td>1.72</td>
<td>4.4</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>2</td>
<td>0.05</td>
<td>4.0</td>
<td>0.82</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>4</td>
<td>0.14</td>
<td>3.5</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>8</td>
<td>0.45</td>
<td>3.2</td>
<td>0.92</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>4</td>
<td>0.1</td>
<td>11</td>
<td>0.37</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>8</td>
<td>0.67</td>
<td>10</td>
<td>0.40</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>16</td>
<td>5.28</td>
<td>10</td>
<td>0.42</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>32</td>
<td>42.0</td>
<td>10</td>
<td>0.41</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>64</td>
<td>312</td>
<td>9</td>
<td>0.43</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>4</td>
<td>0.69</td>
<td>6.0</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>8</td>
<td>5.2</td>
<td>6.0</td>
<td>0.60</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>16</td>
<td>43</td>
<td>6.0</td>
<td>0.60</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>32</td>
<td>305</td>
<td>5.9</td>
<td>0.62</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>4</td>
<td>3.7</td>
<td>17</td>
<td>0.26</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>8</td>
<td>58</td>
<td>16</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>16</td>
<td>925</td>
<td>14</td>
<td>0.30</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>32</td>
<td>14301</td>
<td>14</td>
<td>0.31</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>2</td>
<td>2.45</td>
<td>103</td>
<td>0.041</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>4</td>
<td>153</td>
<td>32</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>8</td>
<td>4811</td>
<td>26</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>2</td>
<td>84</td>
<td>27</td>
<td>0.20</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>4</td>
<td>5501</td>
<td>18</td>
<td>0.3</td>
</tr>
</tbody>
</table>

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 $D_k$.  

---

---
6 Conclusion

The library ranlip was developed to complement the arsenal of existing methods of random variate generation, in cases of unusual distributions in 1-5 variables. While specialized generators are more efficient computationally, they are also costly to develop. Ranlip allows the user to sample from a non-standard distribution by specifying a formula for the probability density. It will be valuable for researchers who frequently face the task of sampling from unusual distributions, for which specialized random variate generators are not available.
7 Appendix

7.1 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. (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 $l_\infty$-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.(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 $D$, $D_k, k = 1, \ldots, K$, 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 $X$. 

22
Step 1 Generate a discrete random variate \( k \in \{1, \ldots, K\} \) 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 independent uniform random number \( Z \in (0,1) \).

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

7.2 Implementation notes

The code in ranlip includes a uniform random number generator ranlux by M.Luescher [10], 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. See discrete.c file for their description and the terms of the license (GNU General Public License).

A procedural interface to ranlip is provided in the file ranlipproc.h. This file declares several functions, that simply pass their arguments to the respective members of one instance of the class CRanLip. These functions are useful for calls to ranlip library from procedural languages, such as Fortran, or other packages, like R, Matlab. An example of usage of ranlipproc.h is provided.
References


Fig. 1. The graph of a polynomial-normal density. The density is given by
\[ \rho(r) = (|r| - 1)^2 \exp\left(-\frac{|r+0.2|^2}{3}\right), \]
where \( r = (x, y) \).
Fig. 2. The graph of the density \( \rho(x, y) = k \exp(-(y - x^2)^2 - \frac{x^2 + y^2}{2}) \).
Fig. 3. The graph of the density of a product of double exponential and normal distributions \( \rho(r) = k \exp(- (r + a)^2 / b) \times \exp(-|r|) \), \( a = (0.2, 0.1) \), \( b = 1.1 \).