Class library `ranlip` for multivariate nonuniform random variate generation

Gleb Beliakov
School of Information Technology, Deakin University
221 Burwood Hwy, Burwood, 3125, Australia
`gleb@deakin.edu.au`

May 10, 2004

Abstract

This report 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 algoritm 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 yeilds 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.

Key words: Nonuniform random variate generation, acceptance/rejection, Lipschitz densities.

Contents

1 Introduction

This report 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 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 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 2 describes theoretical background of this method of construction of the hat function. Section 3 describes the application programming interface (API) of `ranlip` library and provides a number of examples of its usage. Section 4 analyses the computational complexity of the algorithm, and 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. While a large number of efficient algorithms exists for specific distributions [1, 2], frequently the distribution is unknown at the design stage. Universal (or black box) methods have recently gained popularity [3, 6, 4], 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 [2, 4]. 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 [4, 3, 6, 5], but at the same time limit them to unimodal distributions.

In this report 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 $||\,.\,||$ 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

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

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 $Zh_k(X) \leq \rho(X)$ 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 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 $x^m, m = 1, \ldots, 2^n$. 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

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

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 ρ.

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 optimisation problem

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

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

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

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

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

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

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

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

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) 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.

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

```
// 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: multivriate 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 1 and 2 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 3 below illustrates the usage of the procedural interface.

The essential members of the class CRanLip are the following

```
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 ρ: $left_i \leq x_i \leq right_i$. **`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 $(numfine - 1)^{dim}$ smaller hyperrectangles, in order to improve the quality of the overestimate h_k. There will be in total $(num * numfine)^{dim}$ evaluations of ρ (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 ρ. **Should be called after `PrepareHatFunction()` or `PrepareHatFunctionAuto()`.** The parameter `p` is an array of size dim, it will contain the components of the computed ran-

dom 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 destroys 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 ($total\_accepted/count\_total$). It is reset to 0 in `PrepareHatFunction`, `PrepareHatFunctionAuto`, `Seed` and `LoadPartition` methods.

`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 `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.

```
#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: multivriate 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++) {
      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
#include "ranlip.h"
#define dim 3             // 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: multivriate 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;
   MyRnumGen MyGen;
   double left[dim], right[dim], p[dim];
   int i;
// set the domain to be a hypercube
   for(i=0;i<dim;i++) {left[i]=-2; right[i]=2;}
   MyGen.Init(dim,left,right);
   MyGen.PrepareHatFunctionAuto(10,32,0.01);
   MyGen.Seed(10);

   for(i=0;i<1000;i++) {
      MyGen.RandomVec(p);
   // do something with p
   }
   cout<<"acceptance ratio is "<<1000.0/ MyGen.count_total<<endl;
   LipConst=MyGen.Lipschitz; // computed Lipschitz constant
   if(MyGen.count_error>0) { // Lipschitz constant was too low
      LipConst*=2;
      MyGen.PrepareHatFunction(10,32,LipConst);
   }
   for(i=0;i<1000;i++) {
      MyGen.RandomVec(p);
   // do something with p
   }
}
```

```
// Example 3  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 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 ρ will be performed. The memory required by the algorihm 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 neighbours in an n-dimensional mesh by using binary arithmetic.

The tables below illustrates the performance of the `ranlip` library on a test distribution (a mixture of 5 normal distributions). 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 1 correspond to the method `PrepareHatFunction()` with an overestimate of the Lipschitz constant of ρ. The data in Table 2 correspond to the method `PrepareHatFunctionAuto()`, in which the Lipschitz constant was automatically computed.

We would like to point out that an efficient parallelisation of the presented methods for computer clusters is available. This way, combined RAM and power of several workstations can be used, and random variates of higher dimension can be generated. Contact the author for an implementation of `ranlip` which uses MPI standard.

n	num	numfine	preprocessing time (s)	generation time (s $\times 10^{-6}$)	acceptance ratio
2	10	8	0.01	16.0	0.19
	10	16	0.02	10.2	0.30
	10	32	0.1	7.8	0.40
	10	64	0.38	6.6	0.48
	20	8	0.03	9.6	0.33
	20	16	0.1	6.8	0.47
	20	32	0.39	5.2	0.58
	20	64	1.46	5.0	0.65
	80	2	0.03	12.6	0.25
	80	4	0.1	6.4	0.49
	80	8	0.39	4.8	0.67
3	10	4	0.07	134.2	0.026
	10	8	0.49	61.1	0.056
	10	16	3.9	34.0	0.103
	10	32	31.8	21.0	0.17
	10	64	252	14.1	0.24
	20	4	0.51	68.1	0.051
	20	8	4.0	31.0	0.108
	20	16	31.3	18.1	0.19
	20	32	250	11.0	0.30
4	10	4	2.73	651	0.006
	10	8	43.2	283	0.013
	10	16	667	139	0.028
	10	32	10634	74.1	0.05

Table 1. 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 0.271×10^{-6} sec.

n	num	numfine	preprocessing time (s)	generation time (s $\times 10^{-6}$)	acceptance ratio
2	10	8	0.01	5.0	0.55
	10	16	0.02	5.8	0.57
	10	32	0.11	5.6	0.58
	10	64	0.45	5.6	0.58
	20	8	0.04	4.8	0.72
	20	16	0.12	4.6	0.73
	20	32	0.43	4.4	0.74
	20	64	1.74	4.4	0.74
	80	2	0.05	4.0	0.84
	80	4	0.14	3.6	0.91
	80	8	0.45	3.2	0.92
3	10	4	0.1	11	0.37
	10	8	0.67	10	0.39
	10	16	5.28	10	0.41
	10	32	42.0	10	0.41
	10	64	311	9	0.43
	20	4	0.68	6.1	0.58
	20	8	5.2	6.0	0.60
	20	16	41.5	6.0	0.61
	20	32	301	5.9	0.62
4	10	4	3.68	17	0.27
	10	8	57.7	16	0.29
	10	16	923	15	0.30
	10	32	14311	14	0.32
5	10	2	2.46	111	0.046
	10	4	156	30	0.17
	10	8	4863	26	0.19
	20	2	79	27	0.20
	20	4	5434	18	0.29

Table 2. 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.

References

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

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

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

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

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

[6] 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.

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

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

5 Appendix

5.1 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 $D = \{x \in R^n : a_i \le x_i \le b_i\}$, 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.(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.

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

5.2 Implementation notes

The code in `ranlip` includes a uniform random number generator `ranlux` by M.Luescher [7], and the discrete random variate generator based on the *alias* method by A. Walker [8]. 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.