ORATIO Library
Reference manual

Fabian Bastin

November 28, 2011

Contents

Chapter 1

Introduction

Heere sweet Lord, at your service.
Horatio, The Tragedie of Hamlet.

This document covers the version 0.4.4 of the library ORATIO, and is itself on a beta development level. Any suggestion should be sent to `bastin@iro.umontreal.ca`.

The library design is also subject to modifications, so please refer to this documentation if you face linkage problems after the upgrade of the library. You can also consult the file `ChangeLog` at the root of the package, as well as in each subdirectories to read more specific changes.

1.1 A brief overview

ORATIO is a collection of sublibraries, each of then aiming to cover some specific needs. These components are

C Tools : collection of basic C utilities;

Data strucutres : basic data structures as lists and trees;

Numerical utilities : various basic numerical tools, as finite and central difference, special functions,...;

Statistical tools : basic statistical analysis and distributions properties routines;

Random numbers : (pseudo-)random numbers generation;

Quasi-Monte Carlo : quasi-Monte Carlo sequences generation.

Each of these parts is described in more details in the next chapters.

1.2 Requirements and installation

The library makes heavy use of `OpenBlas` (Basic Linear Algebra Subroutines) functions, instead of rewriting the standard operations. This allows the code to use architecture-optimized `OpenBlas` and to rely of well-established standards. Any library implementing these routines will be compatible, but possibly could require the modification of the file `configure.in` to be detected. We currently advocate the `OpenBlas` library, freely available at the address `https://github.com/xianyi/OpenBLAS`.

Once these dependencies are satisfied, the library can be installed with the following instructions (as administrator):

```
./configure
make
make install
```

This documentation can also be generated using the command

```
make pdf
```

which however requires a proper installation of LaTeX.

1.3 Conventions

$x(i)$ designs the i-th component of the vector x. When comparing two vectors x and y in $\mathbb{R}^n$, an inequality $x \leq y$ is meant componentwise, that is $x(i) \leq y(i),\ i = 1, \ldots, n$.

1.4 Credits

The development of the library has been initiated at FUNDP - the University of Namur (Belgium), continued at the CERFACS (Toulouse, France), and is currently pursued at the Université de Montréal (QC, Canada).

Various people have contributed to this library. Their names is written at the beginning of concerned files, and the complete list is reproduced here, by alphabetical order: Adrien Bonneu, Cinzia Cirillo, Pierre Derian, Jordan Goblet, Gabriel Schwanen.

1.5 Miscellaneous

ORATIO stands for *Operational Research, A Toolbox in Operation*. Most of the routines were partially or totally written during the development of the software AMLET (*Another Mixed Logit Estimation Tool*). The name is a reference to the Shapekespeare's masterpiece Hamlet. Horatio was the best friend of Hamlet, as the library ORATIO is the primary companion of the software AMLET.

The painting of the front cover is due to Eugène Delacroix, in 1835, under the name *Hamlet and Horatio in the Graveyard*.

Chapter 2

C Tools

While standard C libraries are quite comprehensive, it remains useful to define some additional utilities in order to alleviate the programming work, and sometimes improve the code portability. The following set of routines and constants aims to achieve this goal.

2.1 Constants

Various constants are proposed, in order to improve code readibility. First, two constants are defined to express boolean conditions:

- TRUE, defined as 1;
- FALSE, defined as 0.

Note that the constants names are written in uppercase. The number of bits associated to the various basic C types can also be designed with the following constants:

```
CHARBITS
SHORTBITS
INTBITS
LONGBITS
PTRBITS
DOUBLEBITS
FLOATBITS
```

The number of bits used for any other type can be computed with the macro

```
CTOOLS_BITS(type)
```

```
typedef void (*C_GENERIC)(void);
```

2.2 File gestion

While C standard libraries have functions to read text and binary files, they not always easy to use. We encapsulate some functions to simplify some basic operations on text files.

2.2.1 Reading

It it possible to read all the content of a text file in one shot, with the function

```
char *c_file_get_text(char *name, long *length)
```

The content is copy, verbatim, into a newly memory allocated characters array, and the special character EOF (for end-of-file) is added. The variable pointed by length, if length is not NULL, contains the size of this character array (include the EOF character).

The current line of the file can be obtained with the function

```
char *c_file_get_line(FILE *input);
```

The returned string is newly allocated, so it has to be freed by the user.

2.2.2 CSV format

A specific file format is the comma-separated values format, which is useful to represent simple data.

If separator is set to `\0`, the constant `C_CSV_DEFAULT_SEPARATOR` is used.

```
C_CSV *c_csv_new(char *separator)

void c_csv_free(C_CSV *csv)
```

2.3 Memory management

```
#define _malloc(type) ((type *)c_malloc(sizeof(type)))
#define _mallocs(nb, type) ((type *)c_malloc((nb)*sizeof(type)))
#define _realloc(ptr, type) ((type *)c_realloc(ptr, sizeof(type)))
#define _reallocs(ptr, nb, type) ((type *)c_realloc(ptr, (nb)*sizeof(type)))
#define _free(var) (if (var) free(var));
```

Chapter 3

DSTRUCT

This library has as objective to allow easy manipulation of standard data structures, from basic operations to less standard, while useful ones. The library is not architecture optimized, so it is possible that you can find architecture-dependent libraries that are more efficient for your own needs, in which case this I would not suggest you to use this library. This library is meant to be general-purpose, while complying with the C language, in order to improve its portability.

3.1 Conventions

Each function of the library has a definition starting with `ds_`, followed by its main purpose (e.g. `list`, `sort`,...), '`_`', and is specific function. If several functions exist for a similar feature, but are specialez for a specific data type, the function specification will be preceded by a letter designing the data type. The main ones are `d` for double, `i` for integer and `g` for generic (`void` pointer).

3.2 Function types

Several routines rely and standardized definition types. An important situation is the comparison of two elements with respect to some user-defined ordre. The comparison functions have to follow to following prototype:

```
typedef int (*DSCompareFunc)(const void *a, const void *b);
```

By convention, a comparison function should return 0 if `a` is considered equal to `b` (with respect to the defined order), a strictly positive number if `a` is

strictly greater than `b`, and a strictly negative number if `a` is strictly smaller than `b`.

3.3 Linked lists

A linked list is probably one of the most used data structure, so the library supports the two main variants: singly-linked lists and doubly-linked lists. The same functions are defined for these two kinds of list, and the convention is to prefix the function name by `ds_list` for a doubly-linked list, and `ds_slist` for a singly-linked list. Both lists take `void *`, allowing the use of any kind of structure.

3.3.1 Singly linked lists

A singly linked list has the type `DS_SList`.

3.3.2 Doubly linked lists

A singly linked list has the type `DS_List`.

3.4 Searching functions

```
int ds_search_double(const void *a, const void *b)

int ds_search_int(const void *a, const void *b)

int ds_search_pint(const void *a, const void *b)

long ds_find_dicho(void *base, int nmemb, size_t size, void *critere,
   DSCompareFunc search)

long ds_find_inter(void *base, int nmemb, size_t size, void *critere,
   DSCompareFunc search)
```

3.5 Sorting and permutation functions

3.5.1 Permutations

A permutation p is defined as vector of size n whose elements are integers in $0, \ldots, n-1]$, all being distinct, such that is a is some array, the application

of the permutation produce an array v defined as

$$v[i] = a[p[i]].$$

The function `ds_perm` and `ds_perm_inp` apply some pertumation to an general array, the first producing a new array, the last replacing the original array by the permuted one.

3.5.2 Heapsort

We have not implemented the Quicksort algorithm since it is already available in the standard C library, and in most cases, it is sufficient (and efficient), but still has a worst case in $O(n^2)$. We present an implementation of the Heapsort algorithm, both for standard types and for generic data. While the implementation is sufficient for most needs, C++ programmers are also adviced to turn for the Standard Templates Library (at least for good implemented ones), since the proposed functions are often reported to be faster than the C implementations, including those present in Standard C Libraries. More efficient, but more difficult to implement, is the Introsort method, due to Musser [13], that combines the usual nice behaviour of Quicksort, while avoiding the worst case, since its complexity follows the $O(n \log n)$ of Heapsort. The `dstruct` does not currenctly propose Introsort.

The basic version of the proposed Heapsort algorithm are the following ones, dealing with double and integer vectors. In order to indicate which data type is concerned, the name `heapsort` is simply prefixed by '`d`' (for double) and '`i`' (for integer).

```
void ds_sort_dheapsort(int n, double *array);

void ds_sort_iheapsort(int n, int *array);
```

Each one takes as arguments the number of elements in the array to sort, and a pointer to the array itself.

The generic version of Heapsort in `dstruct` has a similar declaration than the quicksort standard library, so you can easily change the called function:

```
void ds_sort_gheapsort(void *base, size_t nmemb, size_t size,
       int(*compar)(const void *, const void *));
```

Sorting is nothing else that generating a particular permutation of the original vector, and it sometimes useful to knows this permutation. Each Heapsort algorithm has an equivalent version generating this permutation, defined as in Section 3.5.1.

```
void ds_sort_dheapsort_with_perm(int n, double *array, int *perm);

void ds_sort_iheapsort_with_perm(int n, double *array, int *perm);
```

Chapter 4

Graph algorithms

4.1 Definitions

Let V be a finite set, and denote by

$$E(V) = \{\{u, v\} \,|\, u, v \in V, u \neq v\},$$

the subsets of V of two distinct elements.

Definition 4.1: Graph
A pair $G = (V, E)$ with $E \subseteq E(V)$ is called a graph (on V). The elements of V are the vertices, and those of E the edges of the graph. The vertex set of a graph G is denoted by V_G and its edge set by E_G. Therefore $G = (V_G, E_G)$.

In literature, graphs are also called simple graphs; vertices are called nodes or points; edges are called lines or links. A pair $\{u, v\}$ is usually written simply as uv. Notice that then $uv = vu$. In order to simplify notations, we also write $v \in G$ instead of $v \in V_G$.

4.2 Computer representation

As any graph is basically constituted by nodes and edges, we define a type for the nodes, a type for the edges. The graph itself will be represented by an object of type `GR_Graph`. Various computer representations are possible, but in many applications, graph are sparse, as many possible edges, corresponding to an ordered pair of nodes (in the case of an oriented graph) do not exist. To account of this situation, we decided to represent a graph

simply as a collection of nodes, and each node is associated to a set of input edges and a set of output edges. This representation is well suited for oriented graphs, while for non-oriented graphs, we then need to duplicate each edge. A graph can then be allocated using the function

```
GR_Graph *gr_graph_new(int n)
```

where n is the number of nodes. The user has then to add the nodes and the edges. In order to help node addition, the function `gr_graph_new` also initialize the set of nodes and give (unique) identifiers from 0 to $n-1$ (included) to these nodes. The memory can be freed using the function

```
void gr_graph_free(GR_graph *graph)
```

This function with also free the memory associated to each node of the graph.

The basic type for a node is `GR_Node`, which has the following attributes:

1. `id`: a numeric identifier;
2. `label`: a string label;
3. `value`: an array of real values attached ti this node;
4. `in`: the list of input edges;
5. `out`: the list of output edges.

A node can be created using the function

```
GR_Node *gr_node_new()
```

and freed with

```
void gr_node_free(GR_Node *node)
```

4.3 Graphs algorithms

4.3.1 Shortest paths

Selecting an origin s and a destination t in a graph, the problem is to find a shortest path between these two nodes. Various algorithms exist, and the library provides some simple implementations, aiming to illustrate the techniques.

Topologically-ordered algorithms

The first presented algorithms are inspired from dynamic programming techniques. We assume that the graph is topologically-ordered, i.e. it is ordered and an edge (i, j) can exist only if $i < j$, and that $a_{ij} > 0$ if the edge (i, j) exists. Three algorithms are proposed. The first is a direct translation of the dynamic programming algorithm. If $J(i)$ is the minimum distance from i to t, then

$$J(j) + a_{ij} = J(i),$$

if the edge (i, j) belongs to a shortest path. The function

```
void gr_graph_backward_linking(int n, GR_Node *nodes,
                               int origin, int destination,
                               double *J, int *next);
```

implements the method, fixing the distances from the destination to the origin. On output, `J[i]` contains the distance from node i to the node `destination`, and `next[i]` contains the successor of node i on the shortest path from `origin` to `destination`.

```
void gr_graph_forward_linking(int n, GR_Node *nodes,
                              int origin, int destination,
                              double *D, int *previous);

void gr_graph_accession(int n, GR_Node *nodes,
                        int origin, int destination,
                        double *D, int *previous);
```

Label correcting algorithm

```
void gr_graph_dijkstra(int n, GR_Node *nodes,
                       int origin, int destination,
                       double *dist, int *previous);

void gr_graph_dijkstra_edges(int n, GR_Node *nodes,
                             int origin, int destination,
                             double *dist, GR_Edge **previous,
                             double (*weight)(GR_Edge *edge));
```

The more general algorithm allows to consider edges of negative weight, as long as there is no negative cycle.

```
int gr_graph_labelcorrecting_adj(int n, int s, int t,
                                 int **adj, double **v_adj,
                                 double *u,
                                 int *parent, double *w);
```

Chapter 5

Numerical utilities

ORATIO library includes various numerical code snippets, gathered into the sublibrary `ntools`. The routines concerns various aspects of numerical programming, that can be useful in various situations. By convention, the routines names always start with the prefix `ntools_`. We also try to benefit form preexisting conventions in packages as `OpenBlas` and `LAPACK` in order to make naming schemes as consistent as possible. The presented conventions will also be used in the subsequent chapters.

5.1 Constants and special values

Mathematical constants are represented in the file `constants.h`, and are summarized in Table 5.1.

```
double nt_infinity()
double nt_nan()
```

5.2 Geometric routines

5.2.1 Cartesian and polar coordinates

Given a two-dimensional vector a, the two classical coordinates system are the cartesian and polar ones. In the Cartesian system, a is expressed as a linear combination of the canonical vectors $e_1 = (1, 0)$ and $e_2 = (0, 2)$ (defining the x- and y-axis):

$$a = xe_1 + ye_2.$$

Symbol	Value	Numerical approximation
M_E	e	2.71828182845904523536028747135
M_EULER	$\Gamma'(1)$ (Euler constant)	0.5772156649015328606065
M_LN2	$\ln(2)$	0.69314718055994530941723212146
M_LNPI	$\ln(\pi)$	1.14472988584940017414342735135
M_LN10	$\ln(10)$	2.30258509299404568401799145468
M_LOG2E	$\log_2(e)$	1.44269504088896340735992468100
M_LOG10E	$\log_{10}(e)$	0.43429448190325182765112891892
M_PI	π	3.14159265358979323846264338328
M_PI_2	$\pi/2$	1.57079632679489661923132169164
M_PI_4	$\pi/4$	0.78539816339744830961566084582
M_1_PI	$1/\pi$	0.31830988618379067153776752675
M_2_PI	$2/\pi$	0.63661977236758134307553505349
M_SQRT1_2	$\sqrt{1/2}$	0.70710678118654752440084436210
M_SQRT2	$\sqrt{2}$	1.4142135623730950488
M_SQRT3	$\sqrt{3}$	1.73205080756887729352744634151
M_SQRTPI	$\sqrt{\pi}$	1.77245385090551602729816748334
M_TWO_SQRTPI	$2/\sqrt{\pi}$	1.12837916709551257389615890312
M_TWO17	2^{17}	131072.0
M_TWO53	2^{53}	9007199254740992.0

Table 5.1: Mathematical constants

Therefore, we can write a as the couplet (x, y). In the polar system, (r, θ), where r is the radial distance from the origin, and θ is the counterclockwise angle from the x-axis. Storing the Cartesian coordinates in a vector, we have the two following functions, that perform conversion from Cartesian to polar coordinates and from polar to Cartesian coordinates, respectively:

```
void nt_cartesian2polar(double *x, double r, double theta);
void nt_polar2cartesian(double r, double theta, double *x);
```

5.3 Derivatives

5.3.1 Finite approximations

A common way to compute approximate first and second derivatives is to use finite-difference approximations.

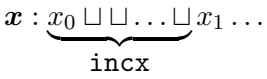


Figure 5.1: Vector representation

Finite-difference gradients

5.4 Linear algebra

5.4.1 Conventions

Each real vector $\boldsymbol{x}$ is represented as a double array whose consecutive elements are separated using a fixed increment, as illustrated in Figure 5.4.1.

When manipulating matrices, we follow the `LAPACK` convention (see for instance Hogben [5], Chapter 75), and denote the number of rows by m and the number of columns by n. We also define the leading dimension as the first dimension of a two-dimensional array (as opposed to the dimension of the matrix stored in that array). Since arrays are stored by rows in C, the leading dimension is often the number of columns, but it can greater than this number, allowing convenient abstraction to access submatrices. The basic memory allocation and deallocation are

```
double **nt_matrix_new(int m, int n)
double **nt_matrix_realloc(double **A, int m, int n)
void nt_matrix_free(double **A)
```

Various additional functions offer basic operations on matrices. The function

```
void nt_matrix_zeros(int m, int n, double *A, int ldA)
```

set all elements of the $m \times n$ leading principal submatrix of A to 0.0. `ldA` is the leading dimension of A. This submatrix can also be set to the $m \times n$ identity matrix with the function

```
void nt_matrix_identity(int m, int n, double *A, int ldA)
```

These functions implicitely assume that the matrix is row-oriented, as said before, but this is not the case with every language. In particular, Fortran stores matrices by columns. Since many numerical routines are available in Fortran, and can be easily linked to C code, it can be useful to specify how

matrices are stored. This can can be done using specialized functions, whose names start by `nt_ma` instead of `nt_matrix`, and whose first argument is the matrix orientation identifier. It has the specific type `enum CBLAS_ORDER`, and can take one of the following constant values:

1. `NT_MATRIX_ROW` or `CblasRowMajor` for a row-oriented matrix;
2. `NT_MATRIX_COLUMN` or `CblasColMajor` for a column-oriented matrix.

The constants `CblasRowMajor` and `CblasColMajor` are proposed in order to be consistent with `OpenBlas` (see Section 1.2) conventions. The corresponding functions list is

```
double **nt_ma_new(const enum CBLAS_ORDER Order, int m, int n)
void nt_ma_diag(const enum CBLAS_ORDER Order,
const int m, const int n, double *A, const int ldA,
double alpha)
void nt_ma_identity(const enum CBLAS_ORDER Order,
    const int m, const int n, double *A, const int ldA)
void nt_ma_scale(const enum CBLAS_ORDER Order, const int m, const int n,
 double alpha, double *A, int ldA)
void nt_ma_zeros(const enum CBLAS_ORDER Order,
 const int m, const int n, double *A, const int ldA)
```

5.4.2 I/O routines

These routines allow simple input/output management for some mathematical constructions.

```
void nt_vect_print(FILE *out, char *name,
                   double *v, int n)

void nt_matrix_print(FILE *out, char *name,
     double **A, int m, int n)
```

5.4.3 Modular arithmetic

Let $m > 0$ be a natural number, and consider the set $\mathbb{Z}_m \stackrel{def}{=} \{0, 1, \ldots, m-1\}$. For $a, b \in \mathbb{Z}_n$ define $a + b$ and ab by performing these operations in $\mathbb{Z}$, and the subtracting multiples of m until the result is in $\mathbb{Z}_m$. In other words, form the sum and product of a and b as integers, divide them by m and take the remainders. These operations are called the addition and multiplication

modulo m. Under these relations, we have that two integers a and b are said to be congruent modulo m if their difference is a multiple of m (in other terms the remainder of the integer division $(a-b)/m$ is equal to 0). We will write $a = b \mod m$.

Multiplication: $ax \mod m$

Mupltiplication modulo m is an important tool in various application, for instance random numbers generations. The main difficulty is that with large m, a crude implementation of $ax \mod m$ could lead to overflows. We examine two ways to circumvent this problem.

Approximate factorization This method is also know as Schrage's method, and is valid if $a^2 < m$ or $a = \lfloor m/i \rfloor$, where $i^2 < m$. We work with integer only, and precompute $q = \lfloor m/a \rfloor$ and $r = m \mod a$. We then

$$y \leftarrow \lfloor x/q \rfloor; \quad x \leftarrow a(x - yq) - yr; \quad \text{If } x < 0 \text{ then } x \leftarrow x + m.$$

We indeed have that

$$\begin{aligned} ax \mod m &= (ax - \lfloor x/q \rfloor m) \mod m \\ &= (ax - \lfloor x/q \rfloor (aq + r)) \mod m \\ &= (a(x - \lfloor x/q \rfloor q) - \lfloor x/q \rfloor r) \mod m \\ &= (a(x \mod q) - \lfloor x/q \rfloor r) \mod m. \end{aligned}$$

Under our assumptions, it is easy to show that all intermediate quantities stay between $-m$ amd m.

The following function computes $axmodm$ and returns the result of the operation

`long nt_modular_schrage(long a, long x, long m)`

While we propose a generic function, it is usally more efficient to directly implement this technique inline, by precomputing q and r, as explined.

Floating-points computation If we assume that we work with double precision numbers coded on 64 bits, with a mantissa represented by 53 bits, the presented technique is valid if $a.m < 2^{53}$, since the we can represent all integers betweem 0 and $2^{53} - 1$ with no rounding errors. The following algorithm produces the desired result.

$$\begin{aligned} &\text{double } m, a, x, v; \quad \text{int } k; \\ &v = a.x; \quad k = \lfloor v/m \rfloor; \quad x = v - k.m; \end{aligned}$$

The following function computes $ax + ymodm$:

```
double nt_modular_daxpy(double a, double x, double c, double m)
```

The name `daxpy` is used as reference to the `OpenBlas` operation $ax + y$.

Power

The multiplication operation is much simpler when x is equal or more generally to a power of a. The following routine performs the operation $a^e \mod m$.

```
long nt_modular_power(long a, long e, long m);
```

5.5 Primes numbers

More advanced and efficient algorithm exist, but their implementation goes far over the purpose of this library. We only propose simple routines, that you should be sufficient for numerous applications, when only a few prime numbers are required. If you require a general but fast prime numbers search, we recommand the use of Atkin's sieve [1].

5.5.1 Successive divisions

The simple methods consists to enumerate naturals and check if their primes by testing if they can be divided by a previously computed prime. The method is less memory consuming that the Eratosthene sieve presented below, but is requires more CPU time. You can compute the first n prime number, with the following function:

```
void nt_primes_simple(unsigned int n, unsigned int *primes)
```

The array `primes` has to be memory allocated. On output, `primes[i]` will containes the $(i-1)^{\text{th}}$ prime. As an illustration, the 10 first prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29.

5.5.2 Eratosthene sieve

```
void nt_primes_eratosthene(unsigned int n, unsigned int *primes)
```

5.6 Special functions

Various functions are directly supported in `ORATIO` as we can frequently encounter them, and either implementaion is not trivial, either we can try to optimize their efficiency.

5.7 B-Splines

Various utility functions are defined to deal with B-splines. A B-spline function of degree p is a polynomial function of degree p, defined on the interval $[a, b]$, that can be expressed as a linear combination of $n + 1$ basis functions $N_{i,p}(u)$, as follows:

$$C(u) = \sum_{i=0}^{n} P_i N_{i,p}(u). \tag{5.1}$$

The coefficients $P_0, P_1, \ldots, P_n$ are called the control points, and u is the knot vector ($u_0 = a, u_1, \ldots, u_m = b$). The basis functions can be constructed by recurrence on the degree p:

$$N_{i,0} = \begin{cases} 1 & \text{if } u \in [u_i, u_{i+1}), \\ 0 & \text{otherwise.} \end{cases}$$

and

$$N_{i,p} = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u),$$

so that n is equal to $m - p - 1$.

There are several types of knot vectors, but one especially convenient is the nonperiodic (or clamped or open) knot vector, which has the form

$$U = \{\underbrace{a, \ldots, a}_{p+1}, u_{p+1}, \ldots, u_{m-p-1}, \underbrace{b, \ldots, b}_{p+1}\},$$

that is the first and last knots have multiplicity $p + 1$. It is possible to show that the function $C(u)$ is $p - 1$ continuously differentiable.

The implementation proposed in the library is based on Piegl and Tiller [17]. A space of B-splines is represented with the type `NT_BSpline`, defining the set of a basis function and the knot vector. A new B-spline space can be created using the function

```
NT_BSpline *nt_bsplines_new(int p, int nk, double *knots, int inck)
```

where, as before, `p` is the degree of the spline, `nk` is the dimension of the knots vector pointed by `knots`, with the increment `inck`, which should be set to 1 is the elements or consecutive. The allocated memory can further be free using

```
void nt_bsplines_free(NT_BSpline *bspline)
```

This function internally creates the knots vector U (with multiplicity p+1 for the extreme points) using the following function, that returns the number of components in U, i.e. `nk` plus $2p$.

```
int nt_bsplines_knots_create(int p, int nk, double *knots,
                             int *nU, double **U)
```

The U dimension can also be recovered with the function

```
int nt_bsplines_knots_dimension(int p, int nk)
```

and a vector of appropriate size can be created with the function

```
double *nt_bsplines_knots_allocate(int p, int nk)
```

A knot vector with all multiplicities equal to 1, and n components, can be adjusted as an open knot vector with the function below.

```
int nt_bsplines_knots_initialize(int p, int *n, double **U,
                                 int realloc)
```

The variable `realloc` has to be set to 1 if a memory reallocation has to be performed, and 0 otherwise. On output, U is the open knot vector, and n is the number of its components, also returned by the function.

The dimension n in (5.1), given the degree p of the spline basis functions, and the dimension $m + 1$ of the knot vector with multiplicities set to 1, can be retrieved with the function

```
int nt_bsplines_space_dimension(int p, int m)
```

In order to define a particular B-Spline in such a space, the user has to specify the vector of control points P. The B-Spline can then be evaluated at a point u with the function

```
double nt_bsplines_evaluate(NT_BSpline *b, double *P,
                            double u)
```

This function is the short form of the routine below, where `n` is the dimension of the open knot vector `U`, `p` is the spline degree, and `N` is a vector of size at least equal to `p`+1, that will be used to recursively evaluate the basis functions:

```
double nt_bsplines_curve_point(int n, int p, double u, double *U,
                               double *P, double *N)
```

5.7.1 Integer power

The standard function `pow`, in the mathematical library, only consider x^y, where x and y are real (with the restriction that y must be an integer if x is negative. When y is some integer, it is possible to simplify the computation by only performing multiplications, whose number can be limited. The function is similar to `pow` as it takes two arguments, x and y, wehere y is restricted to be an integer, and return the value x^y:

```
double nt_ipow(double x, int y)
```

On a Macbook Pro 2.0Ghz, and compiler, gcc-4, our routine appeared significantly faster (by a factor of 2 to 3) than the standard function for simple tests.

5.7.2 Factorial

The value of $n!$, where n is a nonnegative integer, can be obtained with the function

```
double nt_factorial(int n)
```

5.7.3 Powers of 2

It is possible to precalculate an array containing powers of 2 using to function

```
void nt_two_powers_array(int n, unsigned int *p)
```

Then `p[i]` will be equal to 2^i, as long a i is not too large compared to the integer machine precision.

It is also possible to compute the power of 2 closest but inferior to some integer x, i.e i, such that $2^i <= x$, and for all $j > i$, $2^j > x$.

```
int nt_two_power_floor(int x)
```

Similarly, the function

```
int nt_two_power_ceil(int x)
```

returns i, where $2^i >= x$, and for all $j < i$, $2^j < x$.

5.7.4 Radical inverse

Let $i \in \mathbb{N}$, and $b \in \mathbb{N}_0$. The b-ary expansion of i is

$$i = a_0 + a_1 b + \ldots + a_{k-1} b^{k-1},$$

and its radical inverse in base b is

$$\psi_b(i) = a_0 b^{-1} + a_1 b^{-2} + \ldots + a_{k-1} b^{-k}.$$

The value of $\psi_b(i)$ can be computed using the function

```
double nt_radical_inverse(int b, int i);
```

Chapter 6

Simulation

The ORATIO library contains some basic modules allowing simple discrete events simulations experiments. This module has not the ambition to compete with commercial or well-established open-source packages, but to provide support tools for teaching and research.

6.1 Introduction to discrete events simulation

We consider a system characterized by the state $\mathcal{S}(t)$, where t denotes the time. In the discrete events setting, the system state can only change at a countable number of points in time. The state can be modified when (and only when) some event happens, which is assumed to be instantaneous. The system evolution is therefore the consequence of a sequence of events, $e_0, e_1, e_2, \ldots$, occuring at times $0 = t_0 \leq t_1 \leq t_2 \leq \cdots$. Let $\mathcal{S}_i$ the system state directly after e_i, and define the simulation time as the current value of t_i. We can represente the discrete events simulation as illustrated on Figure 6.1.

The core of a discrete events simulation as proposed in ORATIO is the `ESim` structure, which will be responsible of the simulation clock and events list gestion. A new instance can be created with the function

```
ESim *esim_new()
```

and suppressed with the function

```
void esim_free(Esim *sim);
```

The second basic component of a discrete events simulation is an event, that can be created with the function

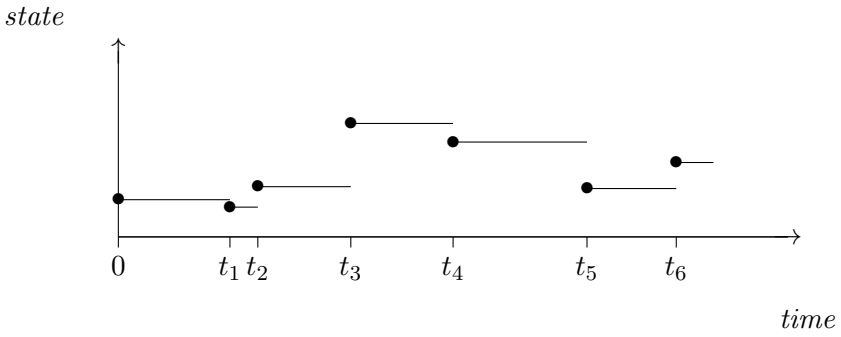


Figure 6.1:

```
ESim_Event *esim_event_new(ESim_Action action);
```

and can be freed with the function

```
void esim_event_free(void *ev);
```

The use of the `void *` type makes the function compatible with the utilities associated with the gestion of lists.

Since an event usually modifies the system state, it has to perform some action. The type `ESim_Action` is a pointer to a function of the form

```
int (* ESim_Action)(void *system);
```

The choice to use C instead of an object-oriented programming approach imply some technical limitations if we want to maintain generality. The event function prototype considers a system as a `void *` pointer, so that every implementable system can be considered. It is the responsability of the programmer to correctly cast the `void *` pointer in the events gestion functions.

Chapter 7

Statistical library

This part of the ORATIO library is devoted to statistical utilities, from basic analysis functions to various analytical distributions. The current version mainly consider univariate distributions, but support of multivariate distributions is planned.

7.1 Measure and probability theory

We start this section with the review of some theoretical background.

7.1.1 Measure space

A probability measure is a measure define on a measurable space $(\Omega, \mathcal{F})$, where Ω is any collection of objects, and $\mathcal{F}$ is a σ-algebra of subsets of Ω. Formally, a subset $\mathcal{F}$ of 2^{Ω}, the power set of a set Ω, the set of all subsets of Ω, is a σ-algebra (also called σ-field) if it has the following properties:

1. $\mathcal{F}$ is not empty;

2. $\mathcal{F}$ is closed under complements: if E is in $\mathcal{F}$ then its the complement $E^C \stackrel{def}{=} \Omega \backslash E$ is also is $\mathcal{F}$;

3. $\mathcal{F}$ is closed under countable unions.

We now equip the measure space with a measure, using the following definition.

Definition 7.1: measure
Given a class $\mathcal{F}$ of subsets of a set Ω, a measure

$$\mu : \mathcal{F} \to \overline{\mathcal{R}}$$

is a function having the following properties:

(a) $\mu(\emptyset) = 0$.

(b) for a countable collection $\{A_j \in \mathcal{F},\ j \in \mathbb{N}\}$ with $A_j \cap A_k = \emptyset$ for $j \neq k$, and $\cup_j A_j \in \mathcal{F}$,

$$\mu\left(\cup_j A_j\right) = \sum_j \mu(A_j).$$

When $(\Omega, \mathcal{F})$ is a measurable space, the triple $(\Omega, \mathcal{F}, \mu)$ is called a measure space.

A probability space is a triple $(\Omega, \mathcal{F}, P)$ consisting of a set Ω (called the sample space), a σ-algebra $\mathcal{F}$ of subsets of Ω (these subsets are called events), and a measure P on $(\Omega, \mathcal{F})$ such that $P(A) \geq 0$ for all $A \in \mathcal{F}$, and $P(\Omega) = 1$ (called the probability measure).

7.1.2 Measurable function

If Σ is a σ-algebra over a set X and $\mathcal{T}$ is a σ-algebra over Y, then a function $f : X \to Y$ is measurable $\Sigma/\mathcal{T}$ if the preimage of every set in $\mathcal{T}$ is in Σ.

7.2 Random variables

7.2.1 Basic properties

Recall that a distribution function $F(x)$ has the following properties

1. $0 \leq F(x) \leq 1$ for all x.
2. $F(x)$ is nondecreasing.
3. $\lim_{x\to\infty} F(x) = 1$ and $\lim_{x\to-\infty} = 0$.

A random variable X is said to be discrete if it can take on at most a countable number of values x_1, x_2,.... The probability that the discrete random variable X takes on the value x_i is given by

$$p(x_o) = P[X = x_i] \text{ for } i = 1, 2, \ldots$$

and we must have

$$\sum_{i=1}^{\infty} p(x_i) = 1.$$

$p(x)$ is called the probability!mass function for the discrete random variable X. If $I = [a, b]$, where a and b are real numbers such that $a \leq b$, then

$$P[X \in I] = \sum_{a \leq x_i \leq b} p(x_i).$$

The distribution function $F(x)$ for the discrete random variable X is given by

$$F(x) = \sum_{x_i \leq x} p(x_i), \text{ for all } -\infty < x < \infty.$$

A random variable X is said to be continuous it there exists a nonnegative function $f(x)$ such that for any set of real numbers B,

$$P[X \in B] = \int_B f(x)dx \quad \text{quad} \quad \int_{-\infty}^{\infty} f(x)dx = 1.$$

Two probability measure $\tilde{P}$ and P on $(\Omega, \mathcal{F})$ are said to be equivalent if, for any event $A \in \mathcal{F}$, $P[A] = 0$ if and only if $\tilde{P}[A] = 0$. In other terms, $\tilde{P}$ and P are equivalent on $(\Omega, \mathcal{F})$ if they have the same set of null events in the σ-field $\mathcal{F}$.

7.2.2 Conditional probability

The conditional probability of an event A given an event B is defined as

$$P[A|B] = \frac{P[A \cap B]}{P[B]}.$$

The difficulty with this definition arises when the event B is too small to have a non-zero probability. In order to circumvent this difficulty, we can defined the regular conditional probability as follows. Let $(\Omega, \mathcal{F}, P)$ be a probability space, and let $T : \Omega \to E$ be a random variable, defined as a Borel-measurable function from Ω to its state space $(E, \mathcal{E})$. Then a regular conditional probability is defined as a function $\nu : E \times \mathcal{F} \to [0, 1]$, called a "transition probability", where $\nu(x, A)$ is a valid probability measure (in its second argument) on $\mathcal{F}$ for all $x \in E$ and a measurable function in E (in its first argument) for all $A \in \mathcal{F}$, such that for all $A \in \mathcal{F}$ and all $B \in \mathcal{E}$

$$P[A \cap T^{-1}(B)] = \int_B \nu(x, A)\, dP[T^{-1}(x)].$$

The measurable space $(\Omega, \mathcal{F})$ is said to have the regular conditional probability property if for all probability measures P on $(\Omega, \mathcal{F})$, all random variables on $(\Omega, \mathcal{F}, P)$ admit a regular conditional probability.

7.2.3 Notations and naming scheme

Consider a random variate X. Each function definition starts with `st_`, as a reference to the library. For analytical distributions, it is then followed by the distribution name, an underscore, and the function purpose. As for generic utilities, only purpose follows. The main utilities are:

pdf : probability distribution fuction (continuous random distribution);

pmf : probability (mass) function (discrete random distribution);

cdf : (cumulative) distribution function.

$$F_X(x) = P[X \leq x] = \begin{cases} \int_{-\infty}^{x} f(y)dy & \text{(continuous case).} \end{cases}$$

We will frequently use $F(x)$ instead of $F_X(x)$, when no confusion is present.

mean : distribution mean if it exists, or in other terms $E[X]$;

median : the quantile that is exceeded with probability 0.5;

variance : distribution variance;

skewness : distribution skewness $\nu = E[(X-\mu)^3]/\sigma^3$, measuring the asymmetry of the distribution.

When we give the density or probability mass function for a range of values, it is implicitely assumed to be null outside this range.

7.2.4 References

There exist various textbooks concernings statistical distributions and tests. The implemented distributions are mainly based on Evans et al. [3] book.

7.3 Descriptive statistics

Descriptive statistics are used to describe the main features of a collection of data in quantitative terms. The functions below allow the computation of basic ones: mean, median, variance, standard deviation and quantiles. All these functions assume that the data are summarized in an observations array, and sometimes that this array is sorted in the range of interest.

7.3.1 Mean, median, variance, standard deviation

Assume that we have a collection of n observations $x_1, \ldots, x_n$, that we denote by $\mathcal{X}$:

$$\mathcal{X} \stackrel{def}{=} \{x_1, \ldots, x_n\}.$$

The empirical arithmetic mean of $\mathcal{X}$ is defined as

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i.$$

Assume that the observations are stored in the array `obs`, using the increment `inc`, and that we consider observations from lower index `low` (included) to upper index `up` (excluded). These index have to be multiplied by `inc` in order to obtain the array position (see Section 5.4.1). The empirical mean $\overline{x}$ can be obtained using the function

```
double st_mean(double *obs, int inc, int low, int up);
```

In probability theory and statistics, a median is described as the number separating the higher half of a sample, a population, or a probability distribution, from the lower half. The median of a finite list of numbers can be found by arranging all the observations from lowest value to highest value and picking the middle one. If there is an even number of observations, this middle is not well defined. The median is then usally computed as

$$\overline{x}_{\text{median}} = \frac{x_{n/2} + x_{n/2+1}}{2}.$$

Using the same representation than before, we can compute $\overline{x}_{\text{median}}$ with the function

```
double st_median(double *obs, int inc, int low, int up);
```

If you want to restrict the median to an observation present in the array, use the following function, that returns the first observation in the half upper part of the array.

```
double st_median_observation(double *obs, int inc, int low, int up);
```

The empirical variance

$$\hat{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2,$$

the usual estimator of the variance

$$\mathrm{Var}(X) = E[(X - E[X])^2],$$

can be computed using the function

```
double st_variance(double mean, double *obs, int inc, int low, int up);
```

Finally, the following function computes both empirical mean and variance.

```
void st_mean_variance(double *obs, int inc, int low, int up,
                      double *mean, double *variance);
```

Consider now a second set of observation $\mathcal{Y} \stackrel{def}{=} \{y_1, \ldots, y_n\}$. The empirical covariance between the sets $\mathcal{X}$ and $\mathcal{Y}$ is given by

$$s_{xy} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y}),$$

and can be computed with the function

```
double st_covariance(int n, double *x, int incx, double meanx,
     double *y, int incy, double meany,
     int l, int u);
```

The parameters l and u correspond to the index of the lowest observation to take into consideration, and the of the first observation to discard, respectively.

7.3.2 Quantiles

The k^{th} "q"-quantile of the population parameter X can be defined as the value "x" such that:

$$P[X \leq x] \geq p \text{ and } P[X \geq x] \geq 1 - p,$$

where

$$p = \frac{k}{q},$$

or equivalently,

$$P[X < x] \leq p \text{ and } P[X > x] \leq 1 - p.$$

7.4 Estimator properties

7.4.1 Bias

An estimator X of a fixed but unknown quantity, θ, is a random variable of vector who associates to the data value that is supposed to approximate the true value θ. The bias is defined as the difference between the expectation of the estimator and the true parameter:

$$B = E[X] - \theta.$$

An estimator is said to be unbiased if $B = 0$. This difference can be approximated with the function

```
double st_bias(double estimator, double value);
```

where `estimator` is $E[X]$ and `value` is θ. The relative bias is defined by dividing the bias by the value of θ, if $\theta \neq 0$:

$$B_{\text{rel}} = \frac{E[X] - \theta}{\theta}.$$

It can be approximated with the function

```
double st_relative_bias(double estimator, double value);
```

If θ is null, the standard bias is returned.

7.4.2 Error

The mean square error of an estimator is a measure that combines the bias and the variance, since both of them usually plays a significant rule in the desired accuracy. We will denote it by MSE, and formally define it as

$$\text{MSE}[X] = E[(X - \mu)^2] = \beta^2 + \sigma^2.$$

The MSE can be approximated with the function

```
double st_mse(double bias, double variance)
```

The square root of MSE$[X]$ is called the absolute error, and can be obtained with the function

```
double st_absolute_error(double bias, double variance)
```

The relative error of X is

$$\mathrm{RE}[X] = \sqrt{\mathrm{MSE}[X]}/|\theta|, \quad \text{for } \theta \neq 0,$$

and can be approximated with

```
double st_relative_error(double bias, double variance,
                         double value)
```

This function returns the absolute error if $\theta = 0$.

$\sqrt{\mathrm{MSE}[X]}$ is a measure of the statistical accuracy of the estimator X, and $\mathrm{RE}[X]$ is a measure or this prediction relatively to the order of θ.

7.5 Univariate distributions

7.5.1 Bernouilli distribution

Bernouilli distribution	`bernouilli(p)`
Range	$x \in \{0, 1\}$.
Probability function	$p(x) = \begin{cases} 1-p & \text{if } x = 0, \\ p & \text{if } x = 1, \end{cases}$

7.5.2 Cauchy distribution

Cauchy distribution	`cauchy(a,b)`
Range	$-\infty < x < \infty$.
Parameters	a: location parameter, the median;
	b: scale parameter.
Distribution	$\frac{1}{2} + \frac{1}{\pi}\arctan\left(\frac{x-a}{b}\right)$
Density	$\frac{1}{\pi b\left(1+\left(\frac{x-a}{b}\right)^2\right)}$
Moments	do not exist.
Median	a
Mode	a

Table 7.1: Cauchy distribution

The Cauchy distribution is unimodal and symmetric, with much heavier tails than the normal. It has no moments. The probability density function is symmetric about a, with upper and lower quartiles, $a \pm b$. The related routines are

```
double st_cauchy_cdf(double x, double a, double b)
double st_cauchy_icdf(double u, double a, double b)
double st_cauchy_median(double a, double b)
double st_cauchy_mode(double a, double b)
```

7.5.3 Chi square distribution

χ^2 distribution	`chi2(k)`
Parameter	k: number of degrees of freedom
Distribution	
Density	

```
double st_chi2_cdf(double x, double k);
double st_chi2_pdf(double x, double k);
double st_chi2_mean(double k)
double st_chi2_variance(double k)
double st_chi2_skewness(double k);
```

7.5.4 Erlang distribution

Erlang distribution	`exponential(alpha, lambda)`
Range	$0 \leq x < \infty$.
Parameters	$\alpha \in \mathbb{N}_0$: shape parameter;
	$\lambda > 0$: scale parameter.
Distribution	$1 - e^{-\lambda x} \left(\sum_{i=0}^{\alpha-1} \frac{(\lambda x)^i}{i!} \right)$
Density	$\frac{\lambda(\lambda x)^{\alpha-1} e^{-\lambda x}}{(\alpha-1)!}$
Mean	$\frac{\alpha}{\lambda}$
Variance	$\frac{\alpha}{\lambda^2}$

Table 7.2: Erlang distribution

The Erlang variate is a gamma variate (see Section 7.5.7) with shape parameter α an integer; it also corresponds to the sum of α i.i.d. exponential variables, with parameter λ.

```
double st_erlang_pdf(double x, double alpha, double lambda);
double st_erlang_mean(double alpha, double lambda);
double st_erlang_stdv(double alpha, double lambda);
double st_erlang_variance(double alpha, double lambda);
```

7.5.5 Exponential distribution

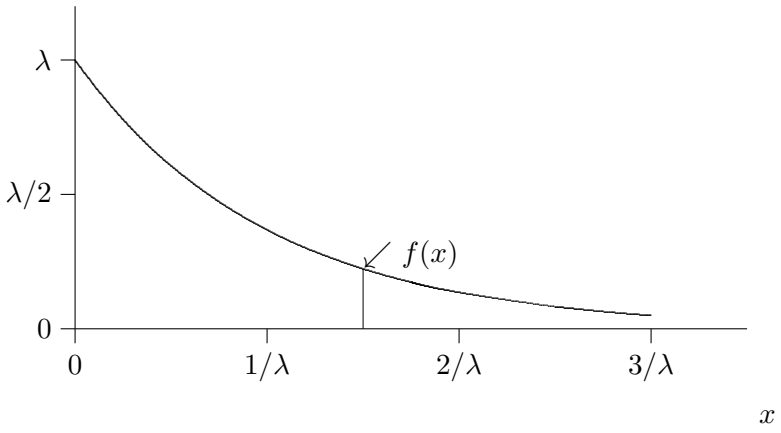
Exponential distribution	`exponential(lambda)`
Range	$0 \leq x < \infty$.
Parameter	$\lambda > 0$: scale parameter.
Distribution	$1 - e^{-\lambda x}$
Density	$\lambda e^{-\lambda x}$
Mean	$\frac{1}{\lambda}$
Variance	$\frac{1}{\lambda^2}$

Table 7.3: Exponential distribution

It's the only continuous distribution which is memoryless, that is

$$P[X > t + x \mid X > t] = \frac{P[X > t + x]}{P[X > t]} = \frac{e^{-\lambda(t+x)}}{e^{-\lambda t}} = e^{-\lambda x} = P[X > x].$$

```
double st_exponential_pdf(double x, double lambda);
double st_exponential_cdf(double x, double lambda);
double st_exponential_icdf(double u, double lambda);
double st_exponential_mean(double lambda);
double st_exponential_stdv(double lambda);
double st_exponential_variance(double lambda);
```




```
double st_exponential_pdf(double x, double lambda);
```

7.5.6 Fisher distribution

```
double st_fisher_pdf(double x, double a, double b)
```

```
double st_fisher_mean(double x, double a, double b)

double st_fisher_variance(double x, double a, double b)
```

$$F(x,\lambda) = \begin{cases} 1 - e^{-\lambda x}, & \text{for } x \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

```
double st_exponential_cdf(double x, double lambda);
```

7.5.7 Gamma distribution

Gamma distribution	`gamma(alpha,lambda)`
Range	$0 < x < \infty$.
Parameters	$\alpha > 0$: shape parameter;
	$\lambda > 0$: scale parameter.
Density	$\frac{\lambda^\alpha x^{\alpha-1} e^{-\lambda x}}{\Gamma(\alpha)}$
Mean	$\frac{\alpha}{\lambda}$
Variance	$\frac{alpha}{\lambda^2}$

Table 7.4: Gamma distribution

Γ is the , defined by

$$\Gamma(\alpha) = \int_0^\infty x^{\alpha-1} e^{-x} dx.$$

In particular, $\Gamma(n) = (n-1)!$ when n is a positive integer.

The probability distribution function of a variate $X \sim \text{Gamma}(\alpha, \lambda)$ is given by the function

```
double st_gamma_pdf(double x, double alpha, double lambda);
```

The cumulative distribution cannot be expressed in closed form but can be expressed in terms of the gamma function parameterized in terms of the shape parameter α and scale parameter λ:

$$F(x;\alpha,\theta) = \int_0^x f(u;\alpha,\lambda)\, du = \frac{\gamma(\alpha,\lambda x)}{\Gamma(\alpha)}.$$

where $\gamma(\alpha, \lambda x)$ is the incomplete gamma function.

The mean, variance, and standard deviation can be obtained with the functions

```
double st_gamma_mean(double alpha, double lambda);
double st_gamma_variance(double alpha, double lambda);
double st_gamma_stdv(double alpha, double lambda);
```

Similarly, the mode can be computed with

```
double st_gamma_mode(double alpha, double lambda);
```

There is no formal expression for the median.

As special cases, $X \sim \text{Exponential}(\lambda)$ when $\alpha = 1$ and X follws the chi-square distribution with 2α degrees of freedom when 2α is an integer and $\lambda = 1/2$. If $X \sim \text{Gamma}(\alpha, \lambda)$, then $1/X \sim \text{Pearson} - 5(\alpha, \lambda)$ and reciprocaly. If $X_1 \sim \text{Gamma}(\alpha_1, \lambda)$ and $X_2 \sim \text{Gamma}(\alpha_2, \lambda)$ are independent, then $X_1/(X_1 + X_2) \sim \text{Beta}(\alpha_1, \alpha_2)$. If $X_1, \ldots, X_k$ are independent random variables and $X_i \sim \text{Gamma}(\alpha_i, \lambda)$ for each i, then $X = X_1 + \ldots + X_k \sim \text{Gamma}(\alpha, \lambda)$ where $\alpha = \alpha_1 + \ldots + \alpha_k$. In particular, if $X_1, \ldots, X_k$ are independent exponentials with the same parameter λ, then $X = X_1 + \ldots + X_k$ is a $\text{Gamma}(k, \lambda)$ random variable. In this case, the $\text{Gamma}(k, \lambda)$ distribution is also called the $\text{Erlang}(k, \lambda)$ distribution.

The gamma distribution function does not have a closed-form expression in general, but if $X \sim \text{Erlang}(k, \lambda)$, then $P[X \leq x] = P[Y \geq k]$ where $Y \sim \text{Poisson}(\lambda x)$. Therefore, in that case,

$$F(x) = P[X \leq x] = 1 - \sum_{j=0}^{k-1} \frac{e^{-\lambda x}(\lambda x)^j}{j!}, \tag{7.1}$$

for $x > 0$. This can be used to compute $F(x)$ when k is small. Conversely, a good numerical approximation of the gamma distribution function can be used to approximate the Poisson distribution function via (7.1). A good algorithm for generating $\text{Gamma}(\alpha, 1)$ random variates suffices for generating gammas with arbitrary parameters, because λ only changes the scale.

7.5.8 Geometric distribution

A geometric random variable corresponds to a sequence of independent Bernouilli trials, where the probability of success at each trial is p, stopped when the first success is obtained. An important property is the memoryless character: $P[X = y + x | X \geq y] = P[X = x]$. The corresponding routines are

Geometric	`geometric(p)`
Range	$x \in \mathbb{N}$.
Parameter	p: Bernouilli probability parameter, $0 < p < 1$.
Distribution function	$1 - (1-p)^{x+1}$
Probability function	$p(1-p)^x$
Mean	$\frac{1-p}{p}$
Variance	$\frac{1-p}{p^2}$
Mode	0

An alternative form of the geometric distribution involves the number of trials up to and including the first success. The distribution function is then $p(1-p)^{x-1}$, for $x > 0$. The geometric distribution is sometimes called the Pascal distribution.

7.5.9 Gumbel distribution

The Gumbel distribution is also known as the Extreme Value distribution (type I), whose main characteristics are given below.

Gumbel	`gumbel(a,b)`
Range	$-\infty < x < \infty$.
Parameters	a: location parameter; $b > 0$: scale parameter.
Density	$f(x) = e^{-e^{-\frac{x-a}{b}}}$
Mean	$a - b\Gamma'(1)$
Variance	$\frac{b^2\pi^2}{6}$
Mode	a
Median	$a - b \ln \ln 2$

$\Gamma'(1)$ is the first derivate of the gamma function $\Gamma(n)$ with respect to n at $n = 1$, and its value is approximately -0.5772156. Its opposite is also known as the Euler constant, available in the `ntools` module under the constant name `M_EULER`.

Gumbel routines rely on the mode a and the scale parameter b. The cumulative distribution can be evaluated at x with the function

```
double st_gumbel_cdf(double x, double a, double b)
```

and its inverse can be computed with the function

```
double st_gumbel_icdf(double u, double a, double b)
```

where u is the value of the cumulative distribution function. The probability distribution function can be computed at x with the following function.

```
double st_gumbel_pdf(double x, double a, double b)
```

The mean, variance, median, and mode can be obtained with the functions below.

```
double st_gumbel_mean(double a, double b)
double st_gumbel_variance(double a, double b)
double st_gumbel_mode(double a, double b)
double st_gumbel_median(double a, double b)
```

7.5.10 Lognormal distribution

Variate $\boldsymbol{L} : \mu, \sigma$.
Two parametrisations exist for a lognormal variate $\boldsymbol{L}$. The lognormal variate with mediam m and with σ denoting the standard deviation of $\log \boldsymbol{L}$ is expressed by $\boldsymbol{L} : m, \sigma$. We choose here the alternative parametrisation $\boldsymbol{L} : \mu, \sigma$, where μ is the mean of $\log \boldsymbol{L}$.
Range $0 \leq x < \infty$.

Lognormal	`lognormal(mu,sigma)`
Distribution	$F(x) = \frac{1}{2} + \frac{1}{2}\text{erf}\left(\frac{\log x - \mu}{\sqrt{2\pi}}\right)$
Density	$f(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{\log x - \mu}{\sigma}\right)^2}$

In the previous relations, `erf` is the error function of x, defined as

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

```
double st_lognormal_pdf(double x, double mu, double sigma)
```

While we use the above form, we could also rely on the Gaussian cumulative function, since it can easily be shown that:

$$F(x) = P[L \leq x] = P\left[X \sim N(0,1) \leq \frac{log(x) - \mu}{\sigma}\right].$$

```
double st_lognormal_cdf(double x, double mu, double sigma);

double st_lognormal_icdf(double q, double mu, double sigma)

double st_lognormal_mean(double mu, double sigma)
```

```
double st_lognormal_stdv(double mu, double sigma)

double st_lognormal_variance(double mu, double sigma)
```

Mean	$e^{\mu+\frac{\sigma^2}{2}}$
Variance	$e^{2\mu+\sigma^2}\left(e^{\sigma^2}-1\right)$
Mode	$e^{\mu-\sigma^2}$
Median	e^{μ}

7.5.11 Normal distribution

```
double st_gauss_pdf(double x)

double st_normal_pdf(double x, double mu, double sigma)

double st_gaussian_cdf(double x, double *result, double*ccum)
```

If x is large (say greater than 5) and that you want to compute

$$1-\Phi(x)=\int_x^{\infty} f(x)dx,$$

it is highly recommanded that you compute $\Phi(-x)$ as a signigicant loss of occuracy can otherwise occur. You can also use the following function, computing the complementary cumulative distribution function:

```
double st_gaussian_ccdf(double x, double *result, double*ccum)
```

Marsaglia [11]

The routine `st_gaussian_icdf` is a C impementation of the method proposed by Wichura [19]. This method is accurate to about 16 digits, for $10^{-316} < \min\{p, 1-p\}$, where p is the point where we want to compute the inverse cumultative function. If p is very close to unity, a serious loss of significance may be incurred in forming $c = 1 - p$. In this circumstance, the user should, if possible, evaluate c directly, and evaluate z_p as $-z_c$. The function definition is

```
double st_gaussian_icdf(const double p);
```

The extension to a normal $N(\mu, \sigma)$ is is immediate:

```
double st_normal_icdf(const double p, double mu, double sigma);

double st_normal_cdf(double x, double mu, double sigma)
```

7.5.12 Poisson distribution

Variate X: λ.
Range $0 \leq x < \infty$, $x \in \mathcal{N}$.
Parameter the mean, $\lambda > 0$.

Probability function

$$f(x) = P[X = x] = \lambda^x \frac{e^{-\lambda}}{x!}.$$

```
double st_poisson_pmf(int x, double lambda)
```

Cumulative distribution function

$$F(x) = P[X \leq x] = \sum_{i=0}^{x} \lambda^i \frac{e^{-\lambda}}{i!}$$

```
double st_poisson_cdf(int x, double lambda)
```

7.5.13 Student distribution

Variate X: ν.

Cumulative distribution function

$$F(x) = \frac{1}{2} + \frac{x\Gamma((\nu+1)/2\,{}_2F_1(1/2,(\nu+1)/2;3/2;-x^2/\nu))}{\sqrt{\pi\nu}\Gamma(\nu/2)},$$

where ${}_2F_1$ is the hypergeometric function.

```
double st_student_cdf(double x, double nu)
```

7.5.14 Triangular distribution

The triangular distribution is unimodal, with a continuous probability density function that is piecewise linear. The pdf is increasing on $[a, c]$ and decreasing on $[c, b]$, and 0 elsewhere.

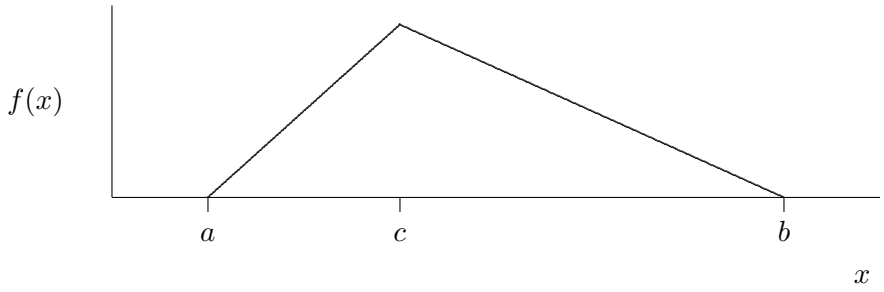


Figure 7.1: Triangular distribution

Triangular	`triangular(a, b, c)`
Range	$a \leq x \leq b$.
Parameters	a: lower limit. $b > a$: upper limit. c, $a \leq c \leq b$: mode.
Density	$f(x) = \begin{cases} \frac{2}{b-a}\frac{x-a}{c-a} & \text{if } a \leq x \leq c, \\ \frac{2}{b-a}\frac{b-x}{b-c} & \text{if } c \leq x \leq b, \\ 0 & \text{otherwise.} \end{cases}$
Distribution function	$F(x) = \begin{cases} \frac{(x-a)^2}{(b-a)(c-a)} & \text{if } a \leq x \leq c, \\ 1 - \frac{(b-x)^2}{(b-a)(b-c)} & \text{if } c \leq x \leq b, \\ 0 & \text{otherwise.} \end{cases}$
Mean	$\frac{a+b+c}{3}$
Variance	$\frac{a^2+b^2+c^2-ab-ac-bc}{18}$
Mode	c
Median	$\begin{cases} a + \sqrt{\frac{(b-a)(c-a)}{2}} & \text{if } c \geq \frac{b-a}{2} \\ b - \sqrt{\frac{(b-a)(b-c)}{2}} & \text{if } c \leq \frac{b-a}{2} \end{cases}$

The related routines are

```
double st_triangular_pdf(double x, double a, double b, double c);
```

The cumulative distribution can be be evaluated with the function

```
double st_triangular_cdf(double x, double a, double b, double c);
```

while the p-quantile can be obtained with

```
double st_triangular_icdf(double p, double a, double b, double c);
```

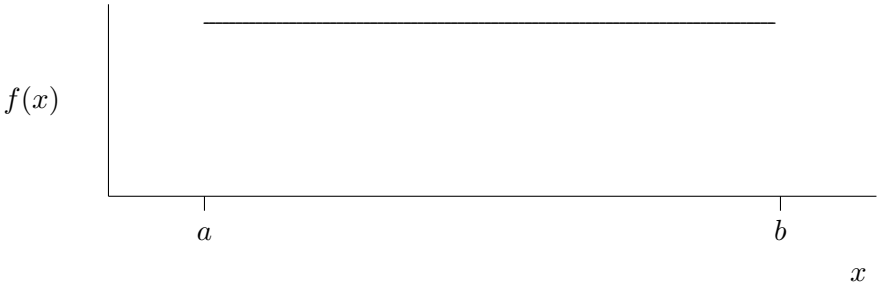
```
double st_triangular_mean(double a, double b, double c);

double st_triangular_variance(double a, double b, double c);

double st_triangular_stdv(double a, double b, double c);

double st_triangular_mode(double a, double b, double c);
```

7.5.15 Uniform distribution

$E[X] = (b-a)/2$ and $\mathrm{Var}[X] = (b-a)^2/12$.

Probability function

$$f(x) = \frac{1}{b-a} \; \forall \; a < x < b.$$




```
double st_uniform_pdf(double x, double a, double b)

double st_uniform_mean(double a, double b)

double st_uniform_variance(double a, double b)
```

7.6 Multivariate distributions

7.6.1 Mutlivariate normal

The standard mutlivariate distribution is obtained by fixing the mean vector to 0 and the variance-covariance matrix Σ to the identity matrix I. Therefore, if X follows a standard multivariate normal distribution, we will write $X \sim N(0, I)$. The density of X at a point x can be computed with the function

```
double st_mvn_standard_pdf(int s, double *x, int incx);
```

Multivariate normal distribution	`mvn(mu,Sigma)`	
Range	$\mathcal{R}^s$	
Density	$f_X(x_1,\ldots,x_s)$	$=$
	$\frac{1}{(2\pi)^{s/2}\lvert\Sigma\rvert^{1/2}} e^{\left(-\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)\right)}$	

7.7 Additional functions

7.7.1 Truncated distributions

We do not provide explicit truncated distributions, since they can easily be obtained from underlying untruncated distributions. While it is possible to consider various truncation forms, from the definition that a truncated distribution is *"a probability distribution obtained from a given distribution by transfer of probability mass outside a given interval to within this interval"* (Ushakov [18]), we will consider here only the case where the mass concentrated outside $[a, b]$ is distributed over the whole of [a, b]. In the case, the cumulative distribution function becomes

$$F_{a,b}(x) = \begin{cases} 0 & x \leq a, \\ \frac{F(x)-F(a)}{F(b)-F(a)} & a < x \leq b, \\ 1 & x > b. \end{cases}$$

The probabibility function can then be easily deduced to be

$$f_{a,b}(x) = \begin{cases} \frac{f(x)}{F(b)-F(a)} & x \in [a, b], \\ 0 & \text{otherwise.} \end{cases}$$

These two functional transformations are implemented by the funcions

```
double st_truncated_pdf(double fx, double Fmin, double Fmax)
```

and

```
double st_truncated_cdf(double Fx, double Fmin, double Fmax)
```

where `Fmax` is $F(b)$, `Fmin` is $F(a)$, `fx` is $f(x)$ and `Fx` is $F(x)$. The returned values are repsectively the cdf and pdf values of the truncated distribution.

7.7.2 Percentiles

7.7.3 Histogram

7.8 Random vectors

A vector $\boldsymbol{X} = (X_1, \ldots, X_d)^T$ follows a multivariate law of cumulative distribution function F if for all $\boldsymbol{x} = (x_1, \ldots, x_d)^T \in \mathcal{R}^d$, we have

$$F(\boldsymbol{x}) = P[\boldsymbol{X} \leq \boldsymbol{x}] = P[X_1 \leq x_1, \ldots, X_d \leq x_d].$$

We will also define marginal laws by considering the cumulative function associated to each component of the random vector:

$$F_j(x) = P[X_j \leq x],\ j = 1, \ldots d.$$

7.9 Empirical distribution

Given the observations $x_1, \ldots, x_n$, that we sort in ascending order to obtain $x_{(1)}, \ldots, x_{(n)}$, the empirical function is defined as follows:

$$\hat{F}_n(x) = \frac{1}{n} \sum_{i=1}^{n} I[x_i \leq x].$$

Figure 7.2 illustrates it.

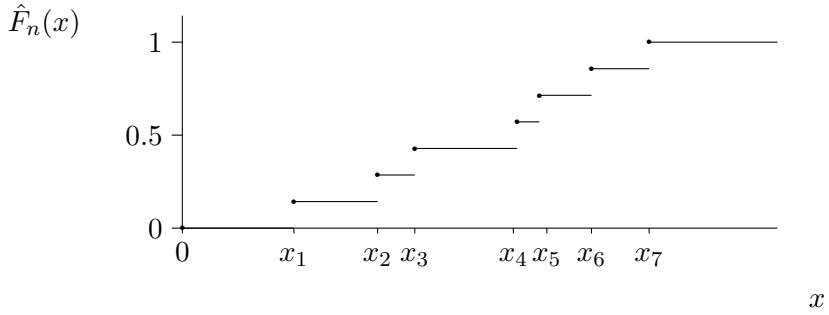


Figure 7.2: Fonction de répartition empirique

```
void st_export_empirical_cdf(FILE *f, int n, double *v, int sorted)
```

7.10 Bootstrap

7.11 Goodness-of-fit tests

7.11.1 Test for fit of a distribution

A classical statistical test is the comparison of numbers of observations in some categories to expected number of observations in these categories under the assumption that these observations follow a given distribution. More precisely, let $\{x_1 \dots x_n\}$ be a sample of n independent realisations of the random variable X. Let $L(x)$ be the unknown distribution of X. The hypothesis H_0 is that the data are drawn from a distribution $L^*(x)$, i.e.:

$$H_0 : L(x) = L^*(x), H_1 : L(x) \qquad \neq L^*(x).$$

The reduction in the degrees of freedom is calculated as $p = s+1$, where s is the number of parameters used in fitting the distribution. Therefore, there will be $(n-p)$ degrees of freedom, where n is the number of categories. The value of the test-statistic is

$$X^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}, \tag{7.2}$$

where

- O_i = an observed frequency;
- E_i = an expected (theoretical) frequency, asserted by the null hypothesis;
- n = the number of possible outcomes of each event (i.e. the number of categories).

Under H_0, we can consider that $E_i - O_i$ is normally distributed. X^2 tends to a χ^2 distribution whi $\nu = n - p$ degrees of freedom. We accept the null hypothesis H_0 at level α if the probability that a chi^2 random variable falls in the interval $(0, \chi^2_{\nu,1-\alpha})$ is at least eqault to $1 - \alpha$. We reject H_0 if the value of X^2 is greater than $\chi^2_{\nu,1-\alpha}$.

```
int st_pearson_chi_test(int n, double *obs, int ldo, double *exp, int lde,
                        double alpha, double *x2, int df);
```

where n is the number of categories, `obs` and `exp` are the numbers of observations and expected numbers for each category; `obs` and `exp` are tabulare with `ldo` and `lde` increments respectively. α is the lest level, and `df` the number of degrees of freedom. On output, $x2$ contains the value of the statistic 7.2. The function returns 0 if we reject H_0, 1 otherwise.

Chapter 8

Random numbers library

8.1 Introduction

8.1.1 Supported distributions

Distributions that are currently supported are summarized in Table 8.1.1. At each distribution is associated a numerical constant. The total number of distribution can be obtained by taking `RAN_CONSTANT_END`-1, and a constant `RAN_CONSTANT` is also defined. A multivariate distribution can be identified with the prefix `m_` before the distribution name.

The number of parameters associated to some distribution can be obtained by postfix `_NPAR` to the numerical constant name. The library supports various continuous distributions, as well as some discrete ones. This is however only valid for univariate distributions since the number of parameters for multivariate distributions depends on the vector dimension.

8.1.2 Terms and symbols

We use these notational conventions:

- $\mathcal{N}$: natural set.
- $\mathcal{N}_0$: natural set, with the exclusion of 0.

8.2 Uniform random numbers generation

A good uniform random generator on the interval $[0, 1]$ is the major component of any good random generator library. Draws from other distributions

Distribution	Identifier	#Parameters
Univariate distributions		
	Discrete distributions	
Bernouilli	`RAN_BERNOUILLI`	1
Geometric	`RAN_GEOMETRIC`	
Integer in $[0, \max[$	`RAN_INTEGER`	
	Continuous distributions	
Arcsinus	`RAN_ARCSINE`	
Cauchy	`RAN_CAUCHY`	
Exponential	`RAN_EXPONENTIAL`	1
Gamma	`RAN_GAMMA`	
Gumbel	`RAN_GUMBEL`	2
Johnson	`RAN_JOHNSON`	
Lognormal	`RAN_LOGNORMAL`	
Normal	`RAN_NORMAL`	
Student	`RAN_STUDENT`	
Triangular	`RAN_TRIANGULAR`	
Truncated normal	`RAN_TRUNCATED_NORMAL`	4
Truncated normal B	`RAN_TRUNCATED_NORMAL_B`	4
Uniform on (0,1)	`RAN_UNIFORM`	0
Multivariate distributions		
	Continuous distributions	
Multivariate normal	`RAN_M_NORMAL`	
Unit sphere	`RAN_M_UNIT_SPHERE`	

Table 8.1: Supported distributions.

are usually obtained by adequately transform an uniformly distributed sample (see Levroye [10] for a survey of these techniques). Many researches have been conducted to develop good random generators, and lead to different algorithms. The search is however far from be over and improved generators will probably be released (see for instance L'Ecuyer [7]). The reader is therefore invited to include such algorithms if she/he feels the need to do it. Algorithms can be tested with the package TestU01 (L'Ecuyer et Simard [8]).

The basic principle of a uniform random numbers generator is to define a transition function $f : \mathcal{S} \to \mathcal{S}$, where $\mathcal{S}$ is the state space. The cardinality of $\mathcal{S}$ is assumed to be finite. The initial state is denoted by s_0, and we will

write

$$s_n = f(s_{n-1}).$$

We will furthermore assume that f is periodic for all n greater of equal to some known τ (often equal to 0), with the period denoted by ρ. In other terms, we have $s_{n+\rho} = s_n$ for all $n \geq \tau$. We denote the output space by $\mathcal{U}$, and we assume here that $\mathcal{U} = (0, 1)$. The output function $g : \mathcal{S} \to \mathcal{U}$ transforms the state s_n into an output value u_n. This behavior is graphically illustrated in Figure 8.2, with $\tau = 0$.

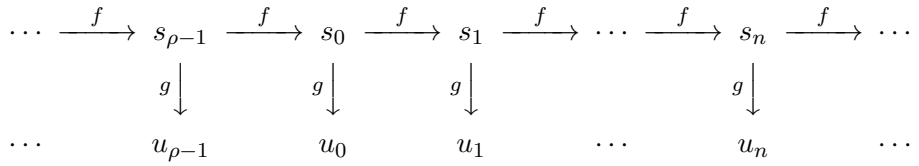


Figure 8.1: Behavior of a uniform random numbers generator

The basic function is

```
Random *ran_random_new(int id, unsigned long *seed, int n);
```

This create an object of type `Random *`. `seed` is a vector of integers that will be used to initialize the state of the random numbers generator, that is dependent of the underlying $U(0, 1)$ generator (see Section 8.2). If the seed is set to NULL, a default seed will be used. The length of this vector also depends of the $U(0, 1)$-generator, but each generator can be initialized with a seed dimension equal to 1. If no specific random numbers generator is specified, i.e. if `id` is set to 0, the `RAN_DEFAULT` generator is used. A specific random numbers generator can be obtained by using the identifier given in Table 8.2. These generators are discussed in more details in the rest of the document.

A simplified random numbers constructor is

```
Random *ran_random();
```

which is equivalent to `ran_random_new(0, NULL, 0);\verb` A copy of the initial seed is stored inside the generator, which can be retrieved with the function

```
unsigned long *ran_random_get_seed(const Random *random);
```

Name	Identifier	Seed size
Default generator	`RAN_DEFAULT`	`RAN_DEFAULT_SEED_LENGTH`
Standard minimal	`RAN_STANDARDMINIMAL`	1
MRG32K3a	`RAN_MRG32K3A`	6
MT19937	`RAN_MT19937`	$[1, \ldots, 624]$

Table 8.2: uniform random numbers generators

You can also change this seed using the following function:

```
void ran_random_set_seed(Random *random, unsigned long *seed, int n)
```

The rng internal seed is stored independently of the given parameter seed.

The state structure is specific to each uniform random generator, so the generic function uses the type `void *`. The reader should refer to the corresponding generator to learn which structure has to used. If this pointer is `NULL`, an arbitrary initial state will be generated using the internal clock, and this value is place in the address pointed by `state`. The state can also be retrieved with the function

```
void *ran_random_get_state(Random *random);
```

The memory associated to the generator has to be freed with the function

```
void ran_random_free(Random *random);
```

The name of the currently used generator can be obtained with the function

```
char *ran_random_generator_name(Random *random);
```

You can use a different algorithm for random generation, by calling the function

```
void random_set_generator(Random *random, int id,
                          unsigned long *seed, int n)
```

`id` is a constant describing the generator to use and seed is an integers vector of dimension n, used to define the initial state, as described in Section 8.2. Table 8.2 gives supported generators and the corresponding constants, as well as the corresponding seed vector size.

8.2.1 Linear Congruential Generators

Linear congruential generators (LCGs) have been introduced by Lehmer [9]. A sequence of integers $Z_1, Z_2, \ldots$ is defined by the recursive formula

$$Z_i = (aZ_{i-1} + c) \mod m. \tag{8.1}$$

A first desirable property of a LCG is that it has full period. The following theorem, proved by Hull and Dobell [6], gives such a characterization.

Theorem 1. *The LCG defined in* (8.1) *has full period if and only if the following three conditions hold:*

1. *the only positive integer that (exactly) divides both m and c is 1;*
2. *if q is a prime number that divides m, then q divides $a - 1$;*
3. *if 4 divides m, then 4 divides a-1.*

If $c = 0$, a period equal to m cannot be achieved, but under some conditions, we can obtain a period of $m - 1$. For m prime, it can be shown that the period is $m - 1$ if a is a primitive element modulo m, that is the smallest integer l for which $a^l - 1$ is divisible by m is $l = m - 1$ (Knuth, 1988, p20). LCGs that fullfil these conditions are called prime modulus multiplicative LCGs (PMMLCGs).

A generic linear congruential generator can be created using the function

```
LCG *ran_lcg_new(unsigned long seed);
```

where seed is the initial state of the generator. The allocated memory memory can be released using the fonction

```
void ran_lcg_free(void *lcg);
```

```
LCG *ran_lcg_new_with_state(unsigned long *state);
LCG *ran_lcg_new_with_seed(unsigned long seed);
unsigned long *ran_lcg_get_state(void *lcg);
void ran_lcg_set_state(void *lcg, void *state);
void ran_lcg_init(Random *random, unsigned long seed);
```

By default, the created generator is the Standard Minimal generator (see Section 8.2.1, but it the parameters can be modified using the function

```
void ran_lcg_set_parameters(LCG *lcg, double a, double c, double m);
```

and a draw can be obtained with the function

```
double ran_lcg_get_val(void *lcg);
```

The following code defines the infamous `RANDU` generator:

```
unsigned long m = 2;

r = ran_lcg_new(seed);
for (i = 0; i < 30; i++) m *= 2;
ran_lcg_set_parameters(r, 65539, 0, m);
```

Standard minimal generator

A popular, but outdated random generator is the standard minimal, following the terminology introduced by Park and Miller [16], while it is known to be not optimal (L'Ecuyer and Simard [8]). The standard minimal has been introduced in 1969 by Lewis et al. [15]. It is a linear congruential generator driven by the recurrence

$$x_{n+1} = 16807 x_n \mod (2^{31} - 1).$$

8.2.2 Multiple Recursive Generator

These generators are based on the recurrence

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \mod m.$$

L'Ecuyer algorithm

8.2.3 Generators in $\mathcal{F}_2$

Mersenne twister generator

The Mersenne twister is a pseudorandom number generator developed in 1997 by Makoto Matsumotoand Takuji Nishimura [12]. While it is reputated to be robust and fast, it is not exempt of defaults, since if fails some tests define in TestU01 (L'Ecuyer and Simard [8]). The state length is a vector of 624 integers.

8.3 Non-uniform random numbers generation

While uniform random number generation are of deep importance, the practician will mainly look for other distributions. The main reference for non-uniform random number generation is the book written by Luc Devroye [2], that can be downloaded free of charge from the author's website, at the address `http://cg.scs.carleton.ca/~luc/rnbookindex.html`. A copy can also be found at the adress `http://www.iro.umontreal.ca/~bastin/books/nonuniformrandomvariates.zip`. We have also used Evans, Hastings and Peacock [3] as reference.

The library aims to use a similar naming scheme amongst the various distributions. Each routine name will start by `ran_`, followed by the distribution name and the function of the routine.

8.4 Univariate distributions

8.4.1 Discrete distributions

Bernouilli distribution

A Bernouilli variable can take two value only: 1 (success) or 0 (failure). The probability of success, denoted by p, is the Bernouilli parameter; $0 \leq p \leq 1$.

You can draw some value from a Bernouilli distribution with parameter p using the function:

```
int ran_bernouilli(const Random *random, double p);
```

You can also create a generator more specifically designed for the Bernouilli distribution with the following function:

```
Random *ran_bernouilli_new(unsigned long *seed, int n, double p);
```

A realization can then be obtained by using the function

```
int ran_bernouilli_get_val(const Random *random);
```

The parameter p can be retrieved with

```
double ran_bernouilli_get_param(const Random *random);
```

On the other side, you can define the parameter using

```
double ran_bernouilli_set_param(Random *random, double p)
```

Geometric distribution

X is geometrically distributed with parameter $p \in (0, 1)$ when

$$P[X = i] = p(1 - p)^{(i-1)}.$$

A for the Bernouilli variate, p is the success probability of one trial, so that $0 \leq p \leq 1$. i is number of trials (up to and including the first success), and therefore $i \in \mathcal{N}_0$. The geometric distribution is important because it is the distribution of the waiting time until success in a sequence of Bernouilli trials. The geometric distribution is also sometimes called the Pascal distribution.

You can obtain a draw from a geometric distribution of parameter p with a general Random object by using the function

```
int ran_geometric(const Random *random, double p)
```

The geometric distribtion object can be created with the function

```
Random *ran_geometric_new(unsigned long *seed, int n, double p)
```

A random number can then be drawn with the function

```
int ran_geometric_get_val(const Random *random)
```

The parameter p can be retrieved and set with the functions

```
double ran_geometric_get_param(const Random *random)
```

```
void ran_geometric_set_param(Random *random, double p)
```

Integer numbers

A random integer variable X, associated to the parameter max, is assumed to be a discrete variable that presents the property that

$$P[X = i] = \frac{1}{\max}, \ \forall i \in [0, \max[\cap \mathcal{N}.$$

Variate X: max. Range max $\in \mathcal{N}_0$.

A draw can be generated with the function

```
void ran_intrandom_set_max(Random *ird, unsigned long max)
```

An integer random generator can be created with the function

```
Random *ran_intrandom_new(unsigned long *seed, int n,
                          unsigned long max)

unsigned long ran_intrandom_get_val(const Random *ird)
```

The maximum value can be retrieved with the function

```
unsigned long ran_intrandom_get_max(const Random *ird)
```

8.4.2 Continuous distributions

Arcsine distribution

The arcsine distribution is a special case of the beta distribution with both parameters equal to 1/2. The density function is:

$$f(x) = \begin{cases} \frac{1}{\pi\sqrt{x(1-x)}} & \text{if } 0 \le x \le 1, \\ 0 & \text{otherwise.} \end{cases}$$

You can pick a number from the arcsine with the function

```
double ran_arcsine(const Random *random)
```

It is also possible to create an object more specifically designed for the arcsine distribution with the function

```
Random *ran_arcsine_new(unsigned long *seed, int n)
```

and then use this function to get some value:

```
double ran_arcsine_get_val(const Random *random)
```

Cauchy distribution

```
double ran_cauchy(const Random *random,
                  double a, double b)

Random *ran_cauchy_new(unsigned long *seed, int n
                       double a, double b)

double ran_cauchy_get_val(const Random *random)

double ran_cauchy_standard_get_val(const Random *random)

double ran_cauchy_get_parameters(const Random *random,
                                 double *a, double *b)

double ran_cauchy_set_parameters(Random *random,
                                 double a, double b)
```

Exponential distribution

The exponential distrbution, described in Section 7.5.5, can be generated with the function

```
double ran_exponential(const Random *random, double lambda)
```

where λ is the distribution parameter. The other functions are

```
Random *ran_exponential_new(unsigned long *seed, int n, double lambda)
double ran_exponential_get_val(const Random *random)
double ran_exponential_get_parameters(const Random *random)
void ran_exponential_set_parameters(Random *random, double lambda)
```

Gamma distribution

$$f(x) = \begin{cases} \frac{1}{\Gamma(a)b^a} x^{a-1} e^{-\frac{x}{b}} & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

```
double ran_gamma(const Random *random, double a, double b)

Random *ran_gamma_new(unsigned long *seed, int n,
                      double a, double b)

double ran_gamma_get_val(const Random *random)

double ran_gamma_get_parameters(const Random *random,
                                double *a, double *b)

double ran_gamma_set_parameters(Random *random,
                                double a, double b)
```

Gumbel distribution

You can obtain a value from a Gumbel distribution with parameters a, b, using the function:

```
double ran_gumbel(const Random *random, double a, double b)
```

You can create a generator more specifically designed for the Gumbel distribution with the following function:

```
Random *ran_gumbel_new(unsigned long *seed, int n,
                       double a, double b)
```

Then a value can be obtained by using the function:

```
int ran_gumbel_get_val(const Random *random)
```

The parameters a, b can be retrieved with:

```
void ran_gumbel_get_parameters(const Random *random,
                               double *a, double *b)
```

On the other side, you can define the parameter using:

```
void ran_gumbel_set_parameters(Random *random,
                               double a, double b)
```

Johnson distribution

The Johnson families are part of the most flexible ones, and are able to mimic most of standard distributions. Its general form is

$$f(x) = \frac{\delta}{\lambda\sqrt{2\pi}} g'\left(\frac{x-\xi}{\lambda}\right) e^{-\frac{\left(\gamma+\delta g\left(\frac{x-\xi}{\lambda}\right)\right)^2}{2}},$$

where g is one of the following four transformations:

$$g(y) = \begin{cases} \ln y & \text{lognormal family} \\ \sinh^{-1}(y) = \ln\left(y + \sqrt{y^2+1}\right) & \text{unbounded family} \\ \ln\left(\frac{y}{1-y}\right) & \text{bounded family} \\ y & \text{normal family.} \end{cases}$$

Variate J: ξ, λ, γ, δ.
Localisation parameter ξ.
Scale parameter λ. Form parameters: γ, δ.

The support of the distribution depends on the transformation g. It is $(-\infty, \infty)$ for the unbounded and normal family, $[\xi, \xi+\lambda]$ for the bounded family, and $[\xi, \infty)$ for the lognormal family.

The associated cumulative distribution function is:

$$F(x) = \Phi\left(\gamma + \delta g\left(\frac{x-\xi}{\lambda}\right)\right).$$

Since the normal and lognormal families reduce to normal and lognormal random variables, we consider only bounded and unbounded families. The bounded family will be identified with the letters `SB`, while the unbounded family will be identified with the letters `SU`.

Unbounded family A draw from an unbounded Johnson variate of parameters ξ, λ, γ, δ can be obtained by calling the function

```
double ran_johnsonSU(const Random *random,
                     double xi, double lambda,
                     double gamma, double delta);
```

As usual, you can define a specific generator using the function

```
Random *ran_johnsonSU_new(unsigned long *seed, int n,
                          double xi, double lambda,
                          double gamma, double delta);
```

and then draw on it with repeated call to

```
double ran_johnsonSU_get_val(const Random *random);
```

The parameters can be retrieved with the function

```
void ran_johnsonSU_get_parameters(const Random *random,
                                  double *xi, double *lambda,
                                  double *gamma, double *delta);
```

and set with

```
void ran_johnsonSU_set_parameters(Random *random,
                                  double xi, double lambda,
                                  double gamma, double delta);
```

Bounded family The functions are the same as for the unbounded case, except that the letters `SU` in function names have to be replaced by `SB`. This results in the following set of functions:

```
double ran_johnsonSB(const Random *random,
                     double xi, double lambda,
                     double gamma, double delta);

Random *ran_johnsonSB_new(unsigned long *state,
                          double xi, double lambda,
                          double gamma, double delta);

double ran_johnsonSB_get_val(const Random *random);
```

```
void ran_johnsonSB_get_parameters(const Random *random,
                                  double *xi, double *lambda,
                                  double *gamma, double *delta);

void ran_johnsonSB_set_parameters(Random *random,
                                  double xi, double lambda,
                                  double gamma, double delta);
```

Lognormal distribution

The density function of a log-normal variate X is

$$f(x) = \begin{cases} \frac{1}{x\sqrt{2\pi}\sigma} e^{-\frac{(\log x-\mu)^2}{2\sigma^2}} & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

We then write $X \sim LN(\mu, \sigma^2)$. The distribution function is

$$F(x) = \begin{cases} \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{\ln x-\mu}{\sqrt{2}\sigma}\right)\right) & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

$X \sim LN(\mu, \sigma^2)$ if and only if $\ln X \sim N(\mu, \sigma^2)$.

```
double ran_lognormal(const Random *random,
                     double mu, double sigma)

double ran_lognormal_icdf(const Random *random,
                          double mu, double sigma)

Random *ran_lognormal_new(unsigned long *seed, int n,
                          double mu, double sigma)

double ran_lognormal_get_val(Random *random)

double ran_lognormal_get_val_with_parameters
  (Random *random, double mu, double sigma)

void ran_lognormal_get_parameters
  (const Random *random, double *mu, double *sigma)

void ran_lognormal_set_parameters(Random *random,
                                  double mu, double sigma)
```

Normal distribution

A random variable follows a normal distribution $N(\mu, \sigma^2)$ if its density function is given by:

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

It is possible to use a generic object of type `random` to get values from a normal distribution with the following function:

```
double ran_normal(const Random *random, double mu, double sigma)
```

The method is based on the CDF inversion, which can also be called using the function

```
double ran_normal_icdf(const Random *random, double mu, double sigma)
```

It can however be preferable to define a random generator dedicaced to deliver random values, with the function

```
Random *ran_normal_new(unsigned long *seed, int n,
                       double mean, double stdv)
```

A value can then be obtained with the function

```
double ran_normal_get_val(Random *random)
```

Is it also possible to get values from an normal $N(\mu, \sigma^2)$ with the function

```
double ran_normal_get_val_with_parameters(Random *random,
          double mean, double sigma)
```

where μ is the variance and σ is the standard deviation.

Student distribution

Density function

$$f(x) = \begin{cases} \frac{1}{2}+ & \text{if } \nu \text{ is odd} \\ \frac{1}{2}+ & \text{if } \nu \text{ is even} \end{cases}$$

Truncated normal distribution

The truncated normal distribution is the probability distribution of a normally distributed random variable whose value is either bounded below or above (or both). Assume that Y $N(\mu, sigma^2)$.

Parameters
- μ: mean of Y;
- σ: standard deviation of Y;
- a, b: lower and upper bounds of X, respectively.

Density function

$$f(x) = \frac{\frac{1}{\sigma}\phi\left(\frac{x-\mu}{\sigma}\right)}{\frac{1}{\sigma}\Phi\left(\frac{b-\mu}{\sigma}\right) - \Phi\frac{1}{\sigma}\left(\frac{a-\mu}{\sigma}\right)}$$

Truncated normal distribution with mass points

Triangular distribution

A triangular distribution generator can be obtained with the function

```
Random *ran_triangular_new(unsigned long *seed, int n,
                           double x_min, double x_max, double mode)

double ran_triangular(const Random *random,
      double x_min, double x_max, double mode)

double ran_triangular_get_val(const Random *random)

void ran_triangular_get_parameters(const Random *random,
   double *x_min, double *x_max, double *mode)

void ran_triangular_set_parameters(Random *random,
   double x_min, double x_max, double mode)
```

8.5 Multivariate distributions

Function dealing with multivariate distributions are similare to the univariate case, except the presence of the prefix `m_` in from of the distribution name.

8.5.1 Mulivariate normal

8.5.2 Unit sphere

Let denote by C^m the n-dimensional unit sphere, that is

$$C^n = \{x \in \mathcal{R}^m \text{ s.t. } \|x\|_2 = 1\}.$$

The following function allows the generation of a random vector uniformly distributed on C^n.

A specific uniform random number generator on C^m can be created with the function

```
Random *ran_m_unit_sphere_new(unsigned long *seed, int n, int m)
```

which than allow to draw uniformly on C^m using the function

```
void ran_m_unit_sphere_get_val(Random *ran, int m, double *x)
```

that stores the drawn vector in x. If you prefer to use a generic random numbers generator, the following function achieves the same effect:

```
void ran_m_unit_sphere(Random *ran, int m, double *x)
```

8.6 Special cases

8.6.1 Ratio of distributions

It is sometimes useful to compute the ratio of two random distributions, which can be performed with the following functions.

```
void ran_generate_ratio_distribution(Random *ran1, Random *ran2,
             int n, double *x1, int incx1,
     double *x2, int incx2,
     double *r, int incr)
```

Both random numbers generators `ran1` and `ran2` must of course be initialized, and producing independent sequences (except in very specific cases). `n` designs the number of draws to produces. The vectors x_1 will contain draws from the first distribution, using `ran_1`, using the increment `incx1` (see Section 5.4.1), while x_2 will contain draws produced by `ran_2`, using the increment `incx2`. The ratios between these draws will be stocked in the vector r, using the increment `incr`.

This function can of course be memory consuming as it stores vectors `x1`, `x2` and `r`. A simpler function is

```
inline double ran_generate_ratio(Random *ran1, Random *ran2,
 double *x1, double *x2)
```

This function simply draws one realization from `ran1`, one from `ran2`, stores them in elements `x1` and `x2` respectively, and returns their ratio.

8.6.2 Truncated distributions

For any random variable X, it is possible to truncate its distribution produced to some interval $[a, b]$ when inversion is used as the generation technique. This can be obtained by setting three parameters of the random generator object:

- the desired lower bound of the distribution function, i.e. $F(a)$;
- the desired upper bound of the distribution function, i.e. $F(b)$;
- the probability mass of a, i.e. $P[X = a]$.

These parameters can be set with the function

```
void ran_random_set_truncation(Random *random, double l,
                               double h, double p);
```

l and h correspond to lower and upper bounds respectively, while p is the value $P[X = a]$. By default, $l = 0$, $h = 1$, and $p = 0$, so that no truncation is performed.

In order to illustrate how the truncation is performed, let $\tilde{F}$ denote the distribution F truncated to the interval $(a, b]$, i.e.,

$$\tilde{F}(x) = \frac{F(x) - F(a)}{F(b) - F(a)}$$

for $a \leq x \leq b$, $\tilde{F}(x) = 0$ for $x \leq a$, and $F(x) = 1$ for $x \geq b$. Let $U \sim U(F(a), F(b))$ and $X = F^{-1}(U)$. Since F is monotone, we have

$$P[X \leq x] = P[U \leq F(x)] = \begin{cases} 0 & \text{if } F(x) \leq F(a) (\text{or if } x \leq a), \\ \frac{F(x)-F(a)}{F(b)-F(a)} & \text{if } F(a) \leq F(x) \leq F(b) (\text{or if } a \leq x \leq b), \\ 1 & \text{if } F(x) \geq F(b) (\text{or if } x \geq b). \end{cases}$$

Therefore, X has distribution $\tilde{F}$.

If F has a jump of size p at a and we want to truncate to $[a, b]$, then the algorithm should return a with probability p. To achieve that, it suffices to generate U uniform over $(F(a) - p, F(b))$.

8.7 Stochastic processes

8.7.1 One-dimensional Brownian motion

Definition 8.1: Standard one-dimensional Brownian motion
A standard one-dimensional Brownian motion on $[0, T]$ is a stochastic process $\{W(t),\ 0 \le t \le T\}$ with the following properties

(a) W(0) = 0;

(b) the mapping $t \to W(t)$ is a continuous function on $[0, T]$, almost surely;

(c) the increments

$$\{W(t_1) - W(t_0), W(t_2) - W(t_1), \ldots, W(t_k) - W(t_{k-1}),\}$$

are independent for any k and any $0 \le t_0 < t_1 < \ldots < t_k \le T$;

(d) $W(t) - W(s) \sim N(0, t - s)$, for any $0 \le s < t \le T$.

For constants μ and $\sigma > 0$, a process $X(t)$ is a Brownian motion with drift μ and diffusion coefficient σ^2 (abbreviated $X \sim BM(\mu, \sigma^2)$) is

$$\frac{X(t) - \mu t}{\sigma}$$

is a standard Brownian motion W. Therefore, we may construct X from a standard Brownian motion W by setting

$$X(t) = \mu t + \sigma W(t).$$

It follows $X(t) \sim N(\mu t, \sigma^2 t)$, and X solves the stochastic differential equation

$$dX(t) = \mu dt + \sigma dW(t).$$

We can also start from a non-zero value $X(0) = \alpha$ by simply adding x to each $X(t)$.

8.8 Random permutations

A random permutation, or shuffling, is a random ordering of a set of objects, that is, a permutation-valued random variable. There are two basic

algorithms for doing this, both popularized by Donald Knuth. The first is simply to assign a random number to each object, and then to sort the cards in order of their random numbers. This will generate a random permutation, unless two of the random numbers generated are the same. This can be eliminated either by retrying these cases, or reduced to an arbitrarily low probability by choosing a sufficiently wide range of random number choices.

The second, generally known as the Knuth shuffle or Fisher-Yates shuffle, is a linear-time algorithm which involves moving through the objects set, swapping each object in turn with another object from a random position in the part of the set that has not yet been passed through (including itself). This is the implemented method, that comes in two versions, one for generic pointers arrays, on for double vectors:

```
void ran_shuffle_generic(Random *r, int n, void *x, int incx);
void ran_shuffle_double(Random *r, int n, double *x, int incx);
```

In these functions, `r` is an initialized random numbers generator, n is the number of elements to shuffle, stored or pointed by the vector x, which has an increment `incx`.

8.9 Input-ouput functions

Two simple functions are available to write random draws in an output file `f`. The first function allows to write a random vector v of size d.

```
void ran_write_vector(FILE *f, double *v, int d);
```

This function is generalized to an array of dimensions $m \times d$, and leading dimension `ldS` (see Section 5.4.1).

```
void ran_export_sample(FILE *f, int m, int d, double *S, int ldS);
```

Chapter 9

Quasi-Monte Carlo techniques

A quasi-Monte Carlo (QMC) method is an approximation of the intregal of f over the d-dimensional hypercube defined as

$$\int_{[0,1)^d} f(x)dx \approx \frac{1}{n}\sum_{i=1}^{n} f(x_i), \tag{9.1}$$

for carefully chosen points $x_1, \ldots, x_n$ in the unit hypercube $[0,1)^d$. In a stochastic framework, we can rewrite (9.1) as

$$E[f(U_1, \ldots, U_d)] \approx \frac{1}{n}\sum_{i=1}^{n} f(x_i).$$

In contrast with MC method, classical QMC approaches are based on carefully designed deterministic point sets. In particular, these point sets can be the first n points of an infinite sequence, where the points are enumerated in a specific order. The construction of the points x_i also depends explicitely on the problem dimension; in particular, the vectors x_i in $[0,1)^s$ cannot be constructed by taking sets of d consecutive elements from a scalar sequence. Moreover, without an upper bound on s, QMC methods are inapplicable.

We will enumerate in this chapter the points of a point set $P_n \in [0,1)^s$ from 0 to $n-1$, for consistency with the usual notation in the QMC litterature. Numerous references exist about QMC techniques. This chapter is partly inspired from Glasserman [4], Chapter 8, while we strongly limit the discussion as it would otherwise leads us outside the scope of this documentation.

9.1 Low discrepancy sequences

As stated before, we want to generate a set of points x_1, x_2,..., x_N covering the unit cube $U^s = [0,1)^S$ as uniformly as possible. This first requires to formalize the notion of uniformity. Given a collection of (Lebesgue) measurable subsets of $[0,1)^d$, the discrepancy of the point set $\{x_1, \ldots, x_n\}$ relative to $\mathcal{A}$ is

$$D(x_1, x_2, \ldots, x_n; \mathcal{A}) = \sup_{A \in \mathcal{A}} \left| \frac{\#\{x_i \in A\}}{n} - \mathrm{vol}(A) \right|,$$

where $\#\{x_i \in A\}$ denotes the numbers of x_i contained in A and $\mathrm{vol}(A)$ represents the volume (measure) of A.

Taking $\mathcal{A}$ to be the collection of all rectangles in $[0,1)^d$ of the form

$$\prod_{j=1}^{d} [u_j, v_h), \quad 0 \le u_j < v_j \le 1,$$

we obtain the ordinary (or extreme)discrepancy $D(x_1, \ldots, x_n)$. Restricting $\mathcal{A}$ to rectangles of the form

$$\prod_{j=1}^{d} [0, u_j),$$

defines the star discrepancy $D^*(x_1, \ldots, x_n)$. From Niederreiter [14], Proposition 2.4, we have

$$D^*(x_1, \ldots, x_n) \le D(x_1, \ldots, x_n) \le 2^d D^*(x_1, \ldots, x_n),$$

so for fixed d, the two quantities have the same order of magnitude.

In dimension $d = 1$, Niederreiter also shows that

$$D^*(x_1, \ldots, x_n) \ge \frac{1}{2n}, \quad D(x_1, \ldots, x_n) \ge \frac{1}{n},$$

and that, in both cases, the minimum is attained by

$$x_i = \frac{2i-1}{2n}, \ i = 1, \ldots, n.$$

This corresponds to the midpoint rule integration on the unit interval.

If we now consider a sequence x_1, x_2,... of points in $[0,1)$, it can be shown (Niederreiter [14], p. 24) that

$$D(x_1, \ldots, x_n) \ge D^*(x_1, \ldots, x_n) \ge \frac{c \log n}{n},$$

for infinitely many n, with c being a constant.

In dimensions $d \geq 2$, it is widely believed (Niederreiter [14], p. 32) that any point set x_1, x_2,..., x_n satisfies

$$D^*(x_1, \ldots, x_n) \geq \frac{c_d(\log n)^{d-1}}{n},$$

and the first n elements of any sequence x_1, x_2,..., satisfy

$$D^*(x_1, \ldots, x_n) \geq \frac{c'_d(\log n)^d}{n},$$

for constants c_d, c'_d depending only on dimension d. It is therefore customary to use the term "low-discrepancy" for methods that achieve a start discrepancy of $O((\log n)^d/n)$. The logarithmic term can be absorbed into any power of n, allowing the looser bound $O(1/n^{1-\epsilon})$, for all $\epsilon > 0$.

9.1.1 Van der Corput sequences

While the library does not offer any implementation of Van der Corput sequences, it is interesting to review them in order to illustrate some general principles.

Denote a base an integer $b \geq 2$. Every positive integer k has a unique representation (called its base-b or b-ary expandion) as a linear combination of nonnegative powers of b with coefficients in $\{0, 1, \ldots, b-1\}$. We can write this as

$$k = \sum_{j=0}^{\infty} a_j(k) b^j,$$

with all but finitely many of the coefficient $a_j(k)$ equal to zero. The radical inverse function ψ_b maps each k to a point in $[0, 1)$ by flipping the coefficients of k about the base-b "decimal" point to get the base-b fraction $.a_0a_1a_2\ldots$. More precisely,

$$\psi_b(k) = \sum_{j=0}^{\infty} \frac{a_j(k)}{b^{j+1}}.$$

The base-b Van der Corput sequence is the sequence $0 = \psi_b(0)$, $\psi_b(1)$, $\psi_b(2)$,... Theorem 3.6 of Niederreiter [14] shows that all Van der Corput sequences are low-discrepancy sequences.

9.2 Digital nets

$$
\begin{aligned}
i &= \sum_{\ell=0}^{k-1} a_{i,\ell} b^{\ell}, \\
\begin{pmatrix} u_{i,j,1} \\ u_{i,j,2} \\ \vdots \end{pmatrix} &= C_j \begin{pmatrix} a_{i,0} \\ a_{i,1} \\ \vdots \\ a_{i,k-1} \end{pmatrix} \mod b, \\
u_{i,j} &= \sum_{\ell=1}^{\infty} u_{i,j,\ell} b^{-\ell}, \\
\boldsymbol{u} &= (u_{i,1}, \ldots, u_{i,t}).
\end{aligned}
$$

```
void moca_XXX_next(XXX, double *x);
```

, where `XXX` is `faure`, `halton` or `sobol`.

9.2.1 Halton sequences

The Halton sequence is a d-dimensional generalization of the Van Der Corput sequence. Let $(p_1, \ldots, p_d)$ be the d first numbers, then x_i is defined by:

$$x_i = (\phi_{p_1}(i), ..., \phi_{p_1}(I)),$$

where $\phi_{p_j}(i)$ is the Van der Corput sequence in base p_j. The Halton sequence satisfies:

$$D_i^*(x) \leq \frac{1}{i} \prod_{j=1}^{d} \frac{p_j \log(p_j i)}{\log p_j} = O\left(\frac{\log^d(i)}{i}\right),$$

with a constant $C_d = \prod_{k=1}^{d} \frac{p_k - 1}{2 \log p_k}$. This constant grows to infinity super-exponentially with dimension.

9.2.2 Scrambled Halton sequences

Even though standard Halton sequences perform very well in low dimensions, correlation problems have been noted between sequences generated from higher primes. For example if we started with the primes 17 and 19, the first 17 pairs of points would have perfect linear correlation. To avoid this, it is common to drop the first 20 entries, or some other predetermined number

depending on the primes chosen. In order to deal with this problem, various other methods have been proposed; one of the most prominent solutions is the scrambled Halton sequence, which uses permutations of the coefficients used in the construction of the standard sequence.

9.2.3 Sobol sequences

The Sobol sequence is a d-dimensional sequence in base 2 and it is a (t, d)-sequence. It is one of the most used sequences for Quasi-Monte Carlo simulation. It was first developped by Sobol and it has been proved to have some additional uniformity property under some initialization conditions. Its construction is based on primitive polynomials over $\mathcal{F}_2$ and XOR operations.

The generator matrices C_j are upper triangular binary matrices with 1's on the diagonal:

$$C_j = \begin{pmatrix} 1 & v_{j,1,2} & \dots & v_{j,1,c} & \dots \\ 0 & 1 & \dots & v_{j,2,c} & \dots \\ \vdots & 0 & \ddots & \vdots & \ddots \\ \vdots & \vdots & \dots & 1 & \dots \end{pmatrix}.$$

To determine the integers $m_{j,c}$, for each j, we first select a primitive polynomial over $\mathcal{F}_2$, say

$$f_j(z) = z^{d_j} + a_{j,1} z^{d_j - 1} + \ldots + a_{j,d_j},$$

of degree d_j, and choose the first d_j integers $m_{j,1}, \ldots, m_{j,d_j}$.

9.2.4 Faure sequences

9.3 Other techniques

9.3.1 Modified Latin Hypercube

9.4 Randomized quasi-Monte Carlo

The construction of a Randomized quasi-Monte Carlo (RQMC) point set starts with a QMC point set $P_n = \{u_0, \ldots, u_{n-1}\}$, that cover the unit hypercube $[0, 1)^s$ in a very uniform way, and randomizes P_n so that after the randomization:

(R.1) it retains its high uniformity when taken as a set and

(R.2) each individual point has the uniform distribution over $[0,1)^s$.

Let $\tilde{P}_n = \{U_0, \ldots, U_{n-1}\}$ denote the randomized points. The estimator of $\mu = E[f(U)]$ based on one copy of the randomization is

$$X_{\text{rqmc}} = Q_n = \frac{1}{n}\sum_{i=0}^{n-1} f(U_i).$$

This randomization is repeated m times, independently, for some positive integer m, with the same P_n. Assume that any points taken from different randomizations are pairwise independent, in the sense that if $U_i^{(j)}$ is the i^{th} point from the j^{th} randomization, then

(R.3) for every i_1, i_2 and $j_1 \neq j_2$, $U_{i_1}^{(j_1)}$ and $U_{i_2}^{(j_2)}$ are independent.

9.4.1 Random shifts

A simple randomization that satisfies Conditions (R1) to (R3)with a small amount of change in the point set, for an arbitrary point set Pn, are the random shift modulo 1 and the random digital shift. They are defined as follows. For the random shift modulo 1, we simply generate a single point U uniformly over [0, 1)t and add it to each point of Pn, coordinate-wise, modulo 1. In the case of a lattice rule, the lattice structure of the points is preserved by the

9.5 Transformation to other distributions

Description	Symbol	#parameters
`constant`	`CONSTANT`	1
`fixed`	`FIXED`	1
`bernouilli`	`BERNOUILLI`	1
`uniform`	`UNIFORM`	2
`normal`	`NORMAL`	2
`lognormal`	`LOGNORMAL`	2
`triangular`	`TRIANGULAR`	2
`truncated_normal`	`TRUNCATED_NORMAL`	4
`truncated_normal_b`	`TRUNCATED_NORMAL_B`	2
errc	ERRC	

Table 9.1: Supported distributions

Appendix A

Legal aspects

A.1 Licenses

All of the code, as well as the library seen as a single unit, is placed under the OSL. v2.1, except the MT19937 generator, which is covered by the BSD license (original implementation is available at `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html`). We reproduce both of these licenses below. File cblas.h is in public domain.

A.1.1 Open Software License v 2.1

This Open Software License (the "License") applies to any original work of authorship (the "Original Work") whose owner (the "Licensor") has placed the following notice immediately following the copyright notice for the Original Work:

Licensed under the Open Software License version 2.1

1) Grant of Copyright License. Licensor hereby grants You a world-wide, royalty-free, non-exclusive, perpetual, sublicenseable license to do the following:

   a) to reproduce the Original Work in copies;

   b) to prepare derivative works ("Derivative Works") based upon the Original Work;

   c) to distribute copies of the Original Work and Derivative Works to the public, with the proviso that copies of Original Work or Derivative Works that You distribute shall be licensed under the Open Software License;

   d) to perform the Original Work publicly; and

   e) to display the Original Work publicly.

2) Grant of Patent License. Licensor hereby grants You a world-wide, royalty-free, non-exclusive, perpetual, sublicenseable license, under patent claims owned or controlled by the Licensor that are embodied in the Original Work as furnished by the Licensor, to make, use, sell and offer for sale the Original Work and Derivative Works.

3) Grant of Source Code License. The term "Source Code" means the preferred form of the Original Work for making modifications to it and all available documentation describing how to modify the Original Work. Licensor hereby agrees to provide a machine-readable copy of the Source Code of the Original Work along with each copy of the Original Work that Licensor distributes. Licensor reserves the right to satisfy this obligation by placing a machine-readable copy of the Source Code in an information repository reasonably calculated to permit inexpensive and convenient access by You for as long as Licensor continues to distribute the Original Work, and by publishing the address of that information repository in a notice immediately following the copyright notice that applies to the Original Work.

4) Exclusions From License Grant. Neither the names of Licensor, nor the names of any contributors to the Original Work, nor any of their trademarks or service marks, may be used to endorse or promote products derived from this Original Work without express prior written permission of the Licensor. Nothing in this License shall be deemed to grant any rights to trademarks, copyrights, patents, trade secrets or any other intellectual property of Licensor except as expressly stated herein. No patent license is granted to make, use, sell or offer to sell embodiments of any patent claims other than the licensed claims defined in Section 2. No right is granted to the trademarks of Licensor even if such marks are included in the Original Work. Nothing in this License shall be interpreted to prohibit Licensor from licensing under different terms from this License any Original Work that Licensor otherwise would have a right to license.

5) External Deployment. The term "External Deployment" means the use or distribution of the Original Work or Derivative Works in any way such that the Original Work or Derivative Works may be used by anyone other than You, whether the Original Work or Derivative Works are distributed to those persons or made available as an application intended for use over a computer network. As an express condition for the grants of license hereunder, You agree that any External Deployment by You of a Derivative Work shall be deemed a distribution and shall be licensed to all under the terms of this License, as prescribed in section 1(c) herein.

6) Attribution Rights. You must retain, in the Source Code of any Derivative Works that You create, all copyright, patent or trademark notices from the Source Code of the Original Work, as well as any notices of licensing and any descriptive text identified therein as an "Attribution Notice." You must cause the Source Code for any Derivative Works that You create to carry a prominent

Attribution Notice reasonably calculated to inform recipients that You have modified the Original Work.

7) Warranty of Provenance and Disclaimer of Warranty. Licensor warrants that the copyright in and to the Original Work and the patent rights granted herein by Licensor are owned by the Licensor or are sublicensed to You under the terms of this License with the permission of the contributor(s) of those copyrights and patent rights. Except as expressly stated in the immediately proceeding sentence, the Original Work is provided under this License on an "AS IS" BASIS and WITHOUT WARRANTY, either express or implied, including, without limitation, the warranties of NON-INFRINGEMENT, MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY OF THE ORIGINAL WORK IS WITH YOU. This DISCLAIMER OF WARRANTY constitutes an essential part of this License. No license to Original Work is granted hereunder except under this disclaimer.

8) Limitation of Liability. Under no circumstances and under no legal theory, whether in tort (including negligence), contract, or otherwise, shall the Licensor be liable to any person for any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or the use of the Original Work including, without limitation, damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses. This limitation of liability shall not apply to liability for death or personal injury resulting from Licensor's negligence to the extent applicable law prohibits such limitation. Some jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, so this exclusion and limitation may not apply to You.

9) Acceptance and Termination. If You distribute copies of the Original Work or a Derivative Work, You must make a reasonable effort under the circumstances to obtain the express assent of recipients to the terms of this License. Nothing else but this License (or another written agreement between Licensor and You) grants You permission to create Derivative Works based upon the Original Work or to exercise any of the rights granted in Section 1 herein, and any attempt to do so except under the terms of this License (or another written agreement between Licensor and You) is expressly prohibited by U.S. copyright law, the equivalent laws of other countries, and by international treaty. Therefore, by exercising any of the rights granted to You in Section 1 herein, You indicate Your acceptance of this License and all of its terms and conditions. This License shall terminate immediately and you may no longer exercise any of the rights granted to You by this License upon Your failure to honor the proviso in Section 1(c) herein.

10) Termination for Patent Action. This License shall terminate automatically and You may no longer exercise any of the rights granted to You by this License as of the date You commence an action, including a cross-claim or counterclaim, against Licensor or any licensee alleging that the Original Work infringes

a patent. This termination provision shall not apply for an action alleging patent infringement by combinations of the Original Work with other software or hardware.

11) Jurisdiction, Venue and Governing Law. Any action or suit relating to this License may be brought only in the courts of a jurisdiction wherein the Licensor resides or in which Licensor conducts its primary business, and under the laws of that jurisdiction excluding its conflict-of-law provisions. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any use of the Original Work outside the scope of this License or after its termination shall be subject to the requirements and penalties of the U.S. Copyright Act, 17 U.S.C. Â§101 et seq., the equivalent laws of other countries, and international treaty. This section shall survive the termination of this License.

12) Attorneys Fees. In any action to enforce the terms of this License or seeking damages relating thereto, the prevailing party shall be entitled to recover its costs and expenses, including, without limitation, reasonable attorneys' fees and costs incurred in connection with such action, including any appeal of such action. This section shall survive the termination of this License.

13) Miscellaneous. This License represents the complete agreement concerning the subject matter hereof. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable.

14) Definition of "You" in This License. "You" throughout this License, whether in upper or lower case, means an individual or a legal entity exercising rights under, and complying with all of the terms of, this License. For legal entities, "You" includes any entity that controls, is controlled by, or is under common control with you. For purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

15) Right to Use. You may use the Original Work in all ways not otherwise restricted or conditioned by this License or by law, and Licensor promises not to interfere with or be responsible for such uses by You.

A.1.2 BSD License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

Bibliography

[1] A.O.L. Atkin and Daniel J. Bernstein. Prime sieves using binary quadratic forms. *Mathematics of Computation*, 73:1023–1030, 2004.

[2] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, 1986.

[3] Merra Evans, Nicholas Hastings, and Brian Peacock. *Statistical Distributions*. John Wiley & Sons, second edition, 1993.

[4] Paul Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, New York, NY, USA, 2004.

[5] Leslie Hogben, editor. *Handbook of Linear Algebra*. CRC Press, Boca Raton, FL, USA, 2006.

[6] T.E. Hull and A. R. Dobell. Random number generators. *SIAM Review*, 4(230–254), 1962.

[7] Pierre L'Ecuyer. Uniform random numbers generators: a review. In S. Andradóttir and K. J. Healy and, editors, *Proceedings of the 29th conference on Winter simulation*, pages 127–134, New York, NY, USA, 1997. ACM Press.

[8] Pierre L'Ecuyer and Richard Simard. Testu01: A c library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4), 2007.

[9] Derrick H. Lehmer. Mathematical methods in large-scale computing units. In *Proceedings of the Second Symposium on Large Scale Digital Computing Machinery*, pages 141–146, Cambridge, United Kingdom, 1951. Harvard University Press.

[10] Luc Levroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, NY, USA, 1986.

[11] George Marsaglia. Evaluating the normal distribution. *Journal of Statistical Software*, 11(5), 2004.

[12] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[13] David R. Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997.

[14] Harald Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. SIAM, Philadelphia, Pa, USA, 1992.

[15] J. M. Miller P. A. W. Lewis, A. S. Goodman. A pseudo-random number generator for the system/360. *IBM Systems Journal*, 8(2):136–146, 1969.

[16] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1888.

[17] L. A. Piegl and W. Tiller. *The NURBS Book*. Springer-Verlag, New York, NY, USA, second edition, 1996.

[18] N.G. Ushakov. *Encyclopaedia of Mathematics*, chapter Truncated distribution. Springer-Verlag, 2001.

[19] Michael J. Wichura. Algorithm as 241: The percentage points of the normal distribution. *Applied Statistics*, 37:477–484, 1988.

Index