Last update: July 20, 2020

Mylib-C Reference Manual

Some Basic Utilities in C

Pierre L'Ecuyer and Richard Simard

Département d'Informatique et de Recherche opérationnelle
Université de Montréal

This document describes a set of basic utility functions implemented in the C language around the years 1995–2000, to be used in the software developed in the *Stochastic Simulation Laboratory* under the supervision of Pierre L'Ecuyer. Many of these tools were originally implemented earlier in the Modula-2 language, and used in our old software to analyze and test random number generators (RNGs) back in the late 1980's. They were translated to C when our old RNG software was converted from Modula-2 to C in the late 1990's. Some of these functions may have counterparts in recent standard C libraries, but we keep them to avoid rewriting our code. They are used in particular in the TestU01 and F2LinearGen libraries.

This reference manual is written in LaTeX and the `.tex` source files are also used to produce the `.h` files of the library, to ensure that they agree with the documentation.

In addition to the functions described here, unit-test functions are provided in the separate folder `mylib-unit-tests`. They permit one to check each function for correctness.

Contributors: Pierre L'Ecuyer, Richard Simard, François Panneton, Francis Picard, Jean-Sébastien Sénécal, Simon-Olivier Laperrière.

Contents

gdef

A few macros and platform-dependent options are defined here. These options are used by other modules to decide when platform-dependent functions must be commented out in the code, or not. Most of these options are set to their true values by the program *configure* in the installation process. The user may choose to set some of them manually. This module also contains functions to test near equality between numbers represented in `double`, and a function that prints the current host name.

Global macros

```
#define FALSE 0
#define TRUE 1
```

```
typedef int lebool;
```

Defines the boolean type `lebool`, whose only possible values are `TRUE` and `FALSE`.

```
#define DIR_SEPARATOR "/"
```

Used to separate directories in the pathname of a file. It is `"/"` on Unix-Linux and most other platforms. It may have to be set to `"\\"` on some platforms.

```
#undef USE_GMP
```

Define this macro if the GNU multi-precision package GMP is available. GMP is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. See the Free Software Foundation web site at `http://www.gnu.org/software/gmp/manual`. A few random number generators in library TestU01 use arbitrary large integers, and they have been implemented with GMP functions. If one wants to use GMP, the GMP header file (`gmp.h`) must be in the search path of the C compiler for included files, and the GMP library must be linked to create executable programs.

Host machine

```
void gdef_GetHostName (char machine[], int n);
```

Returns in `machine` the host name. Will copy at most n characters, so the array `machine[]` should have a size $\geq n$. This is useful, for example, to get the name of the machine on which a program is running.

```
void gdef_WriteHostName (void);
```

Prints the name of the machine on which a program is running. This should work on any Unix or Linux machine.

util

This module offers macros and functions used for testing, print error messages, allocate dynamic memory, and read/write Boolean variables. Some of these "functions" are implemented as macros, for better speed. The module also contains safe prototype functions to open and close files.

```
#include "gdef.h"
#include <stdio.h>
#include <stdlib.h>
```

Macros

`util_Error (s);`

Print the string `s`, then stop the program.

`util_Assert (Assertion, s);`

If `lebool Assertion` is `FALSE` $(= 0)$, this macro prints the string `s` and stops the program.

`util_Warning (Condition, s);`

If `lebool Condition` is `TRUE` $(\neq 0)$, this macro prints the string `s`.

`util_Max (x, y);`

Returns the largest of the two numbers `x`, `y`.

`util_Min (x, y);`

Returns the smallest of the two numbers `x`, `y`.

Prototypes

```
lebool util_nearEqual (double x, double y, double eps);
```

Tests if the two floating-point numbers x and y are nearly equal. Returns `TRUE` if $|x - y| <$ `eps`, and to `FALSE` otherwise.

```
lebool util_nearEqualRel (double x, double y, double eps);
```

Tests if the two floating-point numbers x and y are nearly equal, relatively to the value of $|y|$. Returns `TRUE` if $|x - y| <$ `eps`$|y|$, and to `FALSE` otherwise. This is often used if the result x of a calculation is supposed to equal a known value y, and we want to test if the calculation appears correct. One might then take `eps` somewhere around 10^{-12} to 10^{-15}, for example.

```
lebool util_nearEqualDefault (double x, double y);
```

Tests if the two floating-point numbers x and y are nearly equal, relatively to the value of $|y|$ by calling `util_nearEqualRel` and by using a default value of 1.0E-12 for the bound `eps` on the relative error.

```
FILE * util_Fopen (const char *name, const char *mode);
```

Calls `fopen` (from `stdio.h`) with the same arguments, but checks for errors. Opens or creates file with name `name` in mode `mode`. Returns a pointer to FILE that is associated with the stream. If `name` cannot be accessed, the program stops.

```
int util_Fclose (FILE *stream);
```

Calls `fclose` (from `stdio.h`) with the same arguments, but checks for errors. Closes the file associated with `stream`. If the file is successfully closed, `0` is returned. If an error occurs or the file was already closed, `EOF` is returned.

```
int util_GetLine (FILE *file, char *Line, char c);
```

Reads a line of data from `file`. Blank lines and comments are ignored. A comment is any line whose first non-whitespace character is `c`. If the character `c` appears anywhere on a line that is not a comment, then `c` and the rest of the line are ignored too. The function returns -1 if end-of-file or an error is encountered, otherwise it returns 0.

```
void util_ReadLn (FILE *f);
```

Skips the remainder of the current line of FILE `f` and moves the current position of the pointer to FILE `f` to the beginning of the next line.

```
void util_ReadBool (char S[], lebool *x);
```

Reads a `lebool` value from string `S` and returns it in x. The possible values are `TRUE` and `FALSE`.

```
void util_WriteBool (lebool x, int d);
```

Writes the value of x in a field of width d. If $d < 0$, x is left-justified, otherwise right-justified.

```
void * util_Malloc (size_t size);
```

Calls `malloc` (from `stdlib.h`) with same arguments, but checks for errors. Allocates memory large enough to hold an object of size `size`. A successful call returns the base address of the allocated space, otherwise the programs stops. The standard type `size_t` is defined in `stdio.h`.

```
void * util_Calloc (size_t dim, size_t size);
```

Calls `calloc` (from `stdlib.h`) with same arguments, but checks for errors. Allocates memory large enough to hold an array of `dim` objects each of size `size`. A successful call returns the base address of the allocated space, otherwise the programs stops. The standard type `size_t` is defined in `stdio.h`.

`void * util_Realloc (void *ptr, size_t size);`

Calls `realloc` (from `stdlib.h`) with same arguments, but checks for errors. Takes a pointer to a memory region previously allocated and referenced by `ptr`, then changes its size to `size` while preserving its content. A successful call returns the base address of the resized (or new) space, otherwise the programs stops. The standard type `size_t` is defined in `stdio.h`.

`void * util_Free (void *p);`

Calls `free (p)` (from `stdlib.h`) to free memory allocated by `util_Malloc`, `util_Calloc` or `util_Realloc`. Always returns a `NULL` pointer.

num

This module provides some useful constants and basic tools to manipulate numbers represented in different forms.

```
#include "gdef.h"
```

Constants

```
#define num_Pi      3.14159265358979323846
```

The number π.

```
#define num_ebase   2.7182818284590452354
```

The Euler number e.

```
#define num_Rac2    1.41421356237309504880
```

$\sqrt{2}$, the square root of 2.

```
#define num_1Rac2   0.70710678118654752440
```

$1/\sqrt{2}$.

```
#define num_Ln2     0.69314718055994530941
```

$\ln(2)$, the natural logarithm of 2.

```
#define num_1Ln2    1.44269504088896340737
```

$1/\ln(2)$.

```
#define num_MaxIntDouble   9007199254740992.0
```

Largest integer $n_0 = 2^{53}$ such that all integers $n \le n_0$ are represented exactly as a `double`.

Precomputed powers

```
#define num_MaxTwoExp   64
```

Powers of 2 up to `num_MaxTwoExp` are stored exactly in the array `num_TwoExp`.

```
#define num_TwoExp(x) (uint64_t(1)<<x)
```

```
extern double num_TwoExp[];
```

Contains precomputed powers of 2. One has `num_TwoExp[i]` $= 2^i$ for $0 \le i \le$ `num_MaxTwoExp`. Instead of storing the numbers, this could be replaced by a macro, as in the line above. Simple and no need to store a table. To get the exact value in `double`, just use `(double)num_TwoExp(x)`. One limitation: Does not work for $x \ge 64$.

```
#define num_MAXTENNEGPOW   16
```

Negative powers of 10 up to `num_MAXTENNEGPOW` are stored in the array `num_TENNEGPOW`.

```
extern double num_TENNEGPOW[];
```

Contains the precomputed negative powers of 10. One has `TENNEGPOW[j]`$= 10^{-j}$, for $j = 0, \ldots,$ `num_MAXTENNEGPOW`.

Prototypes

```
#define num_Log2(x) (num_1Ln2 * log(x))
```

Gives the logarithm of x in base 2.

```
int num_IntMin (int x, int y);
```

Returns the smallest integer value between `x` and `y`. This function are the next three ones are deprecated. Use `util_Min` instead.

```
int num_IntMax (int x, int y);
```

Returns the largest integer value between `x` and `y`.

```
double num_DoubleMin (double x, double y);
```

Returns the smallest double value between `x` and `y`.

```
double num_DoubleMax (double x, double y);
```

Returns the largest double value between `x` and `y`.

```
int64_t num_Round64 (double x);
```

Rounds x to the nearest (`int64_t`) integer and returns it.

```
double num_RoundD (double x);
```

Rounds x to the nearest (`double`) integer and returns it.

```
int num_IsNumber (char S[]);
```

Returns `1` if the string `S` begins with a number (with the possibility of spaces and a $+/-$ sign before the number). For example, "$+ 2$" and "4hello" return `1`, while "$- + 2$" and "hello" return `0`.

```
void num_IntToStrBase (int64_t k, int b, char S[]);
```

Returns in `S` the string representation of `k` in base `b`.

`void num_Uint2Uchar (unsigned char output[], uint64_t input[], int L);`

Transforms the L 64-bit unsigned integers contained in `input` into $8L$ characters and puts them into `output`. The order is such that the 8 most significant bits of `input[0]` will be in `output[0]`, the 8 least significant bits of `input[0]` will be in `output[7]`, and the 8 least significant bits of `input[L-1]` will be in `output[8L-1]`. Array `output` must have at least $8L$ elements.

`void num_WriteD (double x, int i, int j, int k);`

Writes `x` to current output. Uses a total of at least `i` positions (including the sign and point when they appear), `j` digits after the decimal point and at least `k` significant digits. The number is rounded if necessary. If there is not enough space to print the number in decimal notation with at least `k` significant digits (`j` or `i` is too small), it will be printed in scientific notation with at least `k` significant digits. In that case, `i` is increased if necessary. Restriction: `j` and `k` must be strictly smaller than `i`.

`void num_WriteBits (uint64_t x, int k);`

Writes `x` in base 2 in a field of at least $\max\{64, |k|\}$ positions. If $k > 0$, the number will be right-justified, otherwise left-justified.

`int64_t num_MultMod (int64_t a, int64_t s, int64_t c, int64_t m);`

Returns $(as{+}c) \mod m$. Uses the decomposition technique of [5] to avoid overflow. Assumptions: $\max(a, s, c) < m < 2^{63}$.

`int64_t num_MultModDirect (int64_t a, int64_t s, int64_t c, int64_t m);`

Returns $(as + c) \mod m$. Uses direct multiplication and addition. Assumptions: $m < 2^{63}$ and $as + c < 2^{63}$.

`double num_MultModDouble (double a, double s, double c, double m);`

Returns $(as + c) \mod m$, assuming that a, s, c, and m are all *integers* less than 2^{35} (represented exactly). Works under the assumption that all positive integers less than 2^{53} are represented exactly in floating-point (in `double`).

`int64_t num_InvEuclid (int64_t m, int64_t z);`

This function computes the inverse $z^{-1} \mod m$ by the modified Euclid algorithm (see [3, p. 325]) and returns the result. If the inverse does not exist, returns 0. This function can handle higher values than `num_InvEuclid32`.

`long num_InvEuclid32 (long m, long z);`

Same as `num_InvEuclid`, but for 32-bit integers.

`uint64_t num_InvExpon (int e, uint64_t z);`

This function computes the inverse $z^{-1} \mod 2^e$ by exponentiation and returns the result. If the inverse does not exist, returns 0. Restriction: $e \le 64$.

```
long num_InvExpon32 (int e, long z);
```

Same as `num_InvExpon`, but for 32-bit integers. Restriction: $e \leq 32$.

```
uint64_t num_gcd (uint64_t a, uint64_t b);
```

Returns the greatest common divisor gcd(`a`,`b`).

```
long num_gcd32 (long a, long b);
```

Same as `num_gcd`, but for 32-bit integers.

```
lebool num_isMersennePrime (int e);
```

Returns `TRUE` iff $2^e - 1$ is a Mersenne prime for $e \leq 110503$.

num2

This module provides procedures to compute certain numerical quantities such as factorials, combinations, Stirling numbers, Bessel functions, gamma functions, and so on. These functions are more involved than those provided by `num`.

```
#include "gdef.h"
#include <math.h>
```

Prototypes

```
double num2_Factorial (int n);
```

The factorial function. Returns the value of $n!$

```
double num2_LnFactorial (int n);
```

Returns the value of $\ln(n!)$, the natural logarithm of the factorial of n. Gives at least 16 decimal digits of precision (relative error $< 0.5 \times 10^{-15}$)

```
double num2_Combination (int n, int s);
```

Returns the value of $\binom{n}{s}$, the number of different combinations of s objects amongst n.

```
#ifdef HAVE_LGAMMA
#define num2_LnGamma lgamma
#else
   double num2_LnGamma (double x);
#endif
```

Calculates the natural logarithm of the gamma function $\Gamma(x)$ at `x`. Our `num2_LnGamma` gives 16 decimal digits of precision, but is implemented only for $x > 0$. The function `lgamma` is from the `ISO C99` standard math library.

```
double num2_Digamma (double x);
```

Returns the value of the logarithmic derivative of the Gamma function $\psi(x) = \Gamma'(x)/\Gamma(x)$.

```
#ifdef HAVE_LOG1P
#define num2_log1p log1p
#else
   double num2_log1p (double x);
#endif
```

Returns an approximation of $\texttt{log}(1 + x)$ which is accurate for small x. The function `log1p` is from the `ISO C99` standard math library.

```
void num2_CalcMatStirling (double *** M, int m, int n);
```

Calculates the Stirling numbers of the second kind and returns them in matrix M with

$$M[i,j] = \left\{ \begin{matrix} j \\ i \end{matrix} \right\} \qquad \text{for } 0 \le i \le m \text{ and } 0 \le i \le j \le n. \tag{1}$$

See [2], Section 1.2.6. Our matrix `M` is the transpose of that in [2]. This procedure allocates memory for the two-dimensional matrix M, and fills it with the values of Stirling numbers; the memory should be released later by invoking `num2_FreeMatStirling`.

```
void num2_FreeMatStirling (double *** M, int m);
```

Frees the memory space used by the Stirling matrix created by calling `num2_CalcMatStirling`. The parameter `m` must be the same as the `m` in `num2_CalcMatStirling`.

```
double num2_VolumeSphere (double p, int t);
```

Calculates the volume V of a sphere of radius 1 in t dimensions using the norm L_p, according to the formula

$$V = \frac{[2\Gamma(1+1/p)]^t}{\Gamma\left(1+t/p\right)}, \qquad p > 0,$$

where Γ is the well-known gamma function. The case of the sup norm L_∞ is obtained by choosing $p = 0$. Restrictions: $p \ge 0$ and $t \ge 1$.

```
double num2_EvalCheby (const double a[], int n, double x);
```

Evaluates a series of Chebyshev polynomials T_j, at point $x \in [-1,\ 1]$, using the method of Clenshaw [1], i.e. calculates and returns

$$y = \frac{a_0}{2} + \sum_{j=1}^{n} a_j T_j(x),$$

where $a_0, \ldots, a_n$ are given in array `a[0..n]`.

```
double num2_BesselK025 (double x);
```

Returns the value of $K_{1/4}(x)$, where K_ν is the modified Bessel's function of the second kind. The relative error on the returned value is less than 0.5×10^{-6} for $x > 10^{-300}$.

mystr

This module offers some tools for the manipulation of character strings.

```
void mystr_Delete (char S[], unsigned int index, unsigned int len);
```

Deletes `len` characters from S, starting at position `index`.

```
void mystr_Insert (char Res[], char Source[], unsigned int Pos);
```

Inserts the string `Source` into `Res`, starting at position `Pos`.

```
void mystr_ItemS (char R[], char S[], const char T[], unsigned int N);
```

Returns in R the N-th substring of S (counting from 0). Substrings are delimited by any character from the set T.

```
int mystr_Match (char Source[], char Pattern[]);
```

Returns `1` if the string `Source` matches the string `Pattern`, and `0` otherwise. The characters "?" and "*" are recognized as wild characters in the string `Pattern`.

```
void mystr_Slice (char R[], char S[], unsigned int P, unsigned int L);
```

Returns in `R` the substring in `S` beginning at position `P` and of length `L`.

```
void mystr_Subst (char Source[], char OldPattern[], char NewPattern[]);
```

Searches for the string `OldPattern` in the string `Source`, and replaces its first occurence with `NewPattern`.

```
void mystr_Position (char Substring[], char Source[], unsigned int at,
                     unsigned int * pos, int * found);
```

Searches for the string `Substring` in the string `Source`, starting at position `at`, and returns the position of its first occurence in `pos`.

addstr

The functions described here are convenient tools for constructing character strings that contain a series of numeric parameters, with their values. For example, suppose one wishes to put "LCG with m = 101, a = 12, s = 1" in the string str, where the actual numbers 101, 12, and 1 must be taken as the values of uint64_t variables m, a, and s. This can be achieved by the instructions:

```
strcpy (str, "LCG with ");
addstr_uint64 (str, " m = ", m);
addstr_uint64 (str, ", a = ", m);
addstr_uint64 (str, ", s = ", s);
```

Each function addstr_... (char *to, const char *add, ...) first appends the string add to the string to, then appends to it a character string representation of the number (or array of numbers) specified by its last parameter. In the case of an array of numbers (e.g., addstr_array_uint64), the parameter high specifies the size of the array, and the elements [0..high-1] are added to str. The ..._int64 and ..._uint64 versions are for 64-bit integers. In all cases, the string to should be large enough to accomodate what is appended.

```
#include "gdef.h"
```

Prototypes

```
void  addstr_int (char *to, const char *add, int n);

void  addstr_uint (char *to, const char *add, unsigned int n);

void  addstr_long (char *to, const char *add, long n);

void  addstr_ulong (char *to, const char *add, unsigned long n);

void  addstr_int64 (char *to, const char *add, int64_t n);

void  addstr_uint64 (char *to, const char *add, uint64_t n);

void  addstr_double (char *to, const char *add, double x);

void  addstr_char (char *to, const char *add, char c);

void  addstr_bool (char *to, const char *add, int b);
```

```
void  addstr_array_int (char *to, const char *add, int high, int val[]);

void  addstr_array_uint (char *to, const char *add, int high,
                         unsigned int val[]);

void  addstr_array_long (char *to, const char *add, int high, long val[]);

void  addstr_array_ulong (char *to, const char *add, int high,
                          unsigned long val[]);

void  addstr_array_int64 (char *to, const char *add, int high, int64_t val[]);

void  addstr_array_uint64 (char *to, const char *add, int high, uint64_t val[]);

void  addstr_array_double (char *to, const char *add, int high, double val[]);

void  addstr_array_char (char *to, const char *add, int high, char val[]);
```

tables

This module provides an implementation of variable-sized arrays (matrices), and procedures to manipulate them. The advantage is that the size of the array needs not be known at compile time; it can be specified only during the program execution. There are also procedures to sort arrays, to print arrays in different formats, and a few tools for hashing tables. The functions `tables_CreateMatrix...` and `tables_DeleteMatrix...` manage memory allocation for these dynamic matrices.

As an illustration, the following piece of code declares and creates a 100×500 table of floating point numbers, assigns a value to one table entry, and eventually deletes the table:

```
double ** T;
T = tables_CreateMatrix_double (100, 500);
T[3][7] = 1.234;
...
tables_DeleteMatrix_double (&T);
```

```
#include "gdef.h"
```

Printing styles

```
typedef enum {
   tables_Plain,
   tables_Mathematica,
   tables_Matlab
   } tables_StyleType;
```

Printing styles for matrices.

Functions to create, delete, sort, and print tables

```
long ** tables_CreateMatrix_long  (int m, int n);
unsigned long ** tables_CreateMatrix_ulong (int m, int n);
int64_t ** tables_CreateMatrix_int64  (int m, int n);
uint64_t ** tables_CreateMatrix_uint64 (int m, int n);
double ** tables_CreateMatrix_double  (int m, int n);
```

Allocates contiguous memory for a dynamic matrix of `m` rows and `n` columns. Returns the base address of the allocated space.

```
void tables_DeleteMatrix_long  (long *** mat);
void tables_DeleteMatrix_ulong (unsigned long *** mat);
void tables_DeleteMatrix_int64  (int64_t *** mat);
void tables_DeleteMatrix_uint64 (uint64_t *** mat);
void tables_DeleteMatrix_double  (double *** mat);
```

Releases the memory used by the matrix `mat` (see `tables_CreateMatrix`) passed by reference;

i.e., using the & symbol. Then, mat is set to NULL.

```
void tables_CopyTab_long (long mat1[], long mat2[], int n1, int n2);
void tables_CopyTab_int64 (int64_t mat1[], int64_t mat2[], int n1, int n2);
void tables_CopyTab_uint64 (uint64_t mat1[], uint64_t mat2[], int n1, int n2);
void tables_CopyTab_double (double mat1[], double mat2[], int n1, int n2);
```

Copies mat2[n1..n2] in mat1[n1..n2].

```
void tables_QuickSort_long (long mat[], int n1, int n2);
void tables_QuickSort_int64 (int64_t mat[], int n1, int n2);
void tables_QuickSort_uint64 (uint64_t mat[], int n1, int n2);
void tables_QuickSort_double (double mat[], int n1, int n2);
```

Sort the tables mat[n1..n2] in increasing order.

```
void tables_WriteTab_long (long mat[], int n1, int n2, int k, int p,
                           char desc[]);
void tables_WriteTab_int64 (int64_t mat[], int n1, int n2, int k, int p,
                            char desc[]);
void tables_WriteTab_uint64 (uint64_t mat[], int n1, int n2, int k, int p,
                             char desc[]);
```

Write the elements n1 to n2 of table mat, k per line, p positions per element. If $k = 1$, the index will also be printed. desc contains a description of the table.

```
void tables_WriteTab_double (double mat[], int n1, int n2, int k,
                             int p1, int p2, int p3, char desc[]);
```

Writes the elements n1 to n2 of table mat, k per line, with at least p1 positions per element, p2 digits after the decimal point, and at least p3 significant digits. If $k = 1$, the index will also be printed. desc contains a description of the table.

```
void tables_WriteSubmatrix_double (double** mat, int i1, int i2, int j1, int j2,
                             int w, int p, tables_StyleType style, char name[]);
```

Writes the submatrix with lines $i1 \le i \le i2$ and columns $j1 \le j \le j2$ of the matrix mat with format style. The elements are printed in w positions with a precision of p digits. name is an identifier for the submatrix.

For Matlab, the file containing the printed submatrix should have the extension .m. For example, if it is named poil.m, i t will be accessed by the simple call poil in Matlab. For Mathematica, if the file is named poil, it will be read using << poil;.

```
void tables_WriteSubmatrix_long (long** mat, int i1, int i2, int j1, int j2,
                                 int w, tables_StyleType style, char name[]);
void tables_WriteSubmatrix_int64 (int64_t** mat, int i1, int i2, int j1, int j2,
                                  int w, tables_StyleType style, char name[]);
void tables_WriteSubmatrix_uint64 (uint64_t** mat, int i1, int i2, int j1,
                           int j2, int w, tables_StyleType style, char name[]);
```

Similar to tables_WriteMatrix_double.

```
int64_t tables_HashPrime (int64_t n, double load);
```

Returns a prime number p to be used as the size (the number of elements) of a hashing table. p will be such that the load factor n/p do not exceed `load`. If `load` is small, an important part of the table will be unused; that will accelerate searches and insertions. This function uses a small sequence of prime numbers; the real load factor may be significantly smaller than `load` because only a limited number of prime numbers are in the table. In case of failure, returns -1.

```
long tables_HashPrime32 (long n, double load);
```

Same as `tables_HashPrime`, but with 32-bit integers.

bitset

This module defines sets of bits and useful operations for such sets. Some of these operations are implemented as macros. Each bit set is stored as a 64-bit unsigned integer, whose bits are simply interpreted as indicators of which elements belong to the set. The bits are numbered from 0 to 63, and *bit 0 is the least significant bit* of this word, which means that the bits are numbered from right to left. If bit j is 1, then element j is a member of the set, otherwise it is not. Other operations not described here can also be applied directly to the `uint64_t` variable.

Constants

```
extern const uint64_t bitset_ONE;
extern const uint64_t bitset_ALLONES;
```

The two constants 1 and $2^{64} - 1$ (a bitset with all ones).

Types

```
typedef uint64_t bitset_BitSet;
```

A set of bits. The bits are numbered starting from 0 for the least significant bit.

Macros

```
#define bitset_Mask1(b) (bitset_ONE << b)
```

Gives a bit set with only bit b set to 1 and all other bits set to 0.

```
#define bitset_MaskLow(b) (bitset_ALLONES >> (64-b))
```

Gives a bit set with all *the lowest (rightmost) b bits* set to 1 and all other bits set to 0.

```
#define bitset_SetBit(s, b) ((s) |= (bitset_Mask1(b)));
```

Sets bit `b` in set `s` to 1.

```
#define bitset_ClearBit(s, b) ((s) &=  (bitset_Mask1(b)))
```

Sets bit `b` in set `s` to 0.

```
#define bitset_FlipBit(s, b) ((s) ^= (bitset_Mask1(b)))
```

Flips bit `b` in set `s`; thus, $0 \to 1$ and $1 \to 0$.

```
#define bitset_GetBit(s, b)  ((s) & (bitset_Mask1(b)) ? 1 : 0)
```

Returns the value of bit b in set s (0 or 1).

```
#define bitset_RotateLeft(s, r)  ((s << r) | (s >> (64 - r)))
```

Rotates the bit set s by r positions to the left.

```
#define bitset_RotateRight(s, r)  ((s >> r) | (s << (64 - r)))
```

Rotates the bit set s by r positions to the right.

```
#define bitset_RotateLeftLocal(s, b, r)
```

Rotates the b lowest (least significant) bits of the set s by r positions to the left. Here, s is considered as a b-bit number kept in the least significant bits of s.

```
#define bitset_RotateRightLocal(s, b, r)
```

Rotates the b lowest (least significant) bits of the set s by r positions to the right. Here, s is considered as a b-bit number kept in the least significant bits of s.

Prototypes

```
bitset_BitSet bitset_ReverseOrderSimple (bitset_BitSet z, int b);
```

Reverses the order of the b least significant bits of z by looping through the bits (simple but very inefficient). Thus, if $b = 4$ and $z = \ldots 0011$, the returned value is $\ldots 1100$.

```
bitset_BitSet bitset_ReverseOrder (bitset_BitSet z, int b);
```

Reverses the order of the b least significant bits of z using a lookup table, as implemented in https://graphics.stanford.edu/~seander/bithacks.html.

```
void bitset_Write (char *desc, bitset_BitSet z, int b);
```

Prints the string desc (which may be empty), then writes the b least significant bits of z considered as an unsigned binary number. This corresponds to the b first elements of z.

bitvector

This module offers facilities to store and manipulate bit vectors of arbitrary length. The bit vectors are stored in arrays of 64-bit unsigned integers, and their length is always rounded up to the nearest multiple of 64. The bit indexation goes from left to right (which is unusual) and starts at 0. This left-to-right ordering is because these vectors are used to construct binary matrices (see `bitmatrix`), for which the elements are usually ordered from left to right. If the required length is not a multiple of 64, the unused bits are simply set to 0.

```
#include "gdef.h"

typedef struct{
   int n;
   uint64_t *vect;
} bitvector_vector;
```

This structure contains a bit vector of 64`n` bits, stored in `n` blocks of 64 bits. Storage space for the `bitvector_vector` should be allocated via `bitvector_allocate()`.

```
void bitvector_allocate (bitvector_vector *v, int b);
```

Allocates memory for a bit vector of `b` bits. There will be $n = \lceil b/64 \rceil$ blocks of 64 bits in the structure and the value of b is not saved.

```
void bitvector_free (bitvector_vector *v);
```

Releases the memory space used by `bitvector_vector` pointed by `v`.

```
void bitvector_display (bitvector_vector *v, int b);
```

Prints the first `b` bits of the bit vector `v`.

```
void bitvector_copy (bitvector_vector *v1, bitvector_vector *v2);
```

Copies the entire contents of `v2` into `v1`.

```
void bitvector_copyPart (bitvector_vector *v1, bitvector_vector *v2, int b);
```

Copies the lowest `b` bits of `v2` into `v1`.

```
void bitvector_clearVector (bitvector_vector *v);
```

Resets all the bits of `v` to zero.

```
void bitvector_setBit (bitvector_vector *v, int b);
```

Sets bit `b` of `v` to 1.

```
int bitvector_getBit (bitvector_vector *v, int b);
```

Returns the value of the `b`-th bit of `v`.

```
void bitvector_clearBit (bitvector_vector *v, int b);
```

Sets bit b of v to 0.

```
int bitvector_hammingWeight (bitvector_vector *v);
```

Returns Hamming weight of v.

```
void bitvector_canonical (bitvector_vector *v, int t);
```

Sets v to the $(\mathtt{t}+1)$-th unit vector, with a 1 in position t. That is, for t = 0, we get te first unit vector $\mathbf{e}_1 = (1, 0, 0, \dots)$.

```
void bitvector_setAllOnes (bitvector_vector *v);
```

Sets v to the bit vector $(1, 1, 1, 1, \dots, 1)$.

```
lebool bitvector_isZero (bitvector_vector *v);
```

Returns TRUE if v is the zero vector. Returns FALSE otherwise.

```
lebool bitvector_areEqual (bitvector_vector *v1, bitvector_vector *v2);
```

Returns TRUE if the bit vectors v1 and v2 are the same, and FALSE otherwise.

```
lebool bitvector_haveCommonBit (bitvector_vector *v1, bitvector_vector *v2);
```

Returns TRUE if at least one bit set to 1 in v1 is also set to 1 in v2 (they have at least one common bit). Returns FALSE otherwise.

```
void bitvector_xor (bitvector_vector *v1, bitvector_vector *v2,
                    bitvector_vector *v3);
```

Performs a bitwise exclusive-or of the contents of v2 and v3, and puts the result in v1.

```
void bitvector_xor3 (bitvector_vector *v1, bitvector_vector *v2,
                     bitvector_vector *v3, bitvector_vector *v4);
```

Performs a bitwise exclusive-or of the contents of v2, v3, and v4, and puts the result in v1.

```
void bitvector_xorSelf (bitvector_vector *v1, bitvector_vector *v2);
```

Performs a bitwise exclusive-or of the contents of v1 and v2, and puts the result in v1.

```
void bitvector_and (bitvector_vector *v1, bitvector_vector *v2,
                    bitvector_vector *v3);
```

Performs a bitwise “and” of the contents of v2 and v3, and puts the result in v1.

```
void bitvector_andSelf (bitvector_vector *v1, bitvector_vector *v2);
```

Performs a bitwise “and” of the contents of v1 and v2, and puts the result in v1.

`void bitvector_andMaskLow (bitvector_vector *v1, bitvector_vector *v2, int t);`

Applies the mask comprised of `t` ones followed by zeros to the bit vector `v2` and puts the result in `v1`.

`void bitvector_andInvMaskLow (bitvector_vector *v1, bitvector_vector *v2, int t);`

Applies the mask comprised of `t` zeros followed by ones to the bit vector `v2` and puts the result in `v1`.

`void bitvector_leftShift (bitvector_vector *v1, bitvector_vector *v2, int b);`

Performs a left shift of `v2` by `b` bits and puts the result in `v1`.

`void bitvector_rightShift (bitvector_vector *v1, bitvector_vector *v2, int b);`

Performs a right shift of `v2` by `b` bits and puts the result in `v1`.

`void bitvector_leftShiftSelf (bitvector_vector *v, int b);`

Performs a left shift of `v` by `b` bits and puts the result in `v`.

`void bitvector_leftShift1Self (bitvector_vector *v);`

Performs a left shift of `v` by one bit and puts the result in `v`.

`void bitvector_rightShiftSelf (bitvector_vector *v, int b);`

Performs a right shift of `v` by `b` bits and puts the result in `v`.

`void bitvector_flip (bitvector_vector *v);`

Flips (or toggle) the values of all the bits of `v` (exchange 0 for 1, and vice-versa).

`void bitvector_setMaskLow (bitvector_vector *v, int t);`

Fills `v` with `t` ones followed by zeros.

`void bitvector_setInvMaskLow (bitvector_vector *v, int t);`

Fills `v` with `t` zeros followed by ones.

`void bitvector_fillRandomBits (bitvector_vector *v);`

Fills the vector `v` with random bits. [1]

[1] From Pierre: This needs to be implemented! For now, use rngstreams to fill using blocks of 32 bits. I realize it would be better to use a 64-bit RNGs, however.

bitmatrix

This module offers facilities to manipulate matrices of binary vectors (from the module `bitvector`). More specifically, each matrix has `r` rows and `t` columns, and each entry is an `b`-bit binary vector. By taking `t=1`, we get an ordinary binary matrix of `r` rows and `b` columns of bits. The more general for with `t` > 1 is very convenient for the analysis of equidistribution properties of $\mathbb{F}_2$-linear random number generators [6, 7, 9].

```
#include "gdef.h"
#include "bitvector.h"

typedef struct{
  bitvector_vector **rows;
  int r;
  int b;
  int t;
} bitmatrix_matrix;
```

This structure represents a matrix of `r` rows and `t` columns, for which each entry is an `b`-bit binary vector. This gives a matrix with `r` rows of `t` $\times$ `b` bits each. The variable `rows` contains an array of `r` arrays of `t` `b`-bit vectors. Storage space for this array should be allocated via `bitmatrix_allocate()`.

```
void bitmatrix_allocate (bitmatrix_matrix* m, int r, int b, int t);
```

Allocates space for a `bitmatrix_matrix` with `r` rows and `t` columns, for which each entry is an `b`-bit binary vector. On each row, the function allocates space for `t` bit vectors of `b` bits each. To allocate a simple 128×128 binary matrix, for example, one can invoke `bitmatrix_allocate (m, 128, 128, 1)`.

```
void bitmatrix_free (bitmatrix_matrix *m);
```

Releases the space taken by the binary matrix `m`.

```
void bitmatrix_display (bitmatrix_matrix *m, int t, int l, int r);
```

Displays (print) the submatrix of `*m` defined by the first `r` rows and the first `l` bits of the first `t` bit vectors.

```
int bitmatrix_hammingWeight (bitmatrix_matrix *m, int r);
```

Returns the Hamming weight of row `r` in matrix `*m`.

```
int bitmatrix_weight (bitmatrix_matrix *m);
```

Returns the sum of non-zero entries in matrix `*m`.

```
void bitmatrix_copypart (bitmatrix_matrix *m1, bitmatrix_matrix *m2,
                         int r, int t);
```

Copies the first `t` bit vectors of the first `r` rows of `m2` into the first `t` bit vectors of the first `r` rows of `m1`.

```
void bitmatrix_copySpecial (bitmatrix_matrix *m1, bitmatrix_matrix *m2,
                            int nl, int *col, int t);
```

Copies the (t-1) bit vectors indicated by the array col, plus the first bit vector, on each of the nl first rows of the matrix m2 to the first t bit vectors on each of the first nl rows of the matrix m1.

```
void bitmatrix_transpose (bitmatrix_matrix *m1, bitmatrix_matrix *m2,
                          int r, int t, int b);
```

Transposes the t matrices of dimension r×b found in m2 and puts the result in m1.

```
void bitmatrix_exchangeRows (bitmatrix_matrix *m, int i, int j);
```

Exchanges rows i and j in the binary matrix *m.

```
void bitmatrix_xorVect (bitmatrix_matrix *m, int r, int s, int min, int max);
```

Performs a exclusive-or between the s-th and r-th rows of m, for the min-th to the (max-1)-th bit vectors only. The result is put in row r of m.

```
lebool bitmatrix_diagonalize (bitmatrix_matrix *m, int kg, int t, int l, int *gr);
```

Diagonalizes the sub matrix of m that consist of the first kg rows and the first l bits of the first t bit vectors on each row. Returns TRUE if the sub matrix is of full rank t*l. In this case, the variable pointed by gr remains unchanged. Otherwise, returns FALSE and the variable pointed by gr is changed to the value of t for which the function would have returned TRUE.

```
int bitmatrix_gaussianElimination (bitmatrix_matrix *m, int r, int l, int t);
```

Returns the rank of the submatrix of m comprised of the first r rows and the first l bits of the first t bit vectors on each row.

```
int bitmatrix_specialGaussianElimination (bitmatrix_matrix *m,
                                          int r, int l, int t, int *indices);
```

Returns the rank of the submatrix of m formed by the first r rows and the first l bits of the bit vectors indicated by the array indices.

```
int bitmatrix_completeElimination (bitmatrix_matrix *m, int r, int l, int t);
```

This function tries to form an identity matrix by elimination. If it fails, it returns FALSE. Otherwise, it returns the rank of the submatrix of m comprised of the first r rows and the first l bits of the first t bit vectors on each row.

```
lebool bitmatrix_inverse (bitmatrix_matrix *minv, bitmatrix_matrix *m);
```

Tries computing the inverse of m, returns TRUE if it succeeded, and puts the results in minv. Otherwise, returns FALSE. The sub matrices of the Matrixs pointed by m and minv that are considered are the ones composed of only the first column of the bit vectors.

```
void bitmatrix_productByVector (bitvector_vector *v1, bitmatrix_matrix *m,
                                bitvector_vector *v2);
```

Puts in `v1` the product of **m** by the vector `v2`.

```
void bitmatrix_product (bitmatrix_matrix *m1, bitmatrix_matrix *m2,
                        bitmatrix_matrix *m3);
```

Compute the matrix product of `m2` by `m3` and puts the results in `m1`. Only the first submatrices are considered (i.e. the matrices composed of the first column of bit vectors).

```
void bitmatrix_power (bitmatrix_matrix *m1, bitmatrix_matrix *m2, int64_t e);
```

Raises binary matrix `m2` to the power `e` and puts the results in `m1`. The exponent e can be negative, in which case, the inverse of `m2` will be raised to the power $|e|$.

```
void bitmatrix_powerOfTwo (bitmatrix_matrix *m1, bitmatrix_matrix *m2,
                           unsigned int e);
```

Raises binary matrix `m2` to the power $2^{\mathtt{e}}$ and puts the results in `m1`. We get $\mathtt{m1} = \mathtt{m2}^{2^e}$.

rngstream

This module provides streams of random numbers constructed as proposed in [8], using the combined multiple recursive generator `Mrg32k3a` proposed in [4] as a backbone generator. This backbone generator has period length $\rho \approx 2^{191}$. In those references, the generator was implemented in floating-point arithmetic using "`double`" variables, whereas here it is implemented using 64-bit integers, using the code proposed by S. Vigna. The seed of the RNG, and the state of a stream at any given step, are 6-dimensional vectors of 32-bit integers. The default initial seed of the first stream is $(12345, 12345, 12345, 12345, 12345, 12345)$, and the seeds of the successive streams are spaced by 2^{127} steps. Substreams are not implemented here.

```
#include "gdef.h"
#include "util.h"
#include "num.h"
#include <stdio.h>
#include <stdlib.h>

typedef struct rngstream_InfoState * rngstream;

struct rngstream_InfoState {
   int64_t Cg[6], Ig[6];
};
```

The state of a stream from the present module. The arrays `Ig` and `Cg` contain the initial state and the current state, respectively.

```
int rngstream_SetPackageSeed (int64_t seed[6]);
```

Sets the seed of the package `rngstreams` to the six integers in the vector `seed`. This will be the initial state of the first stream. If this procedure is not called, the default initial seed is $(12345, 12345, 12345, 12345, 12345, 12345)$. If it is called, the first 3 values of the seed must all be less than $m_1 = 4294967087$, and not all 0; and the last 3 values must all be less than $m_2 = 4294944443$, and not all 0. Returns 0 for valid seeds and exits otherwise.

```
rngstream rngstream_CreateStream ();
```

Creates and returns a new stream whose state variable is of type `rngstream_InfoState`. This procedure reserves space to keep the information relative to the `rngstream`, initializes its seed I_g, and sets B_g and C_g equal to I_g. The seed I_g is equal to the initial seed of the package given by `rngstream_SetPackageSeed` if this is the first stream created, otherwise it is 2^{127} steps ahead of that of the most recently created stream.

```
void rngstream_DeleteStream (rngstream *g);
```

Deletes the stream `*g` created previously by `rngstream_CreateStream`, and recovers its memory. Otherwise, does nothing.

```
void rngstream_ResetStartStream (rngstream g);
```

Reinitializes the stream `g` to its initial state: C_g is set to I_g.

```
int rngstream_SetSeed (rngstream g, int64_t seed[6]);
```

Sets the initial seed I_g of stream `g` to the vector `seed`. This vector must satisfy the same conditions as in `rngstream_SetPackageSeed`. The stream is then reset to this initial seed. The states and seeds of the other streams are not modified. As a result, after calling this procedure, the initial seeds of the streams are no longer spaced 2^{127} values apart. This function should be invoked only at the beginning of the program. Returns 0 for valid seeds and exits otherwise.

```
void rngstream_GetState (rngstream g, int64_t seed[6]);
```

Returns in `seed[]` the current state C_g of stream `g`. This is convenient if we want to save the state for subsequent use.

```
void rngstream_WriteState (rngstream g);
```

Prints (to standard output) the current state of stream `g`.

```
double rngstream_RandU01 (rngstream g);
```

Returns a (pseudo)random number from the uniform distribution over the interval $(0, 1)$, using stream `g`, after advancing the state by one step. The returned number has 32 bits of precision in the sense that it is always a multiple of $1/(2^{32} - 208)$.

```
int rngstream_RandInt (rngstream g, int i, int j);
```

Returns a (pseudo)random number from the discrete uniform distribution over the integers $\{i, i+1, \ldots, j\}$, using stream `g`. Makes one call to `rngstream_RandU01`.

chrono

This module acts as an interface to the system clock to compute the CPU time used by parts of a program. Every variable of type `chrono_Chrono` acts as an independent *stopwatch*. Several such stopwatchs can run at any given time. An object of type `chrono_Chrono` must be declared for each of them. The function `chrono_Init` resets the stopwatch to zero, `chrono_Val` returns its current reading, and `chrono_Write` writes this reading to the current output. The returned value includes part of the execution time of the functions from module `chrono`. The `chrono_TimeFormat` allows one to choose the kind of time units that are used.

Below is an example of how the functions may be used. A stopwatch named `mytimer` is declared and created. After 2.1 seconds of CPU time have been consumed, the stopwatch is read and reset. Then, after an additional 330 seconds (or 5.5 minutes) of CPU time the stopwatch is read again, printed to the output and deleted.

```
double t;
chrono_Chrono *mytimer = chrono_Create ();
     ⋮    (suppose 2.1 CPU seconds are used here.)
t = chrono_Val (mytimer, chrono_sec);     /* Here, t = 2.1 */
chrono_Init (mytimer);
     ⋮    (suppose 330 CPU seconds are used here.)
t = chrono_Val (mytimer, chrono_min);     /* Here, t = 5.5 */
chrono_Write (mytimer, chrono_hms);       /* Prints:  00:05:30.00 */
chrono_Delete (mytimer);
```

On Linux-Unix systems, this module uses the POSIX function `times` to get the CPU time used by a program. On a Windows platform (when the macro `HAVE_WINDOWS_H` is defined), the Windows function `GetProcessTimes` is used instead.

Types

```
typedef struct {
   unsigned long microsec;
   unsigned long second;
   } chrono_Chrono;
```

For every stopwatch needed, the user must declare a variable of this type and initialize it by calling `chrono_Create`.

```
typedef enum {
   chrono_sec,
   chrono_min,
   chrono_hours,
   chrono_days,
   chrono_hms
   } chrono_TimeFormat;
```

Types of units in which the time on a `chrono_Chrono` can be read or printed: in seconds (`sec`)), minutes (`min`), hours (`hour`), days (`days`), or in the `HH:MM:SS.xx` format, with hours, minutes, seconds and hundreths of a second (`hms`).

Timing functions

```
chrono_Chrono * chrono_Create (void);
```

Creates and returns a stopwatch, after initializing it to zero. This function must be called for each new `chrono_Chrono` used. One may reinitializes it later by calling `chrono_Init`.

```
void chrono_Delete (chrono_Chrono * C);
```

Deletes the stopwatch `C`.

```
void chrono_Init (chrono_Chrono * C);
```

Initializes the stopwatch `C` to zero.

```
double chrono_Val (chrono_Chrono * C, chrono_TimeFormat Unit);
```

Returns the time used by the program since the last call to `chrono_Init(C)`. The parameter `Unit` specifies the time unit. Restriction: `Unit = chrono_hms` is not allowed here.

```
void chrono_Write (chrono_Chrono * C, chrono_TimeFormat Unit);
```

Prints the CPU time used by the program since its last call to `chrono_Init(C)`. The parameter `Unit` specifies the time unit.

tcode

This small program extracts compilable code from a TeX or LaTeX document that contains the documentation. It creates a file `FOut` for a compiler like cc (or any other), starting from a file `FIn`. The names of these two files must be given by the user, with appropriate extensions, when calling the program. The two file names (with the extensions) must be different.

Only the text included between the `\code` and `\endcode` delimiters will appear in the second file. Only the following LaTeX commands are allowed between `\code` and `\endcode`:

`\hide`, `\endhide`, `\iffalse`, `\fi`, `\smallcode`, `\smallc`.

Everything else between `\code` and `\endcode` must be legal code in the output file, apart from two exceptions: the TeX command `\def\code`, defining `\code` will not start a region of valid code, nor will `\code` appearing on a line after a TeX comment character `%`.

If one wants code to appear in the compilable file, but be invisible in the LaTeX output file (e.g., `.pdf` or `.dvi`), it suffices to put this code between the delimiters `\hide` and `\endhide`, or between the delimiters `\iffalse` and `\fi`.

The program is called as:

```
tcode  ⟨FIn⟩  ⟨FOut⟩
```

Examples: The following command extracts the *C* code from the LaTeX file `chrono.tex`, and place it in the header file `chrono.h`:

```
tcode  chrono.tex  chrono.h
```

To extract *Java* code from the LaTeX file `Event.tex`, and place it in the file `Event.java`, one would use:

```
tcode  Event.tex  Event.java
```

References

[1] C. W. Clenshaw. Chebychev series for mathematical functions. National Physical Laboratory Mathematical Tables 5, Her Majesty's Stationery Office, London, 1962.

[2] D. E. Knuth. *The Art of Computer Programming, Vol. 1.* Addison-Wesley, Reading, MA, second edition, 1973.

[3] D. E. Knuth. *The Art of Computer Programming, Vol. 2.* Addison-Wesley, Reading, MA, second edition, 1981.

[4] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

[5] P. L'Ecuyer and S. Côté. Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software*, 17(1):98–111, 1991.

[6] P. L'Ecuyer and F. Panneton. Fast random number generators based on linear recurrences modulo 2: Overview and comparison. In *Proceedings of the 2005 Winter Simulation Conference*, pages 110–119. IEEE Press, 2005.

[7] P. L'Ecuyer and F. Panneton. $\mathbf{F}_2$-linear random number generators. In C. Alexopoulos, D. Goldsman, and J. R. Wilson, editors, *Advancing the Frontiers of Simulation: A Festschrift in Honor of George Samuel Fishman*, pages 169–193. Springer-Verlag, New York, 2009.

[8] P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075, 2002.

[9] F. Panneton. *Construction d'ensembles de points basée sur des récurrences linéaires dans un corps fini de caractéristique 2 pour la simulation Monte Carlo et l'intégration quasi-Monte Carlo*. PhD thesis, Département d'informatique et de recherche opérationnelle, Université de Montréal, Canada, August 2004.