SSJ User's Guide

Package `util`

General basic utilities

Version: December 21, 2006

This document describes a set of basic utilities used in the Java software developed in the *simulation laboratory* of the DIRO, at the Université de Montréal. Many of these tools were originally implemented in the Modula-2 language and have been translated in C and then in Java, with some adaptations along the road.

Contents

Num

This class provides a few constants and some methods to compute numerical quantities such as factorials, combinations, gamma functions, and so on.

```
package umontreal.iro.lecuyer.util;

public class Num
```

Constants

```
public static final double DBL_EPSILON = 2.2204460492503131e-16;
```

Difference between 1.0 and the smallest `double` greater than 1.0.

```
public static final int DBL_MAX_EXP = 1024;
```

Largest `int` x such that 2^{x-1} is representable (approximately) as a `double`.

```
public static final int DBL_MIN_EXP = -1021;
```

Smallest `int` x such that 2^{x-1} is representable (approximately) as a normalised `double`.

```
public static final int DBL_MAX_10_EXP = 308;
```

Largest `int` x such that 10^x is representable (approximately) as a `double`.

```
public static final double DBL_MIN = 2.2250738585072014e-308;
```

Smallest normalized positive floating-point `double`.

```
public static final double LN_DBL_MIN = -708.3964185322641;
```

Natural logarithm of `DBL_MIN`.

```
public static final int DBL_DIG = 15;
```

Number of decimal digits of precision in a `double`.

```
public static final double EBASE = 2.7182818284590452354;
```

The constant e.

```
public static final double EULER = 0.57721566490153286;
```

The Euler-Mascheroni constant.

```
public static final double RAC2 = 1.41421356237309504880;
```

The value of $\sqrt{2}$.

```
public static final double IRAC2 = 0.70710678118654752440;
```

The value of $1/\sqrt{2}$.

`public static final double LN2 = 0.69314718055994530941;`

The values of $\ln 2$.

`public static final double ILN2 = 1.44269504088896340737;`

The values of $1/\ln 2$.

`public static final double MAXINTDOUBLE = 9007199254740992.0;`

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

`public static final double MAXTWOEXP = 64;`

Powers of 2 up to `MAXTWOEXP` are stored exactly in the array `TWOEXP`.

`public static final double TWOEXP[]`

Contains the precomputed positive powers of 2. One has `TWOEXP[j]`$= 2^j$, for $j = 0, \ldots, 64$.

`public static final double TEN_NEG_POW[]`

Contains the precomputed negative powers of 10. One has `TEN_NEG_POW[j]`$= 10^{-j}$, for $j = 0, \ldots, 16$.

Methods

`public static double log2 (double x)`

Returns $\log_2$(`x`).

`public static double log1p (double x)`

Returns a value equivalent to $\log(1 + x)$ accurate also for small x. [1]

`public static double factorial (int n)`

Returns the value of $n!$.

`public static double lnFactorial (int n)`

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

`public static double lnGamma (double x)`

Returns the natural logarithm of the gamma function $\Gamma(x)$ evaluated at `x`. Gives 16 decimals of precision, but is implemented only for $x > 0$.

`public static double digamma (double x)`

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

[1] From Richard: Cette fonction existe dans JDK-1.5. Voir `Math.log1p`

```
public static double trigamma (double x)
```

Returns the value of the trigamma function $d\psi(x)/dx$, the derivative of the digamma function, evaluated at x.

```
public static double tetragamma (double x)
```

Returns the value of the tetragamma function $d^2\psi(x)/d^2x$, the second derivative of the digamma function, evaluated at x.

```
public static double combination (int n, int s)
```

Returns the value of $\binom{n}{s}$, the number of different combinations of s objects amongst n. Uses an algorithm that prevents overflows (when computing factorials), if possible.

```
public static double[][] calcMatStirling (int m, int n)
```

Computes and returns the Stirling numbers of the second kind

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

```
public static double volumeSphere (double p, int t)
```

Returns the volume V of a sphere of radius 1 in t dimensions using the norm L_p. It is given by the formula

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

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

```
public static double evalCheby (double S[], int N, double x)
```

Evaluates a series of Chebyshev polynomials T_j at x over the basic interval $[-1,\ 1]$, using the method of Clenshaw [1], i.e., computes and returns

$$y = \frac{S_0}{2} + \sum_{j=1}^{N} S_j T_j(x).$$

```
public static double evalChebyStar (double S[], int N, double x)
```

Evaluates a series of shifted Chebyshev polynomials T_j^* at x over the basic interval $[0,\ 1]$, using the method of Clenshaw [1], i.e., computes and returns

$$y = \frac{S_0}{2} + \sum_{j=1}^{N} S_j T_j^*(x).$$

```
public static double 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 $x > 10^{-300}$.

`public static int multMod (int a, int s, int c, int m)`

Returns $(as + c) \bmod m$. Restriction: assumes that $a, c, s < m$.

`public static long multMod (long a, long s, long c, long m)`

Returns $(as + c) \bmod m$. Uses the class `ArithmeticMod`. Restriction: assumes that $a, c, s < m$.

`public static double multMod (double a, double s, double c, double m)`

Returns $(as + c) \bmod m$. Restriction: assumes that a, s, c are $< m$ and a, s, c, m are $< 2^{35}$.

TextDataReader

Provides static methods to read data from text files.

```
package umontreal.iro.lecuyer.util;
```

```
public class TextDataReader
```

```
public static double[] readDoubleData (Reader input) throws IOException
```

Reads an array of double-precision values from the reader `input`. For each line of text obtained from the given reader, this method trims whitespaces, and parses the remaining text as a double-precision value. This method ignores every character other than the digits, the plus and minus signs, the period (`.`), and the letters `e` and `E`. Moreover, lines starting with a pound sign (`#`) are considered as comments and thus skipped. The method returns an array containing all the parsed values.

```
public static double[] readDoubleData (File file) throws IOException
```

Opens the file referred to by the file object `file`, and calls `readDoubleData` to obtain an array of double-precision values from the file.

```
public static double[] readDoubleData (String file) throws IOException
```

Opens the file with name `file`, and calls `readDoubleData` to obtain an array of double-precision values from the file.

```
public static int[] readIntData (Reader input) throws IOException
```

This is equivalent to `readDoubleData`, for reading integers.

```
public static int[] readIntData (File file) throws IOException
```

This is equivalent to `readDoubleData`, for reading integers.

```
public static int[] readIntData (String file) throws IOException
```

This is equivalent to `readDoubleData`, for reading integers.

```
public static double[][] readDoubleData2D (Reader input) throws IOException
```

Uses the reader `input` to obtain a 2-dimensional array of double-precision values. For each line of text obtained from the given reader, this method trims whitespaces, and parses the remaining text as an array of double-precision values. Every character other than the digits, the plus (`+`) and minus (`-`) signs, the period (`.`), and the letters `e` and `E` are ignored and can be used to separate numbers on a line. Moreover, lines starting with a pound sign (`#`) are considered as comments and thus skipped. The lines containing only a semicolon sign (`;`) are considered as empty lines. The method returns a 2D array containing all the parsed values. The returned array is not always rectangular.

```
public static double[][] readDoubleData2D (File file) throws IOException
```

Opens the file referred to by the file object `file`, and calls `readDoubleData2D` to obtain a matrix of double-precision values from the file.

`public static double[][] readDoubleData2D (String file) throws IOException`

Opens the file with name `file`, and calls `readDoubleData2D` to obtain a matrix of double-precision values from the file.

`public static int[][] readIntData2D (Reader input) throws IOException`

This is equivalent to `readDoubleData2D`, for reading integers.

`public static int[][] readIntData2D (File file) throws IOException`

This is equivalent to `readDoubleData2D`, for reading integers.

`public static int[][] readIntData2D (String file) throws IOException`

This is equivalent to `readDoubleData2D`, for reading integers.

PrintfFormat

This class acts like a `StringBuffer` which defines new types of `append` methods. It defines certain functionalities of the ANSI C `printf` function that also can be accessed through static methods. The information given here is strongly inspired from the `man` page of the C `printf` function.

```
package umontreal.iro.lecuyer.util;

public class PrintfFormat
```

Constructors

`public PrintfFormat()`

Constructs a new buffer object containing an empty string.

`public PrintfFormat (int length)`

Constructs a new buffer object with an initial capacity of `length`.

`public PrintfFormat (String str)`

Constructs a new buffer object containing the initial string `str`.

Methods

`public PrintfFormat append (String str)`

Appends `str` to the buffer.

`public PrintfFormat append (int fieldwidth, String str)`

Uses the `s` static method to append `str` to the buffer. A minimum of `fieldwidth` characters will be used.

`public PrintfFormat append (double x)`

Appends `x` to the buffer.

`public PrintfFormat append (int fieldwidth, double x)`

Uses the `f` static method to append `x` to the buffer. A minimum of `fieldwidth` characters will be used.

`public PrintfFormat append (int fieldwidth, int precision, double x)`

Uses the `f` static method to append `x` to the buffer. A minimum of `fieldwidth` characters will be used with the given `precision`.

`public PrintfFormat append (int x)`

Appends `x` to the buffer.

`public PrintfFormat append (int fieldwidth, int x)`

Uses the `d` static method to append `x` to the buffer. A minimum of `fieldwidth` characters will be used.

`public PrintfFormat append (long x)`

Appends `x` to the buffer.

`public PrintfFormat append (int fieldwidth, long x)`

Uses the `d` static method to append `x` to the buffer. A minimum of `fieldwidth` characters will be used.

```
public PrintfFormat append (int fieldwidth, int accuracy, int precision,
                            double x)
```

Uses the `format` static method with the same four arguments to append `x` to the buffer.

`public PrintfFormat append (char c)`

Appends a single character to the buffer.

`public void clear()`

Clears the contents of the buffer.

`public StringBuffer getBuffer()`

Returns the `StringBuffer` associated with that object.

`public String toString()`

Converts the buffer into a `String`.

`public static String s (String str)`

Same as `s (0, str)`. If the string `str` is null, it returns the string “null”.

`public static String s (int fieldwidth, String str)`

Formats the string `str` like the `%s` in the C `printf` function. The `fieldwidth` argument gives the minimum length of the resulting string. If `str` is shorter than `fieldwidth`, it is left-padded with spaces. If `fieldwidth` is negative, the string is right-padded with spaces if necessary. The `String` will never be truncated. If `str` is null, it calls `s (fieldwidth, ‘‘null’’)`.

The `fieldwidth` argument has the same effect for the other methods in this class.

Integers

`public static String d (long x)`

Same as `d (0, 1, x)`.

`public static String d (int fieldwidth, long x)`

Same as `d (fieldwidth, 1, x)`.

`public static String d (int fieldwidth, int precision, long x)`

Formats the long integer `x` into a string like `%d` in the C `printf` function. It converts its argument to decimal notation, `precision` gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. When zero is printed with an explicit precision 0, the output is empty.

If the one-argument form is used, a `fieldwidth` of 0 is assumed and a `precision` of 1 is used. If the two-arguments method is used, a `precision` of 1 is assumed.

`public static String format (long x)`

Same as `d (0, 1, x)`.

`public static String format (int fieldwidth, long x)`

Converts a long integer to a `String` with a minimum length of `fieldwidth`, the result is right-padded with spaces if necessary but it is not truncated. If only one argument is specified, a `fieldwidth` of 0 is assumed.

`public static String formatBase (int b, long x)`

Same as `formatBase (0, b, x)`.

`public static String formatBase (int fieldwidth, int b, long x)`

Converts the integer `x` to a `String` representation in base `b`.

Restrictions: $2 \le \texttt{b} \le 10$

Reals

`public static String E (double x)`

Same as `E (0, 6, x)`.

`public static String E (int fieldwidth, double x)`

Same as `E (fieldwidth, 6, x)`.

`public static String E (int fieldwidth, int precision, double x)`

Formats a double-precision number `x` like `%E` in C `printf`. The double argument is rounded and converted in the style `[-]d.dddE+-dd` where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is 0, no decimal-point character appears. The exponent always contains at least two digits; if the value is zero, the exponent is `00`.

If the one-argument form is used, a `fieldwidth` of 0 and a `precision` of 6 are used. If the two-arguments form is used, a `precision` of 6 is assumed.

`public static String e (double x)`

Same as `e (0, 6, x)`.

`public static String e (int fieldwidth, double x)`

Same as `e (fieldwidth, 6, x)`.

`public static String e (int fieldwidth, int precision, double x)`

The same as `E`, except that `'e'` is used as the exponent character instead of `'E'`.

`public static String f (double x)`

Same as `f (0, 6, x)`.

`public static String f (int fieldwidth, double x)`

Same as `f (fieldwidth, 6, x)`.

`public static String f (int fieldwidth, int precision, double x)`

Formats the double-precision `x` into a string like `%f` in C `printf`. The argument is rounded and converted to decimal notation in the style `[-]ddd.ddd`, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is explicitly 0, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.

If the one-argument form is used, a `fieldwidth` of 0 and a `precision` of 6 are used. If the two-arguments form is used, a `precision` of 6 is assumed.

`public static String G (double x)`

Same as `G (0, 6, x)`.

`public static String G (int fieldwidth, double x)`

Same as `G (fieldwidth, 6, x)`.

`public static String G (int fieldwidth, int precision, double x)`

Formats the double-precision `x` into a string like `%G` in C `printf`. The argument is converted in style `%f` or `%E`. `precision` specifies the number of significant digits. If it is 0, it is treated as 1. Style `%E` is used if the exponent from its conversion is less than -4 or greater than or equal to `precision`. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.

If the one-argument form is used, a `fieldwidth` of 0 and a `precision` of 6 are used. If the two-arguments form is used, a `precision` of 6 is assumed.

`public static String g (double x)`

Same as `g (0, 6, x)`.

`public static String g (int fieldwidth, double x)`

Same as `g (fieldwidth, 6, x)`.

```
public static String g (int fieldwidth, int precision, double x)
```

The same as `G`, except that `'e'` is used in the scientific notation.

```
public static String format (int fieldwidth, int accuracy, int precision,
                             double x)
```

Returns a `String` containing `x`. Uses a total of at least `fieldwidth` positions (including the sign and point when they appear), `accuracy` digits after the decimal point and at least `precision` significant digits. `accuracy` and `precision` must be strictly smaller than `fieldwidth`. The number is rounded if necessary. If there is not enough space to format the number in decimal notation with at least `precision` significant digits (`accuracy` or `fieldwidth` is too small), it will be converted to scientific notation with at least `precision` significant digits. In that case, `fieldwidth` is increased if necessary.

Intervals

```
public static void formatWithError (int fieldwidth, int fieldwidtherr,
     int accuracy, int precision, double x, double error, String[] res)
```

Stores a string containing `x` into `res[0]`, and a string containing `error` into `res[1]`, both strings being formatted with the same notation. Uses a total of at least `fieldwidth` positions (including the sign and point when they appear) for `x`, `fieldwidtherr` positions for `error`, `accuracy` digits after the decimal point and at least `precision` significant digits. `accuracy` and `precision` must be strictly smaller than `fieldwidth`. The numbers are rounded if necessary. If there is not enough space to format `x` in decimal notation with at least `precision` significant digits (`accuracy` or `fieldwidth` are too small), it will be converted to scientific notation with at least `precision` significant digits. In that case, `fieldwidth` is increased if necessary, and the error is also formatted in scientific notation.

```
public static void formatWithError (int fieldwidth, int fieldwidtherr,
    int precision, double x, double error, String[] res)
```

Stores a string containing `x` into `res[0]`, and a string containing `error` into `res[1]`, both strings being formatted with the same notation. This calls `formatWithError` with the minimal accuracy for which the formatted string for `error` is non-zero. If `error` is 0, the accuracy is 0. If this minimal accuracy causes the strings to be formatted using scientific notation, this method increases the accuracy until the decimal notation can be used.

TableFormat

This class provides methods to format arrays and matrices into `String`s in different styles. This could be useful for printing arrays and subarrays, or for putting them in files for further treatment by other softwares such as *Mathematica*, *Matlab*, etc.

```
package umontreal.iro.lecuyer.util;

public class TableFormat
```

Formating styles

```
public static final int PLAIN
```

Plain text matrix printing style

```
public static final int MATHEMATICA
```

Mathematica matrix printing style

```
public static final int MATLAB
```

Matlab matrix printing style

Functions to convert tables to `String`

```
public static String format (int V[], int n1, int n2, int k, int p)
```

Formats a `String` containing the elements `n1` to `n2` (inclusive) of table `V`, `k` elements per line, `p` positions per element. If $k = 1$, the array index will also appear on the left of each element, i.e., each line `i` will have the form `i V[i]`.

```
public static String format (double V[], int n1, int n2,
                             int k, int p1, int p2, int p3)
```

Similar to the previous method, but for an array of `double`'s. Gives at least `p1` positions per element, `p2` digits after the decimal point, and at least `p3` significant digits.

```
public static String format (int[][] Mat, int i1, int i2,
                             int j1, int j2, int w, int p,
                             int style, String Name)
```

Formats a submatrix of integers.

```
public static String format (double[][] Mat, int i1, int i2,
                             int j1, int j2, int w, int p,
                             int style, String Name)
```

Formats the submatrix with lines $i1 \le i \le i2$ and columns $j1 \le j \le j2$ of the matrix `Mat`, using the formatting style `style`. The elements are formated in `w` positions each, with a precision of `p` digits. The string `Name` provides an identifier for the submatrix.

To be treated by `Matlab`, this string containing the matrix must be copied to a file with extension `.m`. If the file is named `poil.m`, for example, it can be accessed by calling `poil` in `Matlab`. For `Mathematica`, if the file is named `poil`, it will be read using `<< poil;`.

Chrono

`Chrono` is a small class that acts as an interface to the system clock and calculates the CPU time consumed by parts of a program. Part of this class is implemented in the C language and the implementation is unfortunately operating system-dependent. The C functions for the current class have been compiled on a 32-bit machine running Linux and will not work on 64-bit machines. For a *platform-independent* CPU timer (valid only with Java–1.5 or later), one should use the subclass `ChronoSingleThread` which is programmed directly in Java (see the next class in this guide). [2]

Every object of class `Chrono` acts as an independent *stopwatch*. Several `Chrono` objects can run at any given time. The method `init` resets the stopwatch to zero, `getSeconds`, `getMinutes` and `getHours` return its current reading, and `format` converts this reading to a `String`. The returned value includes the execution time of the method from `Chrono`.

Below is an example of how it may be used. A stopwatch named `timer` is constructed (and initialized). When 2.1 seconds of CPU have been consumed, the stopwatch is read and reset to zero. Then, after an additional 330 seconds (or 5.5 minutes) of CPU time, the stopwatch is read again and the value is printed to the output in minutes.

```
Chrono timer = Chrono.createForSingleThread();
    ⋮
           (suppose 2.1 CPU seconds are used here.)
double t = timer.getSeconds();               // Here, t = 2.1
timer.init();
t = timer.getSeconds();                      // Here, t = 0.0
    ⋮
           (suppose 330 CPU seconds are used here.)
t = timer.getMinutes();                      // Here, t = 5.5
System.out.println (timer.format());         // Prints: 0:5:30.00
```

```
package umontreal.iro.lecuyer.util;
```

```
public class Chrono
```

```
public static Chrono createForSingleThread()
```

Creates a `Chrono` instance adapted for a program using a single thread. Under Java 1.5, this method returns an instance of `ChronoSingleThread` which can measure CPU time for one thread. Under Java versions prior to 1.5, this returns an instance of this class. This method must not be used to create a timer for a multi-threaded program, because the obtained CPU times will differ depending on the used Java version.

[2] From Richard: Dans les deux cas, le nouveau `Chrono` d'Éric fonctionne bien en appelant `Chrono timer = Chrono.createForSingleThread()`.

Constructor

`public Chrono()`

Constructs a `Chrono` object and initializes it to zero.

Timing functions

`public void init()`

Initializes this `Chrono` to zero.

`public double getSeconds()`

Returns the CPU time in seconds used by the program since the last call to `init` for this `Chrono`.

`public double getMinutes()`

Returns the CPU time in minutes used by the program since the last call to `init` for this `Chrono`.

`public double getHours()`

Returns the CPU time in hours used by the program since the last call to `init` for this `Chrono`.

`public String format()`

Converts the CPU time used by the program since its last call to `init` for this `Chrono` to a `String` in the `HH:MM:SS.xx` format.

`public static String format (double time)`

Converts the time `time`, given in seconds, to a `String` in the `HH:MM:SS.xx` format.

ChronoSingleThread

The `ChronoSingleThread` class extends the `Chrono` class and computes the CPU time for the current thread only. It is valid only under Java–1.5 since Java–1.5 provides platform-independent facilities to get the CPU time for a single thread through management API. The parent class `Chrono` uses a platform-dependent method (since it is programmed directly in C) to determine the CPU time for all threads. Here is an example of how it may be used:

```
Chrono timer = new ChronoSingleThread();
   ⋮
double t = timer.getSeconds();
timer.init();
t = timer.getSeconds();
   ⋮
t = timer.getMinutes();
System.out.println (timer.format());
```

```
package umontreal.iro.lecuyer.util;

public class ChronoSingleThread extends Chrono
```

Constructor

`public ChronoSingleThread()`

Constructs a `ChronoSingleThread` object and initializes it to zero.

ArithmeticMod

This class provides facilities to compute multiplications of scalars, of vectors and of matrices modulo m. All algorithms are present in three different versions. These allow operations on `double`, `int` and `long`. The `int` and `long` versions work exactly like the `double` ones.

```
package umontreal.iro.lecuyer.util;

public class ArithmeticMod
```

Methods using `double`

```
public static double multModM (double a, double s, double c, double m)
```

Computes $(a \times s + c) \bmod m$. Where `m` must be smaller than 2^{35}. Works also if `s` or `c` are negative. The result is always positive (and thus always between 0 and `m` - 1).

```
public static void matVecModM (double A[][], double s[], double v[],
                               double m)
```

Computes the result of $\mathtt{A} \times \mathtt{s} \bmod m$ and puts the result in `v`. Where `s` and `v` are both column vectors. This method works even if `s` = `v`.

```
public static void matMatModM (double A[][], double B[][], double C[][],
                               double m)
```

Computes $\mathtt{A} \times \mathtt{B} \bmod m$ and puts the result in `C`. Works even if `A` = `C`, `B` = `C` or `A` = `B` = `C`.

```
public static void matTwoPowModM (double A[][], double B[][], double m,
                                  int e)
```

Computes $\mathtt{A}^{2^{\mathtt{e}}} \bmod m$ and puts the result in `B`. Works even if `A` = `B`.

```
public static void matPowModM (double A[][], double B[][], double m,
                               int c)
```

Computes $\mathtt{A}^{c} \bmod m$ and puts the result in `B`. Works even if `A` = `B`.

Methods using `int`

```
public static int multModM (int a, int s, int c, int m)
```

Computes $(a \times s + c) \bmod m$. Works also if `s` or `c` are negative. The result is always positive (and thus always between 0 and `m` - 1).

```
public static void matVecModM (int A[][], int s[], int v[], int m)
```

Exactly like `matVecModM` using `double`, but with `int` instead of `double`.

```
public static void matMatModM (int A[][], int B[][], int C[][], int m)
```

Exactly like `matMatModM` using `double`, but with `int` instead of `double`.

`public static void matTwoPowModM (int A[][], int B[][], int m, int e)`

Exactly like `matTwoPowModM` using `double`, but with `int` instead of `double`.

`public static void matPowModM (int A[][], int B[][], int m, int c)`

Exactly like `matPowModM` using `double`, but with `int` instead of `double`.

Methods using `long`

`public static long multModM (long a, long s, long c, long m)`

Computes $(a \times s + c) \bmod m$. Works also if `s` or `c` are negative. The result is always positive (and thus always between 0 and `m` - 1).

`public static void matVecModM (long A[][], long s[], long v[], long m)`

Exactly like `matVecModM` using `double`, but with `long` instead of `double`.

`public static void matMatModM (long A[][], long B[][], long C[][], long m)`

Exactly like `matMatModM` using `double`, but with `long` instead of `double`.

`public static void matTwoPowModM (long A[][], long B[][], long m, int e)`

Exactly like `matTwoPowModM` using `double`, but with `long` instead of `double`.

`public static void matPowModM (long A[][], long B[][], long m, int c)`

Exactly like `matPowModM` using `double`, but with `long` instead of `double`.

BitVector

This class implements vectors of bits and the operations needed to use them. The vectors can be of arbitrary length. The operations provided are all the binary operations available to the `int` and `long` primitive types in Java.

All bit operations are present in two forms: a normal form and a `self` form. The normal form returns a newly created object containing the result, while the `self` form puts the result in the calling object (`this`). The return value of the `self` form is the calling object itself. This is done to allow easier manipulation of the results, making it possible to chain operations.

```
package umontreal.iro.lecuyer.util;

public class BitVector implements Serializable, Cloneable
```

Constructors

`public BitVector (int length)`

Creates a new `BitVector` of length `length` with all its bits set to 0.

`public BitVector (int[] vect, int length)`

Creates a new `BitVector` of length `length` using the data in `vect`. Component `vect[0]` makes the 32 lowest order bits, with `vect[1]` being the 32 next lowest order bits, and so on. The normal bit order is then used to fill the 32 bits (the first bit is the lowest order bit and the last bit is largest order bit). Note that the sign bit is used as the largest order bit.

`public BitVector (int[] vect)`

Creates a new `BitVector` using the data in `vect`. The length of the `BitVector` is always equals to 32 times the length of `vect`.

`public BitVector (BitVector that)`

Creates a copy of the `BitVector that`.

Methods

`public Object clone()`

Creates a copy of the `BitVector`.

`public boolean equals (BitVector that)`

Verifies if two `BitVector`'s have the same length and the same data.

`public int size()`

Returns the length of the `BitVector`.

`public void enlarge (int size, boolean filling)`

Resizes the `BitVector` so that its length is equal to `size`. If the `BitVector` is enlarged, then the newly added bits are given the value 1 if `filling` is set to `true` and 0 otherwise.

`public void enlarge (int size)`

Resizes the `BitVector` so that its length is equal to `size`. Any new bit added is set to 0.

`public boolean getBool (int pos)`

Gives the value of the bit in position `pos`. If the value is 1, returns `true`; otherwise, returns `false`.

`public void setBool (int pos, boolean value)`

Sets the value of the bit in position `pos`. If `value` is equal to `true`, sets it to 1; otherwise, sets it to 0.

`public int getInt (int pos)`

Returns an `int` containing all the bits in the interval $[\texttt{pos} \times 32, \texttt{pos} \times 32 + 31]$.

`public String toString()`

Returns a string containing all the bits of the `BitVector`, starting with the highest order bit and finishing with the lowest order bit. The bits are grouped by groups of 8 bits for ease of reading.

`public BitVector not()`

Returns a `BitVector` which is the result of the `not` operator on the current `BitVector`. The `not` operator is equivalent to the `~` operator in Java and thus swap all bits (bits previously set to 0 become 1 and bits previously set to 1 become 0).

`public BitVector selfNot()`

Applies the `not` operator on the current `BitVector` and returns it.

`public BitVector xor (BitVector that)`

Returns a `BitVector` which is the result of the `xor` operator applied on `this` and `that`. The `xor` operator is equivalent to the `^` operator in Java. All bits which were set to 0 in one of the vector and to 1 in the other vector are set to 1. The others are set to 0. This is equivalent to the addition in modulo 2 arithmetic.

`public BitVector selfXor (BitVector that)`

Applies the `xor` operator on `this` with `that`. Stores the result in `this` and returns it.

`public BitVector and (BitVector that)`

Returns a `BitVector` which is the result of the `and` operator with both the `this` and `that` `BitVector`'s. The `and` operator is equivalent to the `&` operator in Java. Only bits which are set to 1 in both `this` and `that` are set to 1 in the result, all the others are set to 0.

`public BitVector selfAnd (BitVector that)`

Applies the `and` operator on `this` with `that`. Stores the result in `this` and returns it.

`public BitVector or (BitVector that)`

Returns a `BitVector` which is the result of the `or` operator with both the `this` and `that` `BitVector`'s. The `or` operator is equivalent to the `|` operator in Java. Only bits which are set to 0 in both `this` and `that` are set to to 0 in the result, all the others are set to 1.

`public BitVector selfOr (BitVector that)`

Applies the `or` operator on `this` with `that`. Stores the result in `this` and returns it.

`public BitVector shift (int j)`

Returns a `BitVector` equal to the original with all the bits shifted `j` positions to the right if `j` is positive, and shifted `j` positions to the left if `j` is negative. The new bits that appears to the left or to the right are set to 0. If `j` is positive, this operation is equivalent to the `>>>` operator in Java, otherwise, it is equivalent to the `<<` operator.

`public BitVector selfShift (int j)`

Shift all the bits of the current `BitVector` `j` positions to the right if `j` is positive, and `j` positions to the left if `j` is negative. The new bits that appears to the left or to the rigth are set to 0. Returns `this`.

`public boolean scalarProduct (BitVector that)`

Returns the scalar product of two `BitVector`'s modulo 2. It returns `true` if there is an odd number of bits with a value of 1 in the result of the `and` operator applied on `this` and `that`, and returns `false` otherwise.

BitMatrix

This class implements matrices of bits of arbitrary dimensions. Basic facilities for bits operations, multiplications and exponentiations are provided.

```
package umontreal.iro.lecuyer.util;

public class BitMatrix implements Serializable, Cloneable
```

Constructors

`public BitMatrix (int r, int c)`

Creates a new `BitMatrix` with `r` rows and `c` columns filled with 0's.

`public BitMatrix (BitVector[] rows)`

Creates a new `BitMatrix` using the data in `rows`. Each of the `BitVector` will be one of the rows of the `BitMatrix`.

`public BitMatrix (int[][] data, int r, int c)`

Creates a new `BitMatrix` with `r` rows and `c` columns using the data in `data`. Note that the orders of the bits for the rows are using the same order than for the `BitVector`. This does mean that the first bit is the lowest order bit of the last `int` in the row and the last bit is the highest order bit of the first `int` int the row.

`public BitMatrix (BitMatrix that)`

Copy constructor.

Methods

`public Object clone()`

Creates a copy of the `BitMatrix`.

`public boolean equals (BitMatrix that)`

Verifies that `this` and `that` are strictly identical. They must both have the same dimensions and data.

`public String toString()`

Creates a `String` containing all the data of the `BitMatrix`. The result is displayed in a matrix form, with each row being put on a different line. Note that the bit at (0,0) is at the upper left of the matrix, while the bit at (0) in a `BitVector` is the least significant bit.

`public String printData()`

Creates a `String` containing all the data of the `BitMatrix`. The data is displayed in the same format as are the `int[][]` in `Java` code. This allows the user to print the representation of a `BitMatrix` to be put, directly in the source code, in the constructor `BitMatrix(int[][], int, int)`. The output is not designed to be human-readable.

`public int numRows()`

Returns the number of rows of the `BitMatrix`.

`public int numColumns()`

Returns the number of columns of the `BitMatrix`.

`public boolean getBool (int row, int column)`

Returns the value of the bit in the specified row and column. If the value is 1, return `true`. If it is 0, return `false`.

`public void setBool (int row, int column, boolean value)`

Changes the value of the bit in the specified row and column. If `value` is `true`, changes it to 1. If `value` is `false` changes it to 0.

`public BitMatrix transpose()`

Returns the transposed matrix. The rows and columns are interchanged.

`public BitMatrix not()`

Returns the `BitMatrix` resulting from the application of the `not` operator on the original `BitMatrix`. The effect is to swap all the bits of the `BitMatrix`, turning all 0 into 1 and all 1 into 0.

`public BitMatrix and (BitMatrix that)`

Returns the `BitMatrix` resulting from the application of the `and` operator on the original `BitMatrix` and `that`. Only bits which were at 1 in both `BitMatrix` are set at 1 in the result. All others are set to 0.

`public BitMatrix or (BitMatrix that)`

Returns the `BitMatrix` resulting from the application of the `or` operator on the original `BitMatrix` and `that`. Only bits which were at 0 in both `BitMatrix` are set at 0 in the result. All others are set to 1.

`public BitMatrix xor (BitMatrix that)`

Returns the `BitMatrix` resulting from the application of the `xor` operator on the original `BitMatrix` and `that`. Only bits which were at 1 in only one of the two `BitMatrix` are set at 1 in the result. All others are set to 0.

`public BitVector multiply (BitVector vect)`

Multiplies the column `BitVector` by a `BitMatrix` and returns the result. The result is $A \times v$, where A is the `BitMatrix`, and v is the `BitVector`.

`public int multiply (int vect)`

Multiplies `vect`, seen as a column `BitVector`, by a `BitMatrix`. (See `BitVector` to see the conversion between `int` and `BitVector`.) The result is $A \times v$, where A is the `BitMatrix`, and v is the `BitVector`.

`public BitMatrix multiply (BitMatrix that)`

Multiplies two `BitMatrix`'s together. The result is $A \times B$, where A is the `this BitMatrix` and B is the `that BitMatrix`.

`public BitMatrix power (long p)`

Raises the `BitMatrix` to the power `p`.

`public BitMatrix power2e (int e)`

Raises the `BitMatrix` to power $2^{\mathtt{e}}$.

Nested Class

`public class IncompatibleDimensionException extends RuntimeException`

Runtime exception raised when the dimensions of the `BitMatrix` are not appropriate for the operation.

MathFunction

This interface should be implemented by classes which represent univariate mathematical functions. It is used to pass an arbitrary function of one variable as argument to another function. For example, it is used in `RootFinder` to find the zeros of a function.

```
package umontreal.iro.lecuyer.util;

public interface MathFunction
```

```
public double evaluate (double x);
```

Returns the value of the function evaluated at x.

RootFinder

This class provides methods to solve non-linear equations.

```
package umontreal.iro.lecuyer.util;


public class RootFinder
```

Methods

```
public static double brentDekker (double a, double b,
                                  MathFunction f, double tol)
```

Computes a root x of the function in `f` using the Brent-Dekker method. The interval $[a, b]$ must contain the root x. The calculations are done with an approximate relative precision `tol`. Returns x such that $f(x) = 0$.

Misc

This class provides miscellaneous functions that are hard to classify. Some may be moved to another class in the future.

```
package umontreal.iro.lecuyer.util;

public class Misc
```

Methods

```
public static double quickSelect (double[] t, int n, int k)
```

Returns the k^{th} smallest item of the array t of size n.

```
public static int quickSelect (int[] t, int n, int k)
```

Returns the k^{th} smallest item of the array t of size n.

```
public static int getTimeInterval (double[] times, int start,
                                   int end, double t)
```

Returns the index of the time interval corresponding to time `t`. Let $t_0 \le \cdots \le t_n$ be simulation times stored in a subset of `times`. This method uses binary search to determine the smallest value i for which $t_i \le t < t_{i+1}$, and returns i. The value of t_i is stored in `times[start+i]` whereas n is defined as `end - start`. If $t < t_0$, this returns -1. If $t \ge t_n$, this returns n. Otherwise, the returned value is greater than or equal to 0, and smaller than or equal to $n - 1$. `start` and `end` are only used to set lower and upper limits of the search in the `times` array; the index space of the returned value always starts at 0. Note that if the elements of `times` with indices `start`, ..., `end` are not sorted in non-decreasing order, the behavior of this method is undefined.

```
public static void interpol (int n, double[] X, double[] Y, double[] C)
```

Given the $n + 1$ distinct points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$ [with `X[0]` $= x_i$ and similarly for Y and C], this function computes the $n+1$ coefficients `C[i]` of the Newton interpolating polynomial $P(x)$ of degree n passing through these points:

$$P(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \cdots + c_n(x - x_0)(x - x_1) \cdots (x - x_{n-1}).$$

```
public static double evalPoly (int n, double[] X, double[] C, double z)
```

Given n, X and C as described in `interpol(n, X, Y, C)`, this function returns the value of the interpolating polynomial evaluated at z.

```
public static double simpsonIntegral (MathFunction func, double a,
                                      double b, int numIntervals)
```

Computes and returns an approximation of the integral of `func` over $[a, b]$, using the Simpsons 1/3 method with `numIntervals` intervals. This method estimates

$$\int_a^b f(x)dx,$$

where $f(x)$ is the function defined by `func` and evaluated at x, by dividing $[a, b]$ in $n =$ `numIntervals` interval with length $h = (b - a)/n$. The integral is estimated by

$$\frac{h}{3}(f(a) + 4f(a + h) + 2f(a + 2h) + 4f(a + 3h) + \cdots + f(b))$$

This method assumes that $a \leq b < \infty$, and n is even.

JDBCManager

This class provides some facilities to connect to a SQL database and to retrieve data stored in it. JDBC provides a standardized interface for accessing a database independently of a specific database management system (DBMS). The user of JDBC must create a `Connection` object used to send SQL queries to the underlying DBMS, but the creation of the connection adds a DBMS-specific portion in the application. This class helps the developer in moving the DBMS-specific information out of the source code by storing it in a properties file. The methods in this class can read such a properties file and establish the JDBC connection. The connection can be made by using a `DataSource` obtained through a JNDI server, or by a JDBC URI associated with a driver class. Therefore, the properties used to connect to the database must be a JNDI name (`jdbc.jndi-name`), or a driver to load (`jdbc.driver`) with the URI of a database (`jdbc.uri`).

```
jdbc.driver=com.mysql.jdbc.Driver

jdbc.uri=jdbc:mysql://mysql.iro.umontreal.ca/database?user=foo&password=bar
```

The connection is established using the `connectToDatabase` method. Shortcut methods are also available to read the properties from a file or a resource before establishing the connection. This class also provides shortcut methods to read data from a database and to copy the data into Java arrays.

```
package umontreal.iro.lecuyer.util;

public class JDBCManager
```

Methods

```
public static Connection connectToDatabase (Properties prop)
        throws SQLException
```

Connects to the database using the properties `prop` and returns the an object representing the connection. The properties stored in `prop` must be a JNDI name (`jdbc.jndi-name`), or the name of a driver (`jdbc.driver`) to load and the URI of the database (`jdbc.uri`). When a JNDI name is given, this method constructs a context using the nullary constructor of `InitialContext`, uses the context to get a `DataSource` object, and uses the data source to obtain a connection. This method assumes that JNDI is configured correctly; see the class `InitialContext` for more information about configuring JNDI. If no JNDI name is specified, the method looks for a JDBC URI. If a driver class name is specified along with the URI, the corresponding driver is loaded and registered with the JDBC `DriverManager`. The driver manager is then used to obtain the connection using the URI. This method throws an `SQLException` if the connection failed and an `IllegalArgumentException` if the properties do not contain the required values.

```
public static Connection connectToDatabase (InputStream is)
        throws IOException, SQLException
```

Returns a connection to the database using the properties read from stream `is`. This method loads the properties from the given stream, and calls `connectToDatabase` to establish the connection.

```
public static Connection connectToDatabase (File file)
        throws IOException, SQLException
```

Equivalent to `connectToDatabase (new FileInputStream (file))`.

```
public static Connection connectToDatabase (String fileName)
        throws IOException, SQLException
```

Equivalent to `connectToDatabase (new FileInputStream (fileName))`.

```
public static Connection connectToDatabaseFromResource (String resource)
        throws IOException, SQLException
```

Uses `connectToDatabase` with the stream obtained from the resource `resource`. This method searches the file `resource` on the class path, opens the first resource found, and extracts properties from it. It then uses `connectToDatabase` to establish the connection.

```
public static double[] readDoubleData (Statement stmt, String query)
        throws SQLException
```

Copies the result of the SQL query `query` into an array of double-precision values. This method uses the statement `stmt` to execute the given query, and assumes that the first column of the result set contains double-precision values. Each row of the result set then becomes an element of an array of double-precision values which is returned by this method. This method throws an `SQLException` if the query is not valid.

```
public static double[] readDoubleData (Connection connection, String query)
        throws SQLException
```

Copies the result of the SQL query `query` into an array of double-precision values. This method uses the active connection `connection` to create a statement, and passes this statement, with the query, to `readDoubleData`, which returns an array of double-precision values.

```
public static double[] readDoubleData (Statement stmt, String table,
                                       String column)
        throws SQLException
```

Returns the values of the column `column` of the table `table`. This method is equivalent to `readDoubleData (stmt, "SELECT column FROM table")`.

```
public static double[] readDoubleData (Connection connection,
                                       String table, String column)
        throws SQLException
```

Returns the values of the column `column` of the table `table`. This method is equivalent to `readDoubleData (connection, "SELECT column FROM table")`.

```
public static int[] readIntData (Statement stmt, String query)
        throws SQLException
```

Copies the result of the SQL query `query` into an array of integers. This method uses the statement `stmt` to execute the given query, and assumes that the first column of the result set contains integer values. Each row of the result set then becomes an element of an array of integers which is returned by this method. This method throws an `SQLException` if the query is not valid. The given statement `stmt` must not be set up to produce forward-only result sets.

```
public static int[] readIntData (Connection connection, String query)
        throws SQLException
```

Copies the result of the SQL query `query` into an array of integers. This method uses the active connection `connection` to create a statement, and passes this statement, with the query, to `readIntData`, which returns an array of integers.

```
public static int[] readIntData (Statement stmt, String table,
                                 String column)
        throws SQLException
```

Returns the values of the column `column` of the table `table`. This method is equivalent to `readIntData (stmt, "SELECT column FROM table")`.

```
public static int[] readIntData (Connection connection, String table,
                                 String column)
        throws SQLException
```

Returns the values of the column `column` of the table `table`. This method is equivalent to `readIntData (connection, "SELECT column FROM table")`.

```
public static double[][] readDoubleData2D (Statement stmt, String query)
        throws SQLException
```

Copies the result of the SQL query `query` into a rectangular 2D array of double-precision values. This method uses the statement `stmt` to execute the given query, and assumes that the columns of the result set contain double-precision values. Each row of the result set then becomes a row of a 2D array of double-precision values which is returned by this method. This method throws an `SQLException` if the query is not valid. The given statement `stmt` must not be set up to produce forward-only result sets.

```
public static double[][] readDoubleData2D (Connection connection,
                                           String query)
        throws SQLException
```

Copies the result of the SQL query `query` into a rectangular 2D array of double-precision values. This method uses the active connection `connection` to create a statement, and passes this statement, with the query, to `readDoubleData2D`, which returns a 2D array of double-precision values.

```
public static double[][] readDoubleData2DTable (Statement stmt,
                                                String table)
        throws SQLException
```

Returns the values of the columns of the table `table`. This method is equivalent to `readDoubleData2D (stmt, "SELECT * FROM table")`.

```
public static double[][] readDoubleData2DTable (Connection connection,
                                                String table)
        throws SQLException
```

Returns the values of the columns of the table `table`. This method is equivalent to `readDoubleData2D (connection, "SELECT * FROM table")`.

```
public static int[][] readIntData2D (Statement stmt, String query)
        throws SQLException
```

Copies the result of the SQL query `query` into a rectangular 2D array of integers. This method uses the statement `stmt` to execute the given query, and assumes that the columns of the result set contain integers. Each row of the result set then becomes a row of a 2D array of integers which is returned by this method. This method throws an `SQLException` if the query is not valid. The given statement `stmt` must not be set up to produce forward-only result sets.

```
public static int[][] readIntData2D (Connection connection, String query)
        throws SQLException
```

Copies the result of the SQL query `query` into a rectangular 2D array of integers. This method uses the active connection `connection` to create a statement, and passes this statement, with the query, to `readIntData2D`, which returns a 2D array of integers.

```
public static int[][] readIntData2DTable (Statement stmt, String table)
        throws SQLException
```

Returns the values of the columns of the table `table`. This method is equivalent to `readIntData2D (stmt, "SELECT * FROM table")`.

```
public static int[][] readIntData2DTable (Connection connection,
                                          String table)
        throws SQLException
```

Returns the values of the columns of the table `table`. This method is equivalent to `readIntData2D (connection, "SELECT * FROM table")`.

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, Mass., second edition, 1973.