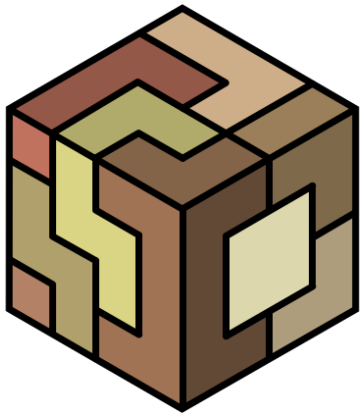
TIFA
Tools for Integer FActorization

An open source library for factoring
small to medium-size composite integers

User's guide

Early draft – work in progress
June 16, 2011



Legal notices

About the TIFA library

The TIFA library has been registered at the french agency for software protection (APP) with the Inter Deposit Digital Number:

`IDDN.FR.001.220019.000.S.A.2011.000.3123`.

About the TIFA logo

About this document

Contents

Document scope

The purpose of this document is to describe the TIFA library from a high level, user-oriented perspective. Specifically, it intends to show you how to:

- use the public TIFA functions in your programs;
- directly run the included stand-alone factorization programs;
- use TIFA's basic benchmark framework for your own tests or programs.

This document does not focus on TIFA's internal functions or architecture, nor is it intended to explain implementation specifics. Readers interested in TIFA's inner working are invited to refer to the detailled Doxygen documentation.

Furthermore, it is expected that the user has some basic knowledge of the implemented factorization algorithms. This guide is not intended to be a introduction to basic number theory or integer factorization techniques.

1 Overview of the TIFA package

1.1 About the TIFA library

TIFA is an acronym standing for "Tools for Integer FActorisation". As its (utterly unoriginal) name implies TIFA is an open source library for integer factorization. Its goal is to provide portable and reasonably fast implementations for several algorithms, with a particular emphasis on the factorization of small to medium-size composites, typically from 40 bits to about 200 bits.

Although it obviously won't break any record by itself, TIFA may be a good companion to more ambitious factorization attempts such as a distributed implementation of the Number Field Sieve, where it could be used to factor the numerous smaller-sized by-products.

1.2 Content of the TIFA package

Actually, TIFA is a little bit more than a library *per se*. The TIFA package supplies:

- a C99 library providing implementations for the following factorization algorithms:
  - CFRAC (Continued FRACtion factorization)
  - ECM (The Elliptic Curve Method)
  - FERMAT (McKee's "fast" variant of Fermat's algorithm)
  - SIQS (Self-Initializing Quadratic Sieve)
  - SQUFOF (SQUare FOrm Factorization)
  - TDIV (the naive trial-division method :-);

- a set of stand-alone factorization programs for each algorithm implemented:
  - cfrac_factors
  - ecm_factors
  - fermat_factors
  - siqs_factors
  - squfof_factors
  - tdiv_factors;
- a set of Perl 5 scripts wrappers and launchers;
- a basic benchmarking framework written in Perl 5 used to assess the performance of TIFA's implementations.

Each of these items will be explained in more details in the next few sections.

2 Obtaining and installing the TIFA package

2.1 Obtaining TIFA

The source code of the TIFA package is available as a compressed tarball at:

http://www.lix.polytechnique.fr/Labo/Jerome.Milan/tifa/download.html

Be warned that the development versions may be sometimes highly experimental, so don't be surprised should these crashes more often than not.

2.2 Installation requirements

The TIFA library relies extensively on the GNU Multi Precision library (GMP) version 4.2 or (preferably) higher, which is downloadable from:

http://gmplib.org/

The TIFA library will not build if GMP is not available on your system.

Quite a few Perl scripts are distributed with the library, so of course, a distribution of Perl is required. TIFA has been developed with a distribution of Perl 5.8, but chances are that older versions can adequately run TIFA's scripts.

Moreover, to correctly use the scripts provided in the `./tools` directory, you will also need the GMP Perl modules. There are available as part of the GMP distribution but be aware that they are not built per default, so if you did not set up the GMP library yourself, chances are that these modules have not been generated. In this case, you'll have to head over to http://gmplib.org/ to download a whole new distribution and generate the Perl modules. Instructions for building the modules are given in GMP's `demos/perl/INSTALL` file. Make sure that the distribution you download matches the one installed on your system, otherwise the Perl modules could fail to compile.

Documentation is directly embedded in the code and extracted with the Doxygen tool, available from:

http://www.stack.nl/~dimitri/doxygen

The TIFA code matches the C99 standard so you'll obviously need a somewhat C99-compliant compiler (knowing that there's not a lot of fully C99-compliant compilers!). TIFA has been developed and tested with the GCC C compiler version 3.3, 4.0, 4.1 and 4.2, but older versions should probably work too.

TIFA does not use the autotools (autoconf, autoheader, automake, etc.) as its build system but relies on the more modern, Python-based SCons build tool. You will consequently need the SCons tool version 2.0 or higher available at:

http://www.scons.org

to compile the TIFA library. Note that you'll also need a distribution of Python version 2.4 or later. Pythoncan be obtained at:

http://www.python.org/

TIFA was essentially developed under Mac OS X and tested under both Mac OS X and GNU/Linux (mainly Fedora). It should compile and run "as is", or at worst with minimal modifications, under most of the Unix[1] or unix-like platforms. It should be stressed that no particular effort was made to ensure compatibility with non-unix systems (such as versions of the widespread Windows operating system) although TIFA should be fairly easy to adapt should there be a need.

2.3 Compiling and installing TIFA

TIFA uses the Python-based SCons build tool in place of the more traditional autotools suite. Provided that you have the SCons tool installed and a relatively recent distribution of Python (2.4 or higher), building the TIFA library should be as easy as, if not easier than, with the autotools. Here are the basic steps to follow:

1. Edit the Python file `BuildOptions.py` in TIFA's top directory. This file describes the parameters used during the compilation and installation tasks, so you should adapt it to your set-up or default values will be used, which is certainly not what you want. The provided `BuildOptions.py` is extensively commented so it should not be difficult to adapt it to suit your needs. For a short overview of all of the relevant parameters, type `scons -h`.

While the `BuildOptions.py` file is actually interpreted as a Python script, we recommand to keep its syntax very simple. In other words, do not try to tap into Python's expressive power in that file. Also, be aware that the environment variables are not interpolated within this file. For example, setting the `PREFIX` variable to `'$HOME'` will not give the expected result. You should actually do the interpolation yourself and write `PREFIX='/home/my_user_name'` instead.

[1] Unix® is a Registered Trademark of the Open Group.

2. Compile the library and its programs by invoking SCons without argument:

```
prompt> scons
```

   Note that for the time being, only a static version of the library is generated. Later releases of the library will probably provide an option to build both static and shared versions.

   Optionally, you can also generate the Doxygen documentation:

```
prompt> scons doc
```

3. Finally, install the library (and, if applicable, the generated documentation):

```
prompt> scons install
prompt> scons doc-install
```

For most users, the above procedure should probably be all they need to know. However, several other SCons commands are available to allow for a finner control on what gets build. The table 1 lists all of these commands, together with brief explanations.

`scons`	Build the TIFA library and its associated programs
`scons doc`	Generate the Doxygen documentation
`scons tests`	Build the test programs (even though they are not really useful as is...)
`scons install`	Install the library and its programs, the Perl modules and scripts, the man pages but not the Doxygen documentation
`scons install-doc`	Install the Doxygen documentation
`scons -c`	Clean everything
`scons -c doc`	Clean the Doxygen documentation
`scons -c tests`	Clean the test programs
`scons dist`	Create a tgz archive for distribution. Note that due to current limitations of SCons, the generated tgz archive will expand in the current directory instead of creating a `tifa-X.X.X` directory and expanding the archive there.

continued on next page ...

... continued from previous page

`scons install-lib`	Install the library only
`scons install-bin`	Install the factorization programs only
`scons install-script`	Install the Perl scripts only
`scons install-conf`	Install the Perl scripts' configuration files only
`scons install-perlmod`	Install the TIFA Perl modules only
`scons install-man`	Install the TIFA Perl modules' man pages only

Table 1: Available SCons commands

As already mentioned, the installation process depends on the variables defined in `BuildOptions.py`. Consequently:

- the TIFA library `libtifa.a` is installed in `LIBDIR`;
- TIFA's header files are installed in `INCLUDEDIR`. Quite a few headers are installed, so it is advised to put them all in a dedicated folder such as `/usr/include/tifa` or `~/include/tifa`;
- the four factorization programs (cfrac_factors, qs_factors, siqs_factors and squfof_factors) are installed in `BINDIR`;
- Perl scripts are installed in `SCRIPTDIR`;
- configuration files for the Perl scripts are installed in `CONFDIR`;
- Perl modules are installed in `PERLMODDIR`;
- man pages for the Perl modules are installed in `MANDIR`;
- the Doxygen documentation is installed in `DOCDIR`.

For more information on SCons build options, type `scons -h` on the command line.

3 The TIFA base library

As stated previously, the TIFA "library" is more than just a library since it also offers stand-alone programs and a wealth of Perl 5 scripts and modules. This section however will deal exclusively with the library part of the TIFA package, which we shall call the "base library". Now, since this document is written with the end user in mind, we will leave aside any discussion about TIFA's internals, focusing instead on pragmatic items more likely to be of interest to a user. We refer the reader to the Doxygen documentation for a more thourough description of TIFA's internals.

3.1 Main TIFA types

The user willing to use the factorization functions provided by TIFA only needs to know a few types, namely the `<type>_array_t` and the `<algo>_params_t` types.

This section just gives the minimal information to get started. Please refer to the Doxygen documentation for a complete description of the data types available and the functions used to handle them.

3.1.1 The `type_array_t` types

The `<type>_array_t` structures define a special kind of arrays which know their current lengths and their allocated memory spaces. These structures are all very similar – the only true difference being the type of the data they can contain. The main `<type>_array_t` structures are `mpz_array_t` (to hold GMP's `mpz_t` large integers) and `uint32_array_t` (to hold the native C99 `uint32_t` integers).

The `<type>_array_t` structures are defined as:

```
struct struct_<type>_array_t {
  uint32_t alloced;
  uint32_t length;
  <type>*  data;
};
typedef struct struct_<type>_array_t <type>_array_t;
```

where:

- `alloced` is the memory space allocated for this array's `data` field, given as a multiple of `sizeof(<type>)`. This is the maximum number of `<type>` that the array can accommodate. This allocated memory will be dynamically increased if needed.
- `length` gives the current number of `type` hold in the array pointed by the structure's `data` field.
- `data` is an array of `<type>` whose size is given by the `alloced` field.

Two functions are provided to create and destroy `<type>_array_t` structures.

`<type>_array_t* alloc_<type>_array(uint32_t length)`

Allocates a new <type>_array_t structure with room for length <type> items and returns a pointer to this newly allocated structure.

void free_byte_array(byte_array_t* array)

Frees a <type>_array_t and associated memory.

3.1.2 The algo_params_t types

The <algo>_params_t structures defines the algorithm-dependant parameters to use to perform a given factorization. To each <algo>_params_t structure is associated a set_<algo>_params_to_default function setting the parameters's values to the default ones, which often gives nearly optimal values. Again, we refer the reader to the Doxygen documentation for a full description.

<algo> can be one of the following:

- cfrac – the continued fraction factorization
- ecm – the elliptic curve method
- fermat – McKee's variant of the venerable Fermat algorithm
- siqs – the self-initializing quadratic sieve
- squfof – the square form factorisation
- tdiv – the naive trial division

3.2 Factorization functions

TIFA provides several factoring modes defined by the factoring_mode_t type. A factoring mode can have one the the following values:

- SINGLE_RUN – perform only a single run of the factorization algorithm.
- FIND_SOME_FACTORS – run the factorization algorithm until either some factors are found or the abort limit (defined on a per-algorithm basis) is reached.
- FIND_SOME_COPRIME_FACTORS – run the factorization algorithm until either some coprime factors are found or the abort limit (defined on a per-algorithm basis) is reached.
- FIND_SOME_PRIME_FACTORS – run the factorization algorithm until either some prime factors are found or the abort limit (defined on a per-algorithm basis) is reached.
- FIND_COMPLETE_FACTORIZATION – run the factorization algorithm until either the complete factorization (as a product of prime numbers) is found or the abort limit (defined on a per-algorithm basis) is reached.

The specific factorization functions have the following prototype :

```
ecode_t
<algo> (
    mpz_array_t *const factors,
    uint32_array_t *const multis,
    const mpz_t n,
    const <algo_name>_params_t *const params,
    const factoring_mode_t mode
)
```

These functions attempt to factor the non perfect square integer `n` with the `<algo>` algorithm, using the set of parameters given by `params` and the factoring mode given by `mode`. The found factors are then stored in `factors`. Additionally, if the factoring mode used is set to `FIND_COMPLETE_FACTORIZATION`, the factors's multiplicities are stored in the array `multis`. If the factoring mode used is different from `FIND_COMPLETE_FACTORIZATION`, `multis` is allowed to be the `NULL` pointer. Otherwise, using a `NULL` pointer will lead to a fatal error.

These functions return an exit code, of type `ecode_t`, whose possible values are detailled in the Doxygen documentation.

The previous functions let you choose a given factorization algorithm. Alternatively, the function `tifa_factor` will automatically choose the most suitable algorithm.

```
ecode_t
tifa_factor (
    mpz_array_t *const factors,
    uint32_array_t *const multis,
    const mpz_t n,
    const factoring_mode_t mode
)
```

Example 1. *Try to factor a number given as an argument on the command line.*

In this (unsafe) toy-example, we suppose that the number to factor is not a prime nor a prime power.

```
#include <stdlib.h>
#include <stdio.h>
#include <gmp.h>
#include <tifa.h>

int main(int argc, char **argv) {
      // Read the number to factor given on
      // the command line.
    mpz_t n;
    mpz_init_set_str(n, argv[1], 10);

      // Note that the allocated memory for the factors
      // array will be dynamically increased if needed.
    mpz_array_t* factors = alloc_mpz_array(8);
```

```
    // We cast to (void) since we won't handle the
    // error code in this basic example.
  (void)tifa_factor(factors, NULL, n, SINGLE_RUN);

    // Print the found factors of n.
  for (int i = 0; i < factors->length; i++) {
      gmp_printf("%Zd\n", factors->data[i]);
  }
    // Clean up and turn off the lights.
  free_mpz_array(factors);
  mpz_clear(n);

  return 0;
}
```

3.3 Changing the list of precomputed primes: the `genprimes.pl` script

Most of the factorization algorithms available in the TIFA library belong to a family of algorithms known as "congruences of squares" methods. Basically speaking, the idea is to factor a lot of smaller numbers on a "factor base" and to use these easier-to-find decompositions to deduce a factor of the number. TIFA uses a list of precomputed primes declared in `lib/data/include/first_primes.h` and defined in `lib/data/src/first_primes.c`. By default, a list of 65536 primes is used, which should be enough for any small to medium-sized integer factorizations. If, for whatever reason, you want to change this precomputed list (for example to reduce the memory footprint of the library) the Perl 5 script `lib/data/scripts/genprimes.pl` can be used. This script generates in the current directory a C header file and a C source file that can then be used to replace the original `first_primes.h` and `first_primes.c` files.

Example 2. *Generate header and source files for 8192 primes.*

The following command would create in the current directory the files `primes.h` and `primes.c` by generating a list of the 8192 smallest prime numbers.

```
prompt> ./genprimes.pl --out primes --nprimes 8192
```

For more information, type `./genprimes.pl --help` on the command line. Alternatively, the complete help page of the `genprimes.pl` script is given in appendix A.

4 Utilities provided in the TIFA package

4.1 Factoring programs

Together with the `libtifa` library, a set of stand-alone factorization programs are available:

- cfrac_factors

- ecm_factors
- fermat_factors
- siqs_factors
- squfof_factors
- tdiv_factors

4.2 The `factoring.pl` script

The previous programs can be used on the command-line but their usage can quickly become cumbersome when manually setting algorithm-dependant parameters. To make these programs easier to use, the `tifa/tools/scripts/factoring.pl` Perl script can be invoked.

Example 3. *Factor a number with CFRAC using "optimal" parameters*

```
factorize.pl --exe=cfrac --use_defaults 31418716710282142372441
```

Example 4. *Factor a number with SIQS using custom parameter values.*

```
factorize.pl --exe=siqs --sieve_half_width=200000                        \
             --nprimes_in_factor_base=256 --nprimes_tdiv_smooth_nb=256 \
             --nrelations=32 --linalg_method=0 --use_large_primes      \
             --nprimes_tdiv=256 23283795980376989165117
```

Since it can quickly become annoying to type all the options directly on the command line, `factorize.pl` can optionally read them in a configuration file with a `"<option> = <value>"` kind of syntax. Refer to the provided `factorize.conf` file for a detailed example of a configuration file.

Example 5. *Use the* `factorize.pl` *script with a configuration file*

```
factorize.pl --conf=factorize.conf 407052622923835557569 57
```

You can also input a whole list of numbers to factor by writing them on a file (one number on each line – comments beginning by # are allowed). In this case, you should also provide an output directory where the traces of the C program executions will be saved.

Example 6. *Factor with SIQS a list of numbers given in a file*

```
factorize.pl --exe=siqs --use_defaults --number_file=numbers.txt \
             --outdir=./traces
```

Refer to appendix B for the partial help page of the `factorize.pl` script.

5 Benchmark framework of the TIFA library

A very basic benchmark framework has been developed in Perl to assess the performance of the implemented algorithms. It is basically composed of three stages:

- Macro-generation of the launching script according to various parameters
- Extraction of results in formated text files
- Graphical representation using Gnuplot.

The next section describes the Perl scripts involved in those processes.

5.1 Perl scripts

5.1.1 `tifa/tools/scripts/benchmarker.pl`

This script macro-generate another Perl script aimed at benchmarking TIFA's factoring programs. The generated benchmark script is *not* automatically executed and should therefore be launched manually.

`benchmark.pl` takes as input a configuration file listing the program to bench and all its relevant parameters, and another file listing the arguments the program will be benched with.

Type `./benchmarker.pl --help` in the `tifa/tools/scripts/` directory or refer to appendix C for more information.

5.1.2 `tifa/tools/scripts/extractres.pl`

As its name suggests, `extractres.pl` extracts timing information from trace files to generate a complete result files.

Type `./extractres.pl --help` in the `tifa/tools/scripts/` directory or refer to appendix D for more information.

5.1.3 `tifa/tools/scripts/plotmaker.pl`

The `plotmaker.pl` script takes as input a result file from a script generated by `benchmarker.pl` and creates several plots according to the different parameters involved during the benchmarks (with can results in a substantial amount of graphs).

For the time being, only Gnuplot (`http://www.gnuplot.info/`) is supported.

Type `./plotmaker.pl --help` in the `tifa/tools/scripts/` directory or refer to appendix E for more information.

6 The TIFA Perl modules

As seen in the previous sections, TIFA comes with several Perl scripts. These scripts uses special TIFA Perl modules which can be found in the `tifa/tools/scripts/Tifa` directory.

In this section, we merely list the available modules and give a very brief description. Each of these modules are described in its own unix man page built during the build process. Once the TIFA library is build and installed, information about a particular module can be accessed using a command similar to '`man Tifa::<module_name>`'.

6.1 `tifa/tools/scripts/Tifa/Bencher.pm`

Implements a general purpose benchmark framework and is used by the `benchmarker.pl` script to benchmark the various factorization programs for a wide selection of parameters values.

Type '`man Tifa::Bencher`' or refer to appendix F for more information.

6.2 `tifa/tools/scripts/Tifa/DataDescriptor.pm`

Writes and reads data description/entries in a file according to a TIFA specific text file format.

Type '`man Tifa::DataDescriptor`' or to appendix G for more information.

6.3 `tifa/tools/scripts/Tifa/FormatConverter.pm`

A mere wrapper for image format conversion utilities.

Type '`man Tifa::FormatConverter`' or to appendix H for more information.

6.4 `tifa/tools/scripts/Tifa/GnuPlotter.pm`

Generates 2D plots from 3D data with Gnuplot.

Type '`man Tifa::GnuPlotter`' or to appendix I for more information.

6.5 `tifa/tools/scripts/Tifa/NumberGenerator.pm`

Generates prime or composite integers using the GMP Perl module.

Type '`man Tifa::NumberGenerator`' or to appendix J for more information.

6.6 `tifa/tools/scripts/Tifa/Program.pm`

Provides an abstraction of a "launchable" and "benchmarkable" command line program.

Type '`man Tifa::Program`' or to appendix K for more information.

6.7 `tifa/tools/scripts/Tifa/ProgramRepository.pm`

A repository of all available (factoring) Tifa::Program's.

Type '`man Tifa::ProgramRepository`' or to appendix L for more information.

6.8 `tifa/tools/scripts/Tifa/Rotor.pm`

Implements an odometer-like counter that can be used to generate all possible n-tuples from n sets of values.

Type '`man Tifa::Rotor`' or to appendix M for more information.

6.9 `tifa/tools/scripts/Tifa/SimpleConfigReader.pm`

A truly minimalist configuration file reader used by TIFA's perl scripts.

Type '`man Tifa::SimpleConfigReader`' or to appendix N for more information.

Appendices

A Help page of the `genprimes.pl` script

```
genprimes.pl - Outputs list of primes as C header and source files
-----------------------------------------------------------------

Usage:
------

   genprimes.pl [--nprimes <number_of_primes>] [--help]
                --out <output_prefix>

This script is part of the TIFA (Tools for Integer FActorization) library.
It generates a list of prime numbers and declares them as a uint32_t array
in a C header and source file.

General parameters/options:
---------------------------

    --out = s
        Output file prefix. The header file and the C source file names
        are given by appending respectively ".h" and ".c".

    --nprimes = i
        Number of primes to generate.
        Default value: 8192

    --help
        Prints this help message.
```

B Help page of the `factorize.pl` script

factorize.pl - A Perl wrapper for TIFA's factorization programs

```
Usage:
------

   factorize.pl [parameters] [options] <number_to_factor>

or:

   factorize.pl [parameters] [options] --numbers <file> --outdir <directory>

The factorize.pl script wraps the various TIFA factorization programs and offers
a user-friendly command line interface which can be easily extended to support
new factoring program, as far as the new programs are wrapped in a
Tifa::Program object and registered in the Tifa::ProgramRepository module.

This script can be used to factor a single composite integer given as an
argument on the command line or to factor a whole set of composite numbers
given in a text file.

General parameters/options:
---------------------------

  --algo=s?
      (optional, but --exe should then be defined)
      Name of the factorization algorithm to use. Should be one of the
      following value:
         cfrac  (The Continued FRACtion algorithm)
           ecm  (The Elliptic Curve Method)
        fermat  (Fermat's algorithm (McKee's speedup))
          siqs  (The Self-Initializing Quadratic Sieve algorithm)
        squfof  (The SQUare FOrm Factorization algorithm)
          tdiv  (The naive trial division algorithm)
       <empty>  (if empty, infers algorithm from program's default name)

  --exe=s
        (optional, but --algo should then be defined)
        Name of the executable command line program. Note that the default value
        is only used if the algo parameter is set to a known algorithm name
        _and_ the exe parameter is not defined or left empty.
        Default: vary according to algorithm used.
                 Type factorize.pl --help <algo_name> for more information.

  --cmd
      Print the command line used to call the underlying C program and exit.
```

```
  --conf=s
      (optional, on command-line only)
      Name of the configuration file listing the values of the factorization
      algorithms' parameters written as: "<option_name> = <value>".
      Type factorize.pl --help config for information about the configuration
      file format.

  --numbers=s
      (optional, on command-line only)
      Do not read the number to factorize on the command line but read a
      set of numbers given in a text file. Numbers can be separated by any
      kind of non digit characters. Comments beginning by # and blank lines
      are allowed in the file. If this option is used the number passed as
      argument -if any- will be discarded.

  --outdir=s
      (mandatory if --numbers is used, on command-line only)
      Name of the directory to contain the various execution traces. Only
      used if the --numbers option is used. If the number to factor is
      directly given on the command line, no trace will be saved.

  --help=s?
      (optional, on command-line only)
      Without argument, prints this help message.
      With an algorithm name, prints the help message related to the use of
      this script with this particular algorithm.

Algorithm-specific parameters/options:
--------------------------------------

Type factorize.pl --help <algo_name> (where <algo_name> can take one of the
value mentionned in the above description of the --algo option) for more
information about algorithm specific parameters and options.
```

C Help page of the benchmarker.pl script

```
benchmarker.pl - A (mostly) generic benchmark script
----------------------------------------------------

Usage:
------

 benchmarker.pl [--conf   <config_file>]      [--macro <output_macro>]
                [--outdir <output_directory>] [--args  <argument_file>]
                [--args   <output_directory>] [--help]

The benchmarker.pl script provides an (almost!) generic framework for
benchmarking programs. It takes as input a configuration file listing the
program to bench and all its relevant parameters, and another file listing the
arguments the program will be benched with. If no argument file is provided, the
benchmarker.pl script will make one by generating composite numbers (a feature
clearly inherited from its non-generic, integer factorization-oriented days).

In any case, this script will create (but NOT execute!) the "real" benchmarking
Perl script which will (once manually launched) execute the program to benchmark
for the whole range of parameters given in the configuration file.

Note that this script can be used to bench any kind of programs as long as
they are wrapped in a Tifa::Program object and declared in the
Tifa::ProgramRepository module. Users willing to extend the list of available
programs should then have a closer look at these two modules.

Usage of this script can be hard to grasp, so the easiest way to get used to
it is probably to just read the provided benchmark.conf configuration file in
addition to this manual.

General parameters/options:
---------------------------

  --args=s
      (optional)
      Name of the argument file listing the arguments to be passed to the
      program to benchmark. If none are provided, composite numbers will be
      generated and used as argument (which is fine with TIFA's factorization
      programs). The number generation can be controlled by the
      bit_length_of_n and nprime_factors_in_n fields in the configuration
      file. See the provided benchmark.conf configuration file for more
      information.
      Default value: none
```

```
  --conf=s
      (optional, default value used if none provided)
      Name of the configuration file listing the values of the parameters
      relevant to the program to benchmark. This program will be benchmarked
      using each possible combination of the parameter values provided in this
      file. The configuration file syntax should match the one defined by the
      Tifa::SimpleConfigReader module. See the man page associated to that
      module for more information. Reading the provided benchmark.conf
      configuration file is also highly advised.
      Default value: benchmarker.conf

  --macro=s
      (optional, default value used if none provided)
      Name of the "real" benchmarking perl script that will be generated by
      benchmarker.pl. Note that this generated script has to be launched
      manually to proceed to the benchmarks.
      Default value: bench.pl

  --outdir=s
      (optional, default value used if none provided)
      Name of the directory where bench results and trace files will be saved.
      Default value: bench_results

  --mode=s
      (optional, value in configuration file used if none provided)
      Program's mode to use.

  --filter=s
      (optional, value in configuration file used if none provided)
      Impose a condition on the set of parameters. A run will be performed only
      if this expression evaluates to non zero. This condition should be written
      as Perl 5.8 code.

  --help
      Prints this short manual.

See also:
---------

  The documentation for the Tifa::Program and Tifa::ProgramRepository modules.
```

D Help page of the `extractres.pl` script

```
Usage:
------

  extractres.pl --in <input_file> --out <output_file>
                --tracedir <trace_directory> --nfactors <nfactors>
                [--help]

The extractres.pl script is used after executing a benchmark macro created by
the benchmarker.pl script. It reads the original result file (in the
Tifa::DataDescriptor format) and the trace files generated during benchmarks to
create a more complete result file by adding extra information found in the
trace files. This script is not genering and is tailored to be used if and only
if a TIFA program has been benchmarked since it expects to find specific
information in the trace files.

General parameters/options:
---------------------------

  --in=s
      (mandatory)
      Name of the input result file as generated by the benchmarking script
      created by benchmarker.pl.

  --out=s
      (mandatory)
      Name of the output result file. This file will contain information
      from the original input file and extra information read in the
      trace files generated during the benchmarks.

  --tracedir=s
      (mandatory)
      Name of the directory containing the trace files.

  --nfactors=i
      (optional, meaningless value used if none provided)
      Number of prime factors composing the integers to factor that were used
      in the benchmarks. If not provided, -1 will be used to indicate that the
      real value is unknown.

See also:
---------

  The documentation for the benchmarker.pl script and the Tifa::DataDescriptor
  module.
```

E Help page of the plotres.pl script

```
plotmaker.pl - Automatic generation of plots from result files
--------------------------------------------------------------

Usage:
------

   plotmaker.pl --in <infile> --outdir <output dir> [--format <format>]
                [--program <plotting_program>] [--conf <config_file>]
                [--prefix <prefix>] [--help]

The plotmaker.pl script will generate plotting macros and plots from a result
file created by a benchmark perl script (as generated by the benchmarker.pl
script) and extracted using extractres.pl.

General parameters/options:
---------------------------

  --in=s
      (mandatory)
      Input file containing the benchmark results in a format understandable
      by the Tifa::DataDescriptor module. Scripts generated by
      extractres.pl follow that convention. See help for extractres.pl and
      the Tifa::DataDescriptor module for more information.

  --program=s
      (optional, default value used it none provided)
      Plot program to use. It actually defines which Tifa plotting module will
      be used to create the plots.
      Available programs:
          gnuplot : Create gnuplot macros and plots using Tifa::GnuPlotter
      Default: gnuplot

  --prefix=s
      (optional, default value used it none provided)
      Prefix used in naming the generated files:
         Data  : <prefix>data_xxxx.txt
         Macros: <prefix>macro_xxxx.macro_format
         Plots : <prefix>plot_xxxx.plot_format
      Default: "" (empty string)

  --conf=s
      (optional, default value used it none provided)
      Name of the configuration file describing the plots to generate.
      This configuration file should be written in pure Perl as it will be
```

```
    directly interpreted "as is". Each new plot description should be pushed
    in the (already defined) @pdesc_array array.
    Default: plotmaker.conf

    A plot description is a structure (in the Class:Struct sense) that
    describes a series of plot. Its member data are:

        x      : The expression giving the abscissa of the plots
        y      : The expression giving the ordinate of the plots
        z      : The variable name of the parameter of the plots
        cond   : Extra conditionnal expression acting as a filter: the
                 plots will be generated considering only the data points
                 verifying this condition
        params : List of variable names that has to be considered as fixed
                 parameters for a given plot. One plot is generated for each
                 combination of values of the variables named in the params
                 array
        graph_title :
        x_axis_title:
        y_axis_title: self explanatory

    The provided plotmaker.conf configuration file gives an example of the
    syntax used.

--outdir=s
    (mandatory)
    Name of the output directory.
    Created files will be arranged in the following fashion:

    <output_dir>/<program>/data/:
        Contains the <program> data files for each individual plot.
    <output_dir>/<program>/macro/:
        Contains the <program> macros for each individual plot. There is
        a macro_xxxx.ext in this directory for each plot_xxxx.txt datafile
        in <output_dir>/<program>/data/, where x is a decimal digit.
        The macro extension 'ext' varies depending on which <program> is used.
    <output_dir>/<program>/<format>/:
        Depending of the output file format selected, contains the
        resulting plot_xxxx.format files, as created by the <program>
        macros macro_xxxx.ext in <output_dir>/<program>/macro/.

--format=s
    (optional, default value used it none provided)
    File format for the various created plots.
    Available formats:
          gnuplot program: "eps", "pdf", "png"
    Default: pdf
```

```
--help
    Prints this help...
```

F Man page of the Tifa::Bencher.pm Perl module

```
Tifa::Bencher(1)              TIFA Documentation              Tifa::Bencher(1)

NAME
       Tifa::Bencher - A general purpose benchmark module

SYNOPSIS
         use Tifa::Bencher;
         $program = new Tifa::Bencher();

REQUIRE
       Perl 5.006002, Carp, Exporter, Tifa::ProgramRepository and Tifa::Rotor.

SUMMARY
       The Tifa::Bencher module implements a general purpose benchmark
       framework.  Given a program and a set of parameters, it can generate a
       benchmark perl script using every combination of the parameters values.

DESCRIPTION
       The Tifa::Bencher module implements a general purpose benchmark
       framework and is used in the TIFA library by the benchmarker.pl script
       to benchmark the various factorization programs for a wide selection of
       parameters values.

       A Tifa::Bencher object takes its parameters (including the name of the
       program to benchmark) from a hashtable mapping each parameter name to
       its value or to an array of values (to perform benchmarks for different
       values of a given parameter). It can then generate a Perl script
       listing all of the commands to execute, that is to say, all the
       invocations of the program to benchmark for every possible combination
       of the parameter values. The generated script should then be launched
       manually to proceed to the benchmarks.

       Note that the key 'exe' must appear in this hashtable since its
       associated value(s) give(s) the name(s) of the program(s) to benchmark.
       Additionally, the program to benchmark should be wrapped in a
       Tifa::Program object and registered in the Tifa::ProgramRepository
       module to be useable (this is the case for all of the TIFA's
       factorization programs).

       Data members

       A Bencher object is defined by the following attributes:

           $args_file    : Name of the argument file.

           $output_dir   : Name of the ouput directory where bench results will
```

```
                    be saved

    $filter       : Perl-like conditional expression used to decide
                    whether or not the program should be benchmarked
                    with a particular combination of parameters.

    %param_values : Hashtable mapping each parameter name to its value
                    or to an array of values. The 'exe' key is mandatory
                    since its associated value gives the name of the
                    program to benchmark.

Available methods

    new()
    set_output_dir($outdir)
    get_output_dir()
    set_args_file($filename)
    get_args_file()
    set_param_values($hash_ref)
    get_param_values()
    set_filter($expression)
    get_filter()
    create_bench_macro($outfile)

Methods description

    new()
        Basic constructor allocating a Tifa::Bencher object.

    set_output_dir($outdir)
        Sets the name of the output directory. Trace files will be saved
        under $outdir/traces as trace_xxxx.txt with xxxx being a unique
        integer identifier.

    get_output_dir()
        Returns the name of the ouput directory.

    set_args_file($filename)
        Sets the name of the argument file.

    get_args_file()
        Returns the name of the argument file.

    set_param_values($hash_ref)
        Sets the values of the parameters relevant to the program to
        bench. $hash_ref is a reference to a hashtable mapping each
        parameter name to its value or to an array of values. In
        particular, the name(s) and the mode(s) (see documentation for
```

```
        Tifa::Program) of the program to benchmark should be given in
        this hashtable.

    get_param_values()
        Returns a reference to a hashtable mapping each parameter
        name to its value or to an array of values.

    set_filter($expression)
        Sets the conditional expression used to decide whether or not
        the program should be benchmarked with a particular combination
        of parameters. $expression should follow Perl's syntax and use
        parameter names as given by 'set_param_values'. In addition to
        these parameter names, the parameters 'algo', 'exe' and 'mode'
        are allowed.

    get_filter()
        Returns the current filter as a string.

    create_bench_macro($outfile)
        Creates a Perl script named $outfile containing all the program
        invocations to benchmark. The created script is not executed and
        should consequently be launched manually to proceed to the real
        benchmarks.

EXAMPLE
    The best example of how to use the Tifa::Bencher module is given by the
    benchmarker.pl script, so it is strongly advised to have a look at
    benchmarker.pl's documentation.

    Here's another toy example of the Tifa::bencher module used to
    benchmark TIFA's implementation of the SQUFOF algorithm for several
    numbers of trial divisions. Note that this implementation is registered
    in the Tifa::ProgramRepository module with the name 'squfof_program'.

      $bencher = new Tifa::Bencher;

      #
      # Create an argument file containing some numbers to factors.
      #
      $argfile = "numbers.txt";

      open(OUT, ">$argfile") or die("Cannot open $argfile");
      print OUT "816379";
      print OUT "123465";
      close(OUT);

      #
      # Parameter hashtable
```

```
#
%params = (
    'exe'          => './squfof_program',
    'nprimes_tdiv' => [15, 30, 45]
);

$bencher->set_args_file($argfile);
$bencher->set_output_dir("benchs");
$bencher->set_param_values(\%params);
$bencher->set_filter("nprimes_tdiv >= 30");

$bencher->create_bench_macro("test.pl");

#
# The generated test.pl script will execute the following commands:
#
#     ./squfof_program 30 816379 > benchs/traces/trace_0.txt
#     ./squfof_program 45 816379 > benchs/traces/trace_1.txt
#     ./squfof_program 30 123465 > benchs/traces/trace_2.txt
#     ./squfof_program 45 123465 > benchs/traces/trace_3.txt
#
# Note that the commands:
#
#     ./squfof_program 15 816379 > benchs/traces/...
#     ./squfof_program 15 123465 > benchs/traces/...
#
# will not be executed since the conditional expression passed to the
# function set_filter is not satisfied for these parameter values.
#
```

EXPORT

No functions are exported from this package by default.

SEE ALSO

The benchmarker.pl script and the Tifa::Program and Tifa::ProgramRepository modules.

G Man page of the Tifa::DataDescriptor.pm Perl module

```
Tifa::DataDescriptor(1)       TIFA Documentation       Tifa::DataDescriptor(1)

NAME
       Tifa::DataDescriptor - Write and read data description/entries in a
       file.

SYNOPSIS
         use Tifa::DataDescriptor;
         $descriptor = new Tifa::DataDescriptor();

REQUIRE
       Perl 5.006002, Carp and Exporter.

DESCRIPTION
       This Perl module is part of the TIFA (Tools for Integer FActorization)
       library.  Its goal is to provide a consistent interface to document and
       write a text data file and to retrieve these documentation and data
       entries from a data file in order to process it.

       Available methods

       This module provides the following methods:

           Tifa::DataDescriptor()
           set_comment_style($style)
           set_field_separator($separator)
           get_field_separator()
           add_field_description($field_name, $description)
           get_field_description($field_name)
           get_all_field_names()
           write_descriptions($file_handle)
           load_descriptions($file_handle)
           write_data_entry($file_handle, @data_values)
           read_next_data_entry($file_handle)

       Methods description

           Tifa::DataDescriptor()
               Basic constructor allocating a Tifa::DataDescriptor object.

           set_comment_style($style)
               Sets the comment style used for the data description.
               Allowed values: "Perl", "C" and "C++".
               Default value : "Perl".

           set_field_separator($separator)
```

```
        Sets the motif used to separate the data fields.
        Default value: ":".

    get_field_separator()
        Returns the motif used to separate the data fields.

    add_field_description($field_name, $description)
        Adds in the data description a field $field_name with its
        description given by $description.

    get_field_description($field_name)
        Gets the description of the field $field_name.

    get_all_field_names()
        Returns an array containing all of the data fields' names.

    write_descriptions($file_handle)
        Writes the data description to an already opened file given
        by $file_handle.

    load_descriptions($file_handle)
        Loads the data description from the opened file referenced
        by $file_handle.

    write_data_entry($file_handle, @data_values)
        Writes a data entry with values given by the array @data_values
        in the opened file referenced by $file_hanfle.

    read_next_data_entry($file_handle)
        Reads the next data entry in the file referenced by $file_handle
        and returns it as a hashtable mapping the field names to their
        values. If no entry is found, returns an empty hashtable.

EXAMPLES
    Example 1: write data description to a file

    This following code snippet gives an example of how to use the
    Tifa::DataDescriptor module to embed data description in a file.

        my $descr    = new Tifa::DataDescriptor();
        my $filename = "os_mascots.txt";
        open(my $handle, ">$filename") or die("Cannot open $filename\n");
        #
        # Fill the descriptor with the data description.
        #
        $descr->set_comment_style("Perl");
        $descr->set_field_separator(":");
        $descr->add_field_description("Name", "Name of mascot");
```

```
    $descr->add_field_description("Species", "Species of mascot");
    $descr->add_field_description("FavOS", "Favorite OS of mascot");
    #
    # Write the data description in the data file.
    #
    $descr->write_descriptions($handle);
    #
    # Write some data entries in the data file.
    #
    my @data = ('Tux', 'penguin', 'Linux');
    $descr->write_data_entry($handle, @data);

    @data = ('Hexley', 'platypus', 'Darwin');
    $descr->write_data_entry($handle, @data);

    @data = ('Beastie', 'daemon', 'FreeBSD');
    $descr->write_data_entry($handle, @data);

    @data = ('Puffy', 'pufferfish', 'OpenBSD');
    $descr->write_data_entry($handle, @data);

    close(my $handle);
```

The following description and entries are written at the current position in the file:

```
    # <DataDescription>
    #
    #    <format> Name:Species:FavOS </format>
    #
    #    <field> Name : Name of mascot </field>
    #    <field> Species : Species of mascot </field>
    #    <field> FavOS : Favorite OS of mascot </field>
    #
    # </DataDescription>

    <data> Tux:penguin:Linux </data>
    <data> Hexley:platypus:Darwin </data>
    <data> Beastie:daemon:FreeBSD </data>
    <data> Puffy:pufferfish:OpenBSD </data>
```

Example 2: read data description in a file

This following code snippet gives an example of how to use the Tifa::DataDescriptor module to retrieve the data description and entries embedded within a file.

```
    my $descr    = new Tifa::DataDescriptor();
```

```
    my $filename = "data.txt";
    open(my $handle, "<$filename") or die("Cannot open $filename\n");
    #
    # Assume that the data file contains comments given in a Perl style.
    #
    $descr->set_comment_style("Perl");
    #
    # Read the data description embedded in the data file.
    #
    $descr->load_descriptions($filename);
    #
    # Read all the data entries and print them on stdout.
    #
    while (%entry = $descr->read_next_data_entry($handle)) {

        foreach my $key (keys %entry) {
            print("\$entry{$key} = $entry{$key}\n");
        }
        print("\n");
    }

    close($handle);
```

EXPORT

No functions are exported from this package by default.

H Man page of the Tifa::FormatConverter.pm Perl module

```
Tifa::FormatConverter(1)      TIFA Documentation      Tifa::FormatConverter(1)

NAME
       Tifa::FormatConverter - A wrapper for format conversion utilities

SYNOPSIS
        use Tifa::FormatConverter;

        my $converter = new Tifa::FormatConverter();
        $converter->convert($from, $to);

REQUIRE
       Perl 5.006002, Carp, Exporter, Tifa::DataDescriptor and the following
       programs: convert (from the ImageMagick suite), ps2pdf, ps2eps, pdf2ps,
       epstopdf, dvips and dvipdf.

DESCRIPTION
       The Tifa::FormatConverter is a mere wrapper for several conversion
       utilities popular in the unix/unix-like world, namely convert (from the
       very powerful ImageMagick suite), ps2pdf, ps2eps, pdf2ps, epstopdf,
       dvips and dvipdf.

       File formats are directly inferred from the extension of the filenames.
       The following extensions/conversions are supported:

        - Pixmap to pixmap/(pseudo-)vectorial (uses convert):

            "png"  to "pdf", "eps", "ps", "jpg", "jpeg", "gif" and "tiff"

            "gif"  to "pdf", "eps", "ps", "jpg","jpeg", "png" and "tiff"

            "tiff" to "pdf", "eps", "ps", "jpg","jpeg", "gif" and "png"

            "jpeg" to "pdf", "eps", "ps", "png", "gif" and "tiff"

            "jpg"  to "pdf", "eps", "ps", "png", "gif" and "tiff"

        - Vectorial to pixmap (uses convert):

            "ps"  to "png", "jpg", "jpeg", "gif" and "tiff"

            "eps" to "png", "jpg", "jpeg", "gif" and "tiff"

            "pdf" to "png", "jpg", "jpeg", "gif" and "tiff"

        - Vectorial to vectorial (uses either ps2eps, ps2pdf, epstopdf,
```

```
   pdf2ps, dvips or dvipdf):

     "ps"  to "eps" and "pdf"

     "eps" to "pdf"

     "pdf" to "ps"

     "dvi" to "ps" and "pdf"

   The Tifa::FormatConverter module does not check whether or not these
   programs are installed on your system. Consequently, one can still use
   this module if some of the programs are not available, provided that
   the desired format conversion does not rely on them.

   Available methods

   This module provides the following methods:

       new()
       convert($from, $to)

   Methods description

       new()
           Basic constructor allocating a Tifa::FormatConverter object.

       convert($from, $to)
           Converts the file $from to a file $to, provided that the format
           conversion is supported. Formats are inferred from the filenames'
           extensions.

EXAMPLE
   Using the Tifa::FormatConverter is completely straightforward. For
   example:

       my $converter = new Tifa::FormatConverter();

       my $eps_file   = "file.eps";
       my $pdf_file   = "file.pdf";
       my $png_file   = "file.png";
       my $jpg_file   = "file.jpg";

       $converter->convert($eps_file, $pdf_file);
       $converter->convert($pdf_file, $png_file);
       $converter->convert($png_file, $jpg_file);

EXPORT
```

No functions are exported from this package by default.

I Man page of the Tifa::GnuPlotter.pm Perl module

```
Tifa::GnuPlotter(1)          TIFA Documentation          Tifa::GnuPlotter(1)

NAME
       Tifa::GnuPlotter - Generate 2D plots from 3D data with gnuplot

SYNOPSIS
         use Tifa::GnuPlotter;
         $reader = new Tifa::GnuPlotter();

REQUIRE
       Perl 5.006002, Carp, Exporter, Tifa::DataDescriptor,
       Tifa::FormatConverter and the gnuplot program.

SUMMARY
       The Tifa::GnuPlotter module facilitates the generation of 2D plots from
       3D data by using the gnuplot program as its underlying engine. However,
       it is not intended to be a generic low-level gnuplot wrapper like
       Chart::Graph for example since it is way too limited for that.

DESCRIPTION
       The Tifa::GnuPlotter module was written during the benchmarking phase
       of the factoring tools of the TIFA library to facilitate the
       automatization of timing plots. Although it was only ment to fulfills
       that very special requirement, it can be helpful in other, similar data
       analysis problems.

       Tifa::GnuPlotter is specifically designed to generate 2D plots from
       3-dimensional data where the extra third dimension is supposed to be a
       varying parameter: the output will be several graphs (actually one for
       each value of the third parameter) drawn together on one plot.

       Generated plots can be in Portable Document Format (PDF), Encapsulated
       PostScript format (EPS) or Portable Network Graphics format (PNG).

       Available methods

       This module provides the following methods:

           new()
           set_gnuplot_program($path_to_gnuplot)
           set_output_format($output_format)
           set_data_filename($filename)
           set_macro_filename($filename)
           set_output_filename($filename)
           set_xvar_description($name, $description)
           set_yvar_description($name, $description)
```

```
    set_zvar_description($name, $description)
    set_xvar_values($values_as_array_ref)
    set_yvar_values($values_as_array_ref)
    set_zvar_values($values_as_array_ref)
    set_xaxis_title($title)
    set_yaxis_title($title)
    set_graph_title($title)
    set_main_label($label)
    set_second_label($label)
    write_data_file()
    generate_macro()
    execute_macro()

Methods description

    new()
        Basic constructor allocating a Tifa::GnuPlotter object.
        Plot output format is by default set to "pdf".

    set_gnuplot_program($path_to_gnuplot)
        Sets the name of the gnuplot executable, including its full path.

    set_output_format($output_format)
        Sets the file format of the generated plots.
        Must be one of "pdf", "eps" or "png".

    set_data_filename($filename)
        Sets the filename of the gnuplot data file to be generated.

    set_macro_filename($filename)
        Sets the filename of the gnuplot macro file to be generated.

    set_output_filename($filename)
        Sets the filename of the gnuplot plot file to be generated.
        If the provided $filename does not end with the proprer file
        extension, the good file extension will be appended according
        to the set output format.

    set_xvar_description($name, $description)
        Sets the name of the x variable together with a short
        description to be used in the generated data file.

    set_yvar_description($name, $description)
        Sets the name of the y variable together with a short
        description to be used in the generated data file.

    set_zvar_description($name, $description)
        Sets the name of the z variable together with a short
```

```
        description to be used in the generated data file.

    set_xvar_values(@x_values)
        Sets the values of the x variables to the ones in the array
        referenced by $values_as_array_ref.

    set_yvar_values(@y_values)
        Sets the values of the x variables to the ones in the array
        referenced by $values_as_array_ref.

    set_zvar_values(@z_values)
        Sets the values of the x variables to the ones in the array
        referenced by $values_as_array_ref.

    set_xaxis_title($title)
        Sets the title of the x axis.

    set_yaxis_title($title)
        Sets the title of the y axis.

    set_graph_title($title)
        Sets the title of the generated graph.

    set_main_label($label)
        Sets the main label that will appear under the graph title.

    set_second_label($label)
        Sets the second label that will appear on the plot.

    write_data_file()
        Writes the gnuplot data file.

    generate_macro()
        Generates the gnuplot macro file.

    execute_macro()
        Executes the generated gnuplot macro file.

EXAMPLE
   The following code snippet shows how to create a 2D plot:

    #
    # The 3-dimensional toy-data...
    #
    my @x_array = (1.0, 2.1, 3.1, 4.0, 1.1, 2.1, 3.0,  3.9);
    my @y_array = (2.1, 3.9, 5.9, 7.8, 3.2, 6.4, 9.4, 12.2);
    my @z_array = (2,   2,     2,   2,   3,   3,   3,    3);
```

```
        $plotter = new Tifa::GnuPlotter();
        #
        # Set the relevant parameters...
        #
        $plotter->set_xvar_description("time", "Time Elapsed (s)");
        $plotter->set_yvar_description("distance", "Travelled Distance (m)");
        $plotter->set_zvar_description("acceleration",
                                       "Theoretical Acceleration");
        #
        # Of course, one should in principle check that all the data
        # arrays have the same length.
        #
        $plotter->set_xvar_values(\@x_array);
        $plotter->set_yvar_values(\@y_array);
        $plotter->set_zvar_values(\@z_array);

        $plotter->set_xaxis_title("Time (s)");
        $plotter->set_yaxis_title("Distance (m)");
        $plotter->set_graph_title("Travelled distance as a function of time");
        $plotter->set_main_label('Projectile: book\nWhen: after the final');
        $plotter->set_second_label('Measured accelerations match theory!');

        $plotter->set_data_filename("flying_book.dat");
        $plotter->set_macro_filename("flying_book.gp");
        $plotter->set_output_filename("flying_book.pdf");

        $plotter->write_data_file();
        $plotter->generate_macro();
        #
        # Executing the flying_book.gp macro will produce a PDF file
        # displaying two graphs drawn on the same plot, one for each
        # value of the "acceleration" parameter.
        #
        $plotter->execute_macro();
```

EXPORT

No functions are exported from this package by default.

J Man page of the Tifa::NumberGenerator.pm Perl module

```
Tifa::NumberGenerator(1)      TIFA Documentation      Tifa::NumberGenerator(1)

NAME
       Tifa::NumberGenerator - Generate prime or composite integers.

SYNOPSIS
         use Tifa::NumberGenerator;
         use GMP::Mpz;
         $generator = new Tifa::NumberGenerator();

REQUIRE
       Perl 5.006002, Carp, Exporter, GMP::Mpz and GMP::Rand.

DESCRIPTION
       This Perl module is part of the TIFA (Tools for Integer FActorization)
       library.  It uses the GNU Multi-Precision Perl modules GMP::Mpz and
       GMP::Rand to generate (possibly random) prime or composite non-square
       numbers.

       Available methods

       This module provides the following methods:

           Tifa::NumberGenerator()
           use_random()
           dont_use_random()
           generate_prime($size)
           generate_composite($size, $nfactors)

       Methods description

           Tifa::NumberGenerator()
               Basic constructor allocating a Tifa::NumberGenerator object.

           use_random()
               Invoke this method to generate numbers randomly.

           dont_use_random()
               Invoke this method to generate numbers deterministically.

           generate_prime($size)
               Returns a prime of $size bits as a GMP::Mpz object.

           generate_composite($size, $nfactors)
               Returns a non-square composite integer of $size bits (obtained
               by multiplying $nfactors prime integers of roughly equal sizes)
```

as a GMP::Mpz object.

A note on deterministic generation

(Pseudo-) random generation can be toggled on and off by using the use_random() and dont_use_random() method.

When a prime is generated deterministically (i.e. when not using a random generator), the smallest prime of the required bit length is returned.

When a composite integer is generated deterministically, the smallest prime numbers of the required bit length are multiplied together to yield the final composite. In this case, each prime number is used only once in the multiplication.

A note on primality

The Tifa::NumberGenerator() uses the GMP:Mpz module to check for the primality of a given number. These tests are actually Miller-Rabin tests of compositeness rather than primality tests strictly speaking. However, "prime" numbers generated by the Tifa::NumberGenerator() have indeed a very good chance of being truly prime, the probability of returning a composite being extremely small.

EXAMPLE

This following code snippet gives an example of how to use the Tifa::NumberGenerator() module.

```
my $generator = new Tifa::NumberGenerator();
#
# Generate numbers randomly with $generator->use_random() or in
# a deterministic fashion with $generator->dont_use_random().
#
$generator->use_random();
#
# Generate a prime of 120 bits.
#
my $prime = $generator->generate_prime(120);
#
# Generate a composite of 120 bits from the multiplication
# of 4 prime numbers of (roughly) equal lengths.
#
my $composite = $generator->generate_composite(120, 4);
```

EXPORT

No functions are exported from this package by default.

K Man page of the Tifa::Program.pm Perl module

```
Tifa::Program(1)              TIFA Documentation              Tifa::Program(1)

NAME
       Tifa::Program - An abstraction of a TIFA command line program

SYNOPSIS
         use Tifa::Program;
         $program = new Tifa::Program();

REQUIRE
       Perl 5.006002, Carp, Class::Struct and Exporter.

SUMMARY
       The Tifa::Program module is a weird module whose only raison d'etre is
       to provide an abstraction of what a launchable and benchmarkable
       command line program is. Most, if not all, of TIFA users do not need to
       known about this module.

DESCRIPTION
       The Tifa::Program module provide an abstraction of what a launchable
       and benchmarkable command line program is. In less cryptic words, its
       goal is to provide other TIFA's scripts and modules with a unified way
       to describe and interact with the implemented factorization programs.
       The motivation for such a module is to be able to add easily new
       (factorization) programs that can be launched or benchmarked by TIFA's
       scripts while containing code changes to a few places only. As a TIFA
       user you probably do not need to known anything about this module
       unless you plan to use TIFA's benchmarking and plotting framework for
       programs other than TIFA's included factorization programs.

       Data members

       A Program object is defined by the following attributes:

           $algo  : Name of the algorithm implemented.

           $descr : Short description of the algorithm.

           $exe   : Name of the command line program that is wrapped.

           $help  : Help message for the command line program that is wrapped.

           %modes : An hashtable mapping mode names to 'modes'. A mode
                    represents a particular form of invocation of the
                    program given by $exe, and is defined by:
```

```
                - A description $descr
                - A list of relevant parameter names
                - A command line template

            For example, TIFA's CFRAC program can be called with
            different numbers of parameter according to whether or
            not we should let the program choose optimal parameter
            values. In this case, we could have a mode where all
            parameters have to be specified and another one where
            only a few of them are needed. (See example in the next
            section.)

    $cur_mode : Current mode name.

    @cur_param_names : List of parameter names relevant to the current
                       mode.

    @cur_param_descrs: List of parameter descriptions relevant to the
                       current mode. The descriptions are given in the
                       same parameter order than in @cur_param_names.

    @cur_param_types : List of parameter types relevant to the current
                       mode. The types are given in the same parameter
                       order than in @cur_param_names.

    $cur_cmdline: Command line template relevant to the current mode.

    $default_mode: Default mode name.

    @all_param_names : Array listing all of the parameter names

    %all_param_to_descrs_hash: Hashtable mapping the parameter names
                               to their descriptions;

    %all_param_to_types_hash : Hashtable mapping the parameter names
                               to their types

Available methods

This module provides the following methods:

    new()
    set_algo($algo)
    get_algo()
    set_descr($descr)
    get_descr()
    set_exe($exe)
    get_exe()
```

```
    set_help($help)
    get_help()
    set_default_mode($mode)
    get_default_mode()
    add_mode($key, $name, $descr,
             \@parnames, \@pardescs, \@partypes, $cmdline)
    set_mode($mode)
    get_mode()
    get_all_param_names()
    get_all_param_to_descrs_hash()
    get_all_param_to_types_hash()
    get_param_names()
    get_param_descrs()
    get_param_types()
    get_cmdline()
    get_mode_descr()
    get_all_modes()
    get_getopt_strings()
    execute(\%hash, $arg, $postcmd)
    make_cmd(\%hash, $arg, $postcmd)

Methods description

    new()
        Basic constructor allocating a Tifa::Program object.

    set_algo($algo)
        Sets the name of the algorithm implemented by the Program
        object to $algo.

    get_algo()
        Gets the name of the algorithm implemented by the Program object.

    set_descr($descr)
        Sets the description of the algorithm implemented to $descr.

    get_descr()
        Gets the description of the algorithm implemented.

    set_exe($exe)
        Sets the name of the command line program to $exe.

    get_exe()
        Gets the name of the command line program.

    set_help($help)
        Sets the help message to $help.
```

```
get_help()
    Gets the help message as a string.

set_default_mode($mode)
    Sets the name of the default mode of the Program object to
    $mode. Exits if mode $mode does not exist.

get_default_mode()
    Gets the name of the default mode of the Program object.

add_mode($key, $name, $descr,
         \@parnames, \@pardescs, \@partypes, $cmdline)
    Adds a mode to the Program object. The mode is identified by:

        - a name $name
        - a description string $descr,
        - a list of relevant parameter names given by a reference to
          an array @parnames
        - a list of relevant parameter descriptions given by a
          reference to an array @pardescs
        - a list of relevant parameter types given by a reference to
          an array @partypes
        - a template of the command line $cmdline.

    The parameter order used in the arrays @parnames, @pardescs,
    and @partypes are assumed to be the same, ie: $parnames[$i],
    $pardescs[$i] and $partypes[$i] refer to the same parameter.

    The type of a parameter is given by a string with one of the
    following value:

        - 'string'  if the parameter is a string
        - 'int'     if the parameter is a positive integer
        - 'extint'  if the parameter is an extended integer
        - 'float'   if the parameter is a float
        - 'switch'  if the parameter is just an option switch

set_mode($mode)
    Sets the current mode to the one name $mode. Reverts to the
    default mode if no mode named $mode exists.

get_mode()
    Gets the name of the current mode.

get_all_param_names()
    Returns an array containing all the potentially relevant
    parameter names, regardless of the current mode.
```

get_all_param_to_descrs_hash()
    Returns a hashtable mapping all the parameter names (regardless of the current mode), to their descriptions.

get_all_param_to_types_hash()
    Returns a hashtable mapping all the parameter names (regardless of the current mode), to their types.

get_param_names()
    Returns an array containing all the relevant parameter names in the current mode.

get_param_descrs()
    Returns an array containing all the relevant parameter descriptions in the current mode.

get_param_types()
    Returns an array containing all the relevant parameter types in the current mode.

get_cmdline()
    Returns the command line template in the current mode.

get_mode_descr($mode)
    Gets the description of the mode named $mode.

get_all_modes()
    Returns an array containing all the mode names.

get_getopt_strings()
    Returns an array of strings that can be passed as parameters to the GetOptions function of the Getopt::Long module. These strings are of the form <param_name>(=(s|i|o|f))?.
    For exemple: "help", "infile=s", etc. All of the parameters are included here, regardless of the current program's mode. Also options for selecting the program's mode are included. These are of the form <mode_name>=s.

execute(\%param_values, $args, $postcmd)
    Executes the command line program $self->{exe} in the current mode with the parameter values being given by the hashtable %param_values (whose reference is passed as a parameter) for the arguments given in $args. Note that the %param_values hashtable should map the parameter names to their actual values. Optionnally, a third argument $postcmd can be passed. It will be appended to the command string and can be used to implement pipes, i/o redirections, etc.

```
            Returns 'undef' if some parameters were missing in the hashtable,
            in which case the program is not executed.

            Returns the return-value of the call to the underlying Perl
            'system' function otherwise.

        make_cmd(\%param_values, $args, $postcmd)
            Identical to the execute function except that the command
            obtained is not executed, but is returned as a string.

EXAMPLE
       This following code snippet gives an example of how the Tifa::Program
       module can be used.

            #
            # Let's wrap the CFRAC program: cfrac_program
            #
        $cfrac_program = new Tifa::Program();

        $program->set_algo("cfrac");
        $program->set_descr("This is the Continued FRACtion algorithm");
        $program->set_exe("cfrac_program");

        @params = (
            "exe",
            "nprimes_in_factor_base",
            "nprimes_tdiv_smooth_nb",
            "nrelations",
            "lsr_method",
            "use_large_primes",
            "nprimes_tdiv",
        );
            #
            # The command line template can be used down the road to produce
            # the exact command to launch the factoring program
            #
        $cmdline =  "exe nprimes_in_factor_base "
                   ."nprimes_tdiv_smooth_nb nrelations lsr_method "
                   ."use_large_primes nprimes_tdiv ";
            #
            # Default mode: We should specify all of the parameter values on
            # the command line
            #
        $program->add_mode("default",
                           "Default mode: specify all parameter values",
                           @params, $cmdline);

        $program->set_default_mode("default");
```

```
        @params  = ("exe", "nprimes_tdiv");
        $cmdline =  "exe nprimes_tdiv";
            #
            # Best mode: Let CFRAC use the precomputed optimal parameter values
            # depending on the size of the number to factor
            #
        $program->add_mode("best", "Best mode: let CFRAC choose optimal values",
                           @params, $cmdline);

        $program->set_mode("best");
            #
            # For example, all of the factoring programs are wrapped in such
            # a way in the Tifa::ProgramRepository module, so that the
            # other TIFA scripts and/or modules can use them without having to
            # know any specifics about the programs. This makes possible to
            # reuse the benchmark framework with an user-defined Tifa::Program
            # without delving into the scripts' internals.
            #

EXPORT
       No functions are exported from this package by default.

SEE ALSO
       The Tifa::ProgramRepository module.
```

L Man page of the Tifa::ProgramRepository.pm Perl module

```
Tifa::ProgramRepository(1)    TIFA Documentation    Tifa::ProgramRepository(1)

NAME
       Tifa::ProgramRepository - A repository of available Tifa::Program's

SYNOPSIS
        use Tifa::ProgramRepository;

        %algo_to_program = Tifa::ProgramRepository::get_algo_to_program_hash();

REQUIRE
       Perl 5.006002, Carp, Class::Struct, Exporter and Tifa::Program.

SUMMARY
       The Tifa::ProgramRepository module acts as a repository of all
       available Tifa::Program objects.

DESCRIPTION
       The Tifa::ProgramRepository module acts as a repository of all
       available (factoring) Tifa::Program's. The listed Tifa::Program objects
       can then be used in other scripts or modules.

       Available methods

           get_algo_to_program_hash()
           get_algo_list()
           get_program_list()
           get_program_from_name($progname)
           get_all_param_names()
           get_all_param_to_descrs_hash()
           get_all_param_to_types_hash()

       Methods description

           get_algo_to_program_hash()
               Returns a hashtable mapping algorithm names to the Tifa::Program
               objects implementing it.

           get_algo_list()
               Returns an array listing the names of the implemented algorithm.

           get_program_list()
               Returns an array listing the Tifa::Program objects available.

           get_program_from_name($progname)
               Returns a reference to the Tifa::Program object whose default
```

```
            program name is given by $progname (or the base name of $progname
            if $progname is a path).

        get_all_param_names()
            Returns an array listing all the parameter names from all
            programs.

        get_all_param_to_descrs_hash()
            Returns a hashtable mapping all the parameter names (from all
            programs and regardless of their current modes), to their
            descriptions.

        get_all_param_to_types_hash()
            Returns a hashtable mapping all the parameter types (from all
            programs and regardless of their current modes), to their
            descriptions.

EXAMPLE
       This following code snippet gives an example of how the
       Tifa::ProgramRepository module can be used.

        %algo_to_program = Tifa::ProgramRepository::get_algo_to_program_hash();

        $program = $algo_to_program{"squfof"};

        %param_vals = {
            "exe"   => "./squfof_program",
            "ntdiv" => 128,
        };

        $program->execute(\%param_vals, 816379, "> trace.txt");

EXPORT
       No functions are exported from this package by default.

SEE ALSO
       The Tifa::Program module's man page.
```

M Man page of the Tifa::Rotor.pm Perl module

```
Tifa::Rotor(1)                TIFA Documentation                Tifa::Rotor(1)

NAME
       Tifa::Rotor - A basic odometer-like rotor/counter

SYNOPSIS
         use Tifa::Rotor;
         $rotor = new Tifa::Rotor(@array);

REQUIRE
       Perl 5.006002, Carp, and Exporter.

SUMMARY
       The Tifa::Rotor module is yet another weird TIFA module implementing an
       odometer-like counter that can be used to generate all possible
       n-tuples from n sets of values.

DESCRIPTION
       The Tifa::Rotor module implements an odometer-like counter that can be
       used to generate, step by step, all possible n-tuples (a, b, ..., n)
       from n sets: A={0, 1, ..., a_max}, B={0, 1, ..., b_max}, ..., N={0, 1,
       ..., n_max}.

       In TIFA, it is used in the benchmark.pl and plotmaker.pl scripts to
       loop through all of the possible combinations of the parameter values
       to either: 1) generate an exhaustive benchmark for a wide set of inputs
       or 2) generate plots for each of these combinations.

       Available methods

       This module provides the following methods:

           new(@array)
           reset()
           get_state()
           set_state(@array)
           increment()
           is_zero()

       Methods description

           new(@array)
               Basic constructor allocating a Tifa::Rotor object from an array
               of integer @array. The length of @array gives the number of
               "wheels" of the rotor, or in other words, the size of the
               n-tuples generated. The values of @array sets the upper limit
```

```
            on each value of a given n-tuple. For example, if
            @array = (5, 2, 7) then the 3-tuples generated will be of
            the form:
                (a, b, c) with 0 <= a < 5; 0 <= b < 2 and 0 <= c < 7.

        reset()
            Resets the state of the rotor to the (0, ..., 0) n-tuple.

        get_state()
            Returns the current state of the rotor as an array of integer.

        set_state(@array)
            Sets the current state of the rotor to @array. @array must have
            the required length and its values must be consistent with the
            ranges provided to the constructor or an error will be raised.

        increment()
            Increments the rotor, changing its state and thus providing a
            new n-tuple.

        is_zero()
            Returns 1 if the rotor's current state is (0, ..., 0).
            Returns 0 otherwise.

EXAMPLE
       This following code snippet gives an example of how (and why) the
       Tifa::Rotor module can be used... No, it is not _that_ useless!

        #
        # In this example, we'd like to perform some operations on every
        # possible combination of the velocity, angle and height values.
        #
        %values = (
            "velocity"=>[1, 3, 5],    # velocity can take one of these 3 values
            "angle"   =>[20, 45, 70], # angle can take one of these 3 values
            "height"  =>[10, 20]      # height can take one of these 2 values
        );
        @params = keys %values;

        @sizes = ();
        foreach $p (@params) {
            push(@sizes, scalar @{$values{$p}} );
        }

        $rotor = new Tifa::Rotor(@sizes);

        do {
            @ntuple = $rotor->get_state(); # initial state is (0, 0, 0)
```

```
    %combo  = ();

    foreach $i (0 .. scalar(@ntuple)-1) {
        #
        # Fetch the parameter values using the indices given by the
        # rotor's state.
        #
        $parname = $params[$i];
        $combo{$parname} = $values{$parname}[$ntuple[$i]];
    }
    #
    # Now %combo is an hash holding a possible combination of
    # the parameter values. It could be represented as:
    #
    # %combo = ("velocity" => <X>, "angle" => <Y>, "height" => <Z>);
    #
    # Do something with it before stepping to the next combination...
    #
    $rotor->increment(); # step to the next rotor state
 } while (! $rotor->is_zero);
```

In this particularly simple example, there is no real use for the Tifa::Rotor module as three "for" loops would do the job more straitforwardly. However in more general cases the %values could be extracted from a configuration file, passed as a parameter, etc., so using a Rotor object makes for a much more reuseable code.

EXPORT

No functions are exported from this package by default.

N Man page of the Tifa::SimpleConfigReader.pm Perl module

```
Tifa::SimpleConfigReader(1)   TIFA Documentation   Tifa::SimpleConfigReader(1)

NAME
       Tifa::SimpleConfigReader - Truly minimalist configuration file reader

SYNOPSIS
         use Tifa::SimpleConfigReader;
         $reader = new Tifa::SimpleConfigReader();

REQUIRE
       Perl 5.006002, Carp and Exporter.

SUMMARY
       This is a truly minimalist configuration file reader used by the
       scripts of the TIFA library. As such, the needs it fits are precisely
       defined and consequently its feature set and tolerance to syntax errors
       are very limited. This is certainly not a general purpose configuration
       file parser, so its use outside of the TIFA libray is really
       discouraged.

DESCRIPTION
       Full-featured, general purpose configuration file parsers abound on the
       Comprehensive Perl Archive Network, from the most simple readers (e.g.
       ConfigReader::Simple) to the most sophisticated parsers (e.g.
       Config::Scoped) understanding blocks and interpolations. The objective
       of the Tifa::SimpleConfigReader module is extremely modest: this module
       is not trying to re-invent (a worse version of) the wheel but to fit a
       very specific need encountered during the development of the TIFA
       library. So, yes, its feature set is very restricted and next to no
       syntaxic sugar is allowed in the configuration files (welcome back to
       the 70s!).

       Configuration file format

       Comments beginning by a '#' are allowed in the configuration files
       provided that they stand on their own line.

       Three kinds of parameters are allowed: scalars, arrays and hashtables.

       Scalar parameter values are given using the most basic notation:

           <scalar_param> = <value>

       Array parameter values can be given either by providing a list, or by
       providing a range. A list is introduced by the "list" keyword followed
       by a comma-delimited list of values in brackets:
```

```
    <array_param> = list(<value1>, <value2>, <value3>)
```

A range is described by the "range" keyword and its three values given by the "from", "to" and "increment" keywords:

```
    <array_param> = range(from <value1> to <value2> increment <value3>)
```

Finally, hashtable parameters are introduced by the "hash" keyword:

```
    <hash_param> = hash(<key1> => <value1>, ..., <keyN> => <valueN>)
```

Directives can span multiples lines, as long as line breaks are explicitely marked by ' \' (a space followed by a backslash). For example, the following code will be interpreted correctly:

```
    #
    # This is correct: line breaks are explicitely indicated
    #
    my_important_parameter = range(             \
                                from 10         \
                                to   120        \
                                increment 5     \
                             )
```

However, the following snippet will produce a syntax error:

```
    #
    # This is incorrect: line breaks are not explicitely indicated
    #
    my_important_parameter = range(
                                from 10
                                to   120
                                increment 5
                             )
```

Another restriction lies in the format of the parameter names: they should imperatively begin by a letter or by an underscore, although digits are allowed in the other positions.

Generally speaking, Tifa::SimpleConfigReader is quite leniant with whitespaces, be it in the beginning of a line, before of after the = sign, in a list of values or in a range definition.

If several directives for the same scalar parameter are given in the configuration file, previous values are discarded. However, if several directives for the same array or hash parameter are given, all the values will be kept in the array or hash. For example, let's consider

the following configuration snippet:

```
    location = Stanford
    location = Hamburg
    location = Geneva
    #
    # The final value for the "location" key is Geneva.
    #
    years = list(1985, 1995)
    years = range(from 2010 to 2016 increment 2)
    #
    # The final "years" array will contain 1985, 1995, 2010, 2012, 2014
    # and 2016.
    #
    positions = hash(CMS   => Octant5)
    positions = hash(Atlas => Octant7)
    positions = hash(Atlas => Octant1)
    #
    # The final "positions" hashtable will contain the mapping
    # CMS => Octant5 and Atlas => Octant1.
    #
```

Available methods

This module provides the following methods:

```
    new()
    get_parameter_hash()
    set_scalar_param_names(@names)
    set_array_param_names(@names)
    set_hash_param_names(@names)
    get_scalar_param_names()
    get_array_param_names()
    get_hash_param_names()
    print_all_param_values()
    read_config_file($filename);
    accept_unknown_param()
    dont_accept_unknown_param()
    use_fatal_warnings()
    use_non_fatal_warnings()
```

Methods description

```
    new()
        Basic constructor allocating a Tifa::SimpleConfigReader object.

    get_parameter_hash()
        Returns a hashtable mapping the names of the parameters read in
```

the configuration file to their respective values.

set_scalar_param_names(@names)
Sets the names of the parameters supposed to be scalars. Each entry in the @names array should be the name of a parameter expected to be read in the configuration file.

set_array_param_names(@names)
Sets the names of the parameters supposed to be arrays. Each entry in the @names array should be the name of a parameter expected to be read in the configuration file.

set_hash_param_names(@names)
Sets the names of the parameters supposed to be hashtables. Each entry in the @names array should be the name of a parameter expected to be read in the configuration file.

get_scalar_param_names()
Returns the names of the scalar parameters as an array.

get_array_param_names()
Returns the names of the array parameters as an array.

get_hash_param_names()
Returns the names of the hash parameters as an array.

print_all_param_values()
Prints all of the parameter names and their respective values.

read_config_file($filename);
Reads a configuration file $filename in the current context and stores the values of the parameters read.

accept_unknown_param()
Accept unknown parameter names to appear in the configuration file and treat them as if their were declared via the set_*_param_names() functions. This can be used if the parameters likely to appear in the configuration file are not known in advance.

dont_accept_unknown_param()
Do not allowed undeclared parameters names for a strict check of the configuration file syntax.
This is the default behaviour.

use_fatal_warnings()
Exit if an unknown parameter name if found. (This setting has no effect if unknown parameter names were explicitely allowed by

```
            calling the accept_unknown_param() function.)
            This is the default behaviour.

        use_non_fatal_warnings()
            Warn if an unknown parameter name if found and ignore it, but
            do not exit. (This setting has no effect if unknown parameter
            names were explicitely allowed by calling the
            accept_unknown_param() function.)

EXAMPLE
       This following code snippet gives an example of how to use the
       Tifa::SimpleConfigReader module to parse a simple configuration file.
       Let the configuration file be given by:

        #
        # Write some meaningful comments here...
        #
        accelerator = Large Hadron Collider
        location    = Geneva

        #
        # Now, some array definitions...
        #
        experiments = list(Alice, Atlas, CMS, LHCb, TOTEM)
        bunch_nb    = range(from 0 to 2834 increment 1)
        objectives  = list(everything)

        #
        # And finally, a hashtable
        #
        positions = hash(Atlas => Octant1, CMS => Octant5)

       The following code reads this configuration file and prints the values
       of the parameters on the standard output:

        $reader = new Tifa::SimpleConfigReader();
        #
        # Inform the SimpleConfigReader object about the parameters defined
        # in the configuration file. Be warned that any parameter not
        # specified via the set_*_param_names methods could be treated as a
        # syntax error if the function accept_unknown_param() has not been
        # called.
        #
        @scalar_names = ("accelerator", "location");
        @array_names = ("experiments", "bunch_nb", "objectives");

        $reader->set_scalar_param_names(@scalar_names);
        $reader->set_array_param_names(@array_names);
```

```
 $filename = "config.txt";

 $reader->read_config_file($filename);
 $reader->print_all_param_values();
 #
 # Parameter values are now accessible via the get_parameter_hash()
 # method
 #
 %all_params = $reader->get_parameter_hash();
 $where      = $all_params{"location"};
 @detectors  = @{$all_params{"experiments"}};

 print("$detectors[1] is located near $where.\n");
```

The (truncated) output is:

```
    location =  Geneva
    accelerator =  Large Hadron Collider
    bunch_nb = {
        0
        1
        2
        [...]
        2833
        2834
    }
    experiments = {
        Alice
        Atlas
        CMS
        LHCb
        TOTEM
    }
    objectives = {
        everything
    }
    positions = {
        CMS => Octant5
        Atlas => Octant1
    }
    Atlas is located near Geneva.
```

EXPORT

No functions are exported from this package by default.