Calibrator: un software libre para obtener parámetros empíricos necesarios en modelos de simulación

J. Burguete

29 de octubre de 2015

Capítulo 1

Generando el fichero ejecutable a partir del código fuente

El código fuente en Calibrator está escrito en lenguaje C. El programa ha sido compilado y probado en los siguientes sistemas operativos:

- Debian kFreeBSD y Linux 8,
- DragonFly BSD 4.2,
- FreeBSD 10,
- NetBSD 7.0,
- OpenBSD 5.8,
- Windows 7[1],
- and Windows 8.1[1].

Es probable que también puede compilarse y funcione en otros sistemas operativos, otras distribuciones de software y otras versiones pero no ha sido probado.

Para generar el fichero ejecutable a partir de código fuente, un compilador de C (GCC [2015] o Clang [2015]), los sistemas de configuración Autoconf [2015] y Automake [2015], el programa de control de la creación de ejecutables GNU-Make [2015] and las siguienres librerías de software libre son necesarias:

- Libxml [2015]: Librería requerida para leer el fichero principal de entrada en formato XML.
- GSL [2015]: Librería científica usada para generar los números pseudo-aleatorios usados por los algoritmos genético y de Monte-Carlo.

[1] Windows 7 y Windows 8.1 son marcas registradas de Microsoft Corporation.

- GLib [2015]: Librería requerida para analizar las plantillas de los ficheros de entrada y para implementar algunos tipos de datos, funciones útiles y rutinas usadas para paralelizar la carga computacional en los diferentes procesadores de la máquina.
- GTK+ [2015]: Librería opcional usada para dibujar la interfaz gráfica interactiva.
- OpenMPI [2015] o MPICH [2015]: Librerías opcionales. Cuando están instaladas en el sistema una de ellas es usada para permitir la paralelización del cálculo en múltiples computadoras.

Las indicaciones proporcionadas en Install-UNIX [2015] pueden seguirse para instalar todas estas utilidades.

En OpenBSD 5.8, antes de generar el código, deben seleccionarse versiones adecuadas de Autoconf y Automake haciendo en un terminal:

```
$ export AUTOCONF_VERSION=2.69 AUTOMAKE_VERSION=1.15
```

En sistemas Windows hay que instalar MSYS2 (http://sourceforge.net/projects/msys2) y las librerías y utilidades requeridas. Para ello se pueden seguir las instrucciones detalladas en https://github.com/jburguete/install-unix.

Una vez que todas las utilidades necesarias han sido instaladas, hay que descargar el código de Genetic. Luego puede compilarse haciendo en un terminal:

```
$ git clone https://github.com/jburguete/genetic.git
$ cd genetic/0.6.1
$ ./build
```

El siguiente paso es descargar el código fuente de Calibrator, enlazarlo con el de Genetic y compilar todo junto haciendo:

```
$ git clone https://github.com/jburguete/calibrator.git
$ cd calibrator/1.1.26
$ ln -s ../../genetic/0.6.1 genetic
$ ./build
```

Capítulo 2

Interfaz

2.1. Formato en línea de comandos

- La línea de comandos en modo secuencial es (donde X es el número de tareas a ejecutar paralelamente):

```
$ ./calibratorbin [-nthreads X] input_file.xml
```

- La línea de comandos en modo paralelizado en diferentes computadoras con MPI es (donde X es el número de tareas a ejecutar en cada nodo):

```
$ mpirun [MPI options] ./calibratorbin [-nthreads X] input_file.xml
```

- La sintaxis del programa simulador ha de ser:

```
$ ./simulator_name input_file_1 [input_file_2] [...] output_file
```

  Hay dos opciones para el fichero de salida. Puede comenzar con un número que indique el valor de la función objetivo o puede ser un fichero de resultados que tiene que ser evaluado por un programa externo (el evaluador) comparando con un fichero de datos experimentales.

- En caso de la última opción del punto anterior, la sintaxis del programa para evaluar la función objetivo tiene que ser (donde el fichero de resultados debe comenzar con el valor de la función objetivo):

```
$ ./evaluator_name simulated_file experimental_file results_file
```

2.2. Ficheros de entrada

2.2.1. Main input file

This file has to be in XML format with a tree type structure as the represented in figure 2.1.

The main XML node has to begin with the key label *çalibrate"*. The available properties are:

simulator : to indicate the simulator program.

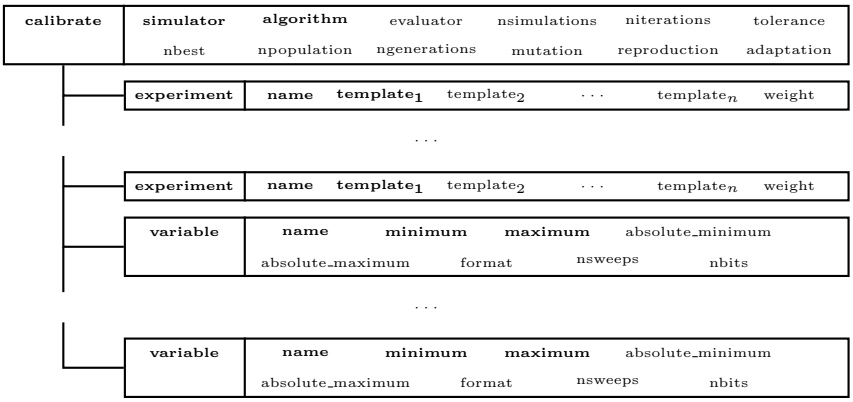

Figura 2.1: Structure of the main input file. Mandatory nodes and properties are in bold. Others properties can be also mandatory depending on the selected optimization algorithm.

evaluator : optional. It specifies the evaluator program if required.

algorithm : to set the optimization algorithm. Three value are currently available:

sweep : sweep brute force algorithm. It requires for each variable:

nsweeps : number of sweeps to generate each variable in every experiment.

Monte-Carlo : Monte-Carlo brute force algorithm. It requires on the main XML node:

nsimulations : number of simulations to run for each iteration in every experiment.

genetic : genetic algorithm. It requires the following parameters in the main XML node:

npopulation : number of population entities.

ngenerations : number of generations.

mutation : mutation ratio.

reproduction : reproduction ratio.

adaptation : adaptation ratio.

And for each variable:

nbits : number of bits to encode each variable.

niterations : number of iterations (default 1) to perform the iterative algorithm.

nbest : number of best simulations to calculate convergence interval on next iteration for the iterative algorithm (default 1).

tolerance : tolerance parameter to relax the convergence interval of the iterative algorithm (default 0).

The first type of child XML nodes has to begin with the key label *."experiment"*. It details the experimental data and it contains the properties:

name : name of the input data file with experimental results to calibrate.

templateX : X-th input data file template for the simulation program.

weight : weight (default 1) to apply in the objective function (see (3.1)).

The second type of child XML nodes has to begin with the key label *"variable"*. It specifies the variables data and it has the properties:

name : variable label. On the X-th variable, the program parse all input file templates creating the input simulation files by replacing all @variableX@ labels by this name.

minimum, maximum : variable extreme values. The program creates the input simulation files by replacing all @valueX@ labels in the input file templates by a value between these extreme values on the X-th variable, depending on the optimization algorithm.

absolute_minimum, absolute_maximum : absolute variable extreme values. On iterative methods, the tolerance can increase initial *minimum* or *maximum* values in each iteration. These values are the allowed extreme values compatible with the model parameter limits.

precision : number of decimal digits of precision. 0 apply for integer numbers.

2.2.2. Template files

$N_{experiments} \times N_{inputs}$ template files must be written to reproduce every input file associated to every experiment (see figure 3.1). All the template files are syntactically analyzed by Calibrator to replace the labels as follows in order to generate the simulation program input files:

@variableX@ : is replaced by the label associated to the X-th empirical parameter defined in *main input file*;

@valueX@ : is replaced by the value associated to the X-th empirical parameter calculated by the optimization algorithm using the format defined in *main input file*;

2.3. Interactive GUI application

An alternative form to execute the software is to perform the interactive GUI application, called *Calibrator*. In the figure 2.2 a plot of the main window of this tool is represented. The main windows enable us to access to every variable, coefficient, algorithm and simulation softwares.

2.4. Output files

2.4.1. Results file

Calibrator generates a file named *result* where the best combination of variables and the corresponding calculated error are saved.

Figura 2.2: Main window of Calibrator GUI application.

2.4.2. Variables file

The program generates also a file named *variables* where all combinations of variables checked in the calibration are saved in columns being the last column the associated error.

Capítulo 3

Organization of Calibrator

Let us assume that $N_{parameters}$ empirical parameters are sought desired so that the results from a simulation model are the best fit to $N_{experiments}$ experimental data and that the simulator requires N_{inputs} input files. The structure followed by Calibrator is summarized in *main input file*, where both $N_{experiments}$ and N_{inputs} are specified. Furthermore, it contains the extreme values of the empirical parameters and the chosen optimization algorithm. Then, Calibrator reads the $N_{experiments} \times N_{inputs}$ templates to build the simulator input files replacing key labels by empirical parameter values created by the optimization algorithm. There are two options: either the simulator compares directly the simulation results with the *experimental data file*, hence generating a file with the value of the error, or an external program called *evaluator* is invoked to compare with the *experimental data file* and to produce the error value. In both cases this error value is saved in an *objective value file*. Then for each experiment, an objective value o_i is obtained. The final value of the objective function associated with the experiments is calculated by means of:

$$J = \sqrt{\frac{1}{N_{experiments}} \sum_{i=1}^{N_{experiments}} |w_i \, o_i|^2}, \tag{3.1}$$

with w_i the weight associated to the i-th experiment, specified in the *main input file*. Figure 3.1 is a sketch of the structure.

The whole process is repeated for each combination of empirical parameters generated by the optimization algorithm. Furthermore, Calibrator automatically parallelizes the simulations using all the available computing resources.

The required format for the *main input file* and the *template files* are described in next section.

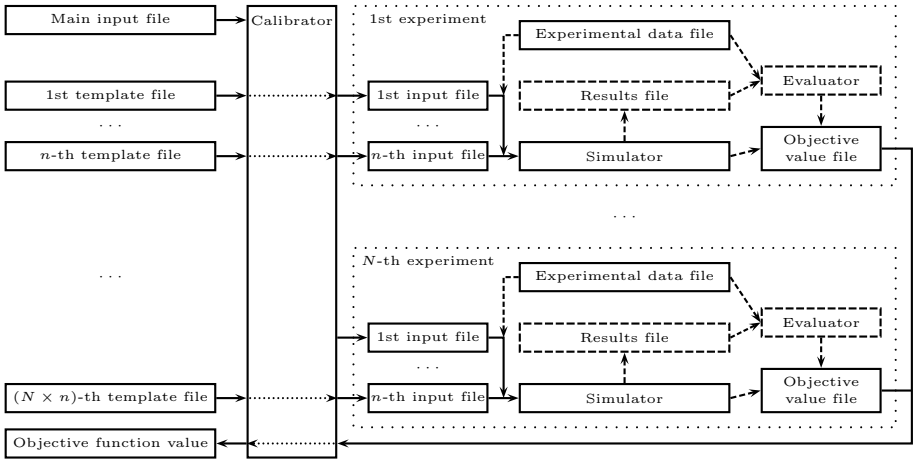

Figura 3.1: Flowchart of the interactions among Calibrator, the input files and the simulation and evaluation programs to produce an objective function value for each empirical parameters combination generated by the optimization algorithm.

Capítulo 4

Optimization methods

The optimization methods implemented in Calibrator are next presented. The following notation will be used:

$N_{simulations}$: number of simulations made for each iteration.

$N_{iterations}$: number of iterations on iterative methods.

N_{total} : total number of simulations.

In iterative methods $N_{total} = N_{simulations} \times N_{iterations}$. In pure brute force methods $N_{iterations} = 1 \Rightarrow N_{total} = N_{simulations}$.

4.1. Sweep brute force method

The sweep brute force method finds the optimal set of parameters within a solution region by dividing it into regular subdomains. To find the optimal solution, the domain interval $x_i \in (x_{i,min}, x_{i,max})$ is first defined for each variable x_i. Then, a regular partition in $N_{x,i}$ subintervals is made. Taking into account this division of the solution space, the number of required simulations is:

$$N_{simulations} = N_{x,1} \times N_{x,2} \times \cdots, \tag{4.1}$$

where $N_{x,i}$ is the number of sweeps in the variable x_i.

In figure 4.1 the (x, y) domain is defined by the intervals $x \in (x_{min}, x_{max})$ and $y \in (y_{min}, y_{max})$. Both x and y intervals are divided into 5 regions with $N_x = N_y = 5$. The optimal will be found within the region by evaluating the error of each (x_i, y_i) set of parameters hence requiring 25 evaluations. Note that the computational cost increases strongly as the number of variables grow.

Brute force algorithms present low convergence rates but they are strongly parallelizable because every simulation is completely independent. If the computer, or the computers cluster, can execute N_{tasks} parallel tasks every task do N_{total}/N_{tasks} simulations, obviously taking into account rounding effects (every task has to perform a natural number of simulations). In figure 4.2 a flowchart of this parallelization scheme is represented. Being independent each task, a distribution on different execution threads may be performed exploding the full parallel capabilities of the machine where Calibrator is run.

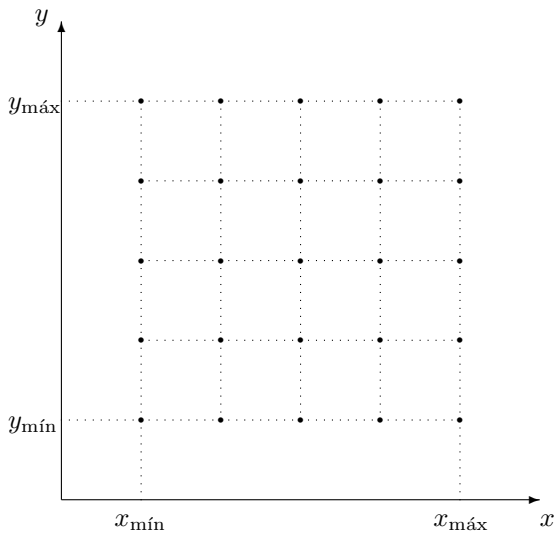

Figura 4.1: Diagram showing an example of application of the sweep brute force method with two variables for $N_x = N_y = 5$.

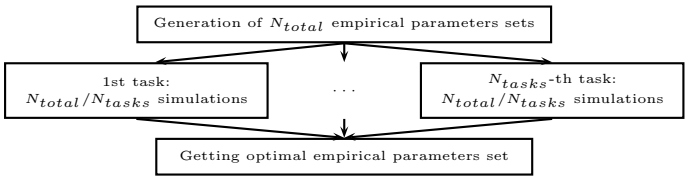

Figura 4.2: Flowchart of the parallelization scheme in Calibrator for brute force methods (sweep and Monte-Carlo).

4.2. Monte-Carlo method

Monte-Carlo based methods run simulations using aleatory values of the variables assuming uniform probability within the extreme values range. Figure 4.3 shows the structure of an example using two variables.

Monte-Carlo method is also easily parallelizable following a strategy as the flowchart represented in the figure 4.2.

4.3. Iterative algorithm applied to brute force methods

Calibrator allows to iterate both sweep or Monte-Carlo brute force methods in order to seek convergence. In this case, the best results from the previous iteration are used to force new intervals in the variables for the following iteration. Then for N^j_{best}, the subset of the best simulation results in the j-th iteration, the following quantities are defined:

$x^b_{\text{máx}} = \underset{i \in N_{best}}{\text{máx}}\ x^j_i$: Maximum value of variable x in the subset of the best simulation results from the j-th iteration.

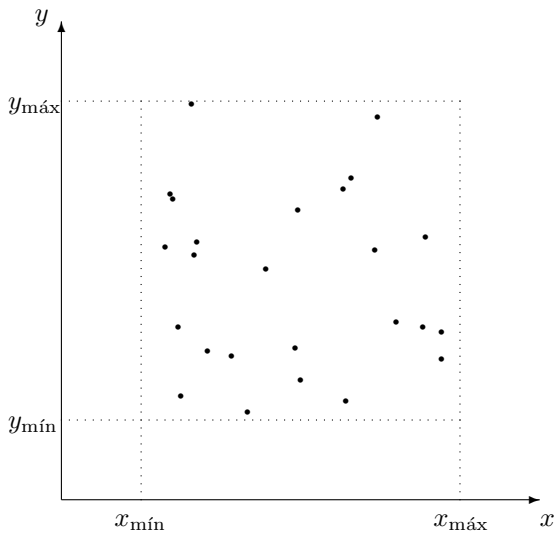

Figura 4.3: Diagram illustrating a Monte-Carlo brute force method with two variables and $N_{simulations} = 25$.

$x^b_{\text{mín}} = \underset{i \in N_{best}}{\text{máx}}\ x^j_i$: Minimum value of variable x in the subset of the best simulation results from the j-th iteration.

A new interval in the variable x is defined to build the optimization values in the next $(j+1)$ iteration so that:

$$x_i^{j+1} \in \left[x^{j+1}_{\text{mín}},\ x^{j+1}_{\text{máx}}\right], \tag{4.2}$$

with:

$$x^{j+1}_{\text{máx}} = \frac{x^b_{\text{máx}} + x^b_{\text{mín}} + \left(x^b_{\text{máx}} - x^b_{\text{mín}}\right)(1 + tolerance)}{2},$$
$$x^{j+1}_{\text{mín}} = \frac{x^b_{\text{máx}} + x^b_{\text{mín}} - \left(x^b_{\text{máx}} - x^b_{\text{mín}}\right)(1 + tolerance)}{2},$$

being *tolerance* a factor increasing the size of the variable intervals to simulate the next iteration. Figure 4.4 contains a sketch of the procedure used by the the iterative algorithm to modify the variables intervals in order to enforce convergence.

The iterative algorithm can be also easily parallelized. However, this method is less parallelizable than pure brute force methods because the parallelization has to be performed for each iteration (see a flowchart in the figure 4.5).

4.4. Genetic method

Calibrator also offers the use of a genetic method Genetic Genetic [2015] with its default algorithms. It is inspired on the ideas in GAUL [2015], but it has been fully reprogrammed involving more modern external libraries. The code in Genetic is also open source under BSD license. Figure 4.6 shows the flowchart of the genetic method implemented in Genetic.

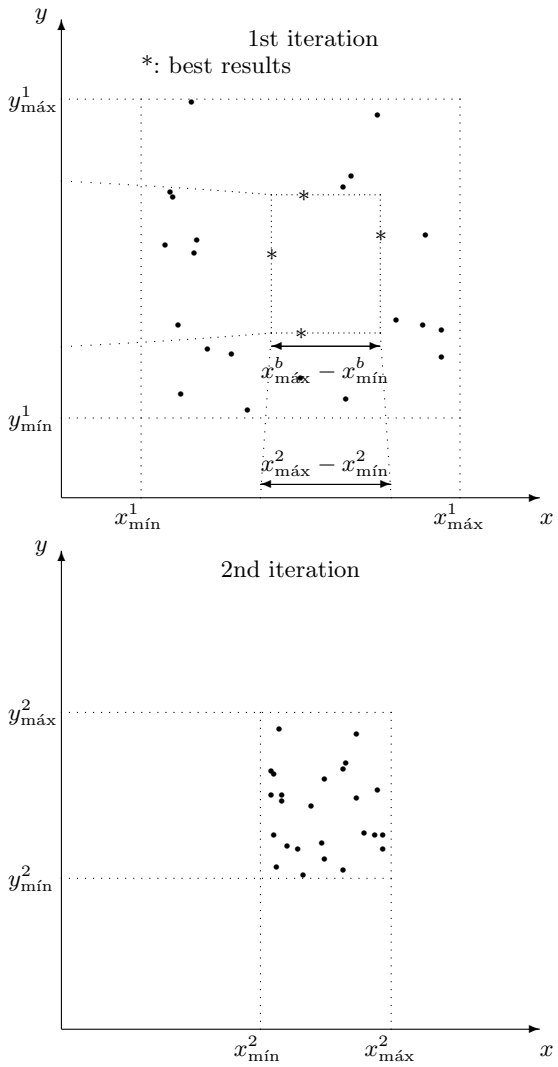

Figura 4.4: Diagram representing an example of the iterative algorithm applied to a Monte-Carlo brute force method with two variables for $N_{simulations} = 25$, $N_{best} = 4$ and two iterations.

4.4.1. The genome

The variables to calibrate/optimize are coded in Genetic using a bit chain: the genome. The larger the number of bits assigned to a variable the higher the resolution. The number of bits assigned to each variable, and therefore the genome size, is fixed and the same for all the simulations. Figure 4.7 shows an example for the coding of three variables. The value assigned to a variable x is determined by the allowed extreme values $x_{\text{mín}}$ and $x_{\text{máx}}$, the binary number assigned in the genome to variable I_x and by the number of bits assigned to

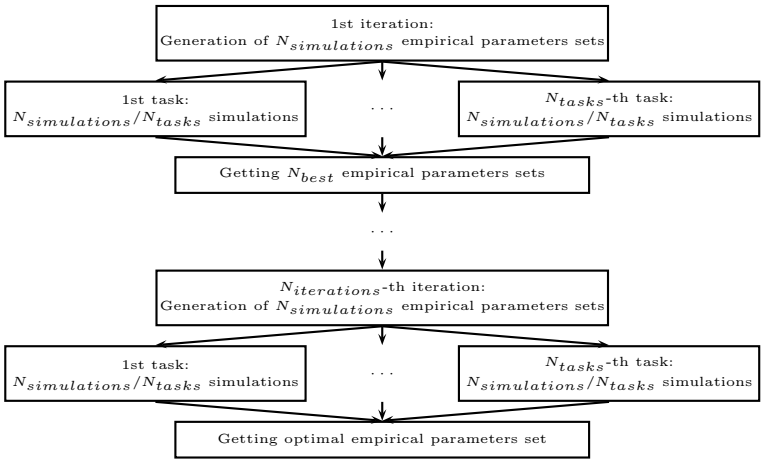

Figura 4.5: Flowchart of the parallelization scheme in Calibrator for the iterative method.

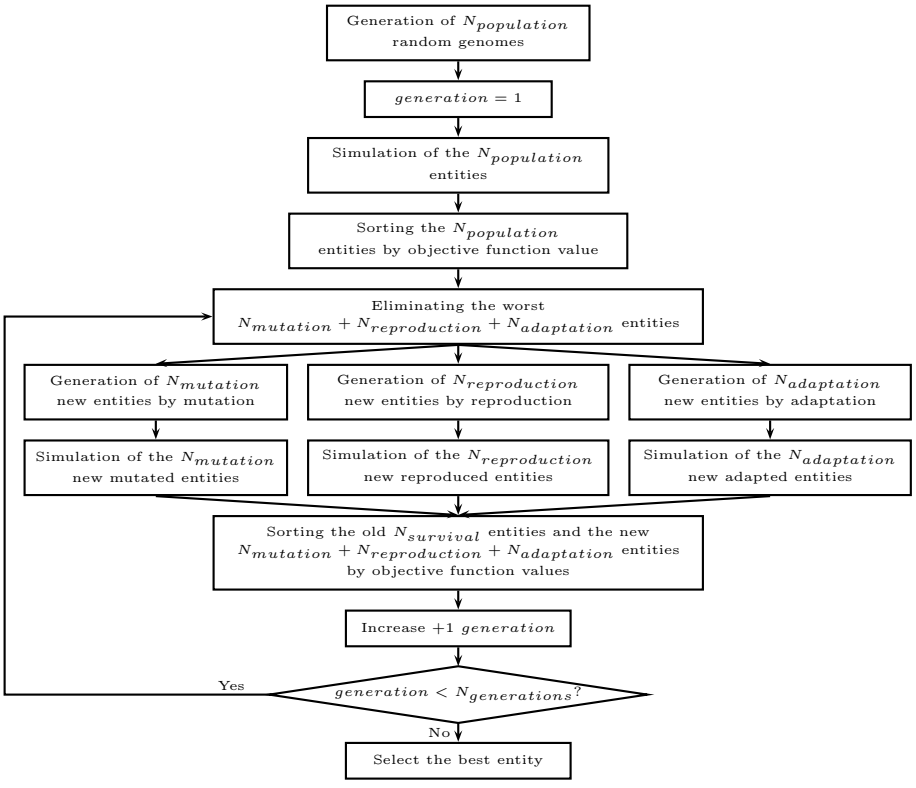

Figura 4.6: Flow diagram of the genetic method implemented in Genetic.

variable N_x according to the following formula:

$$x = x_{\text{mín}} + \frac{I_x}{2^{N_x}} \left(x_{\text{máx}} - x_{\text{mín}}\right). \tag{4.3}$$

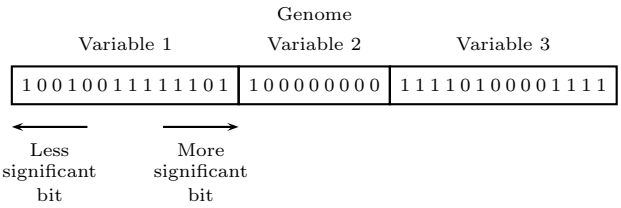

Figura 4.7: Example coding three variables to optimize into a genome. The first and third variables have been coded with 14 bits, and the second variable has been coded with 9 bits.

4.4.2. Survival of the best individuals

In a population with $N_{population}$ individuals, in the first generation all the cases are simulated. The input variables are taken from the genome of each individual. Next, in every generation, $N_{population} \times R_{mutation}$ individuals are generated by mutation, $N_{population} \times R_{reproduction}$ individuals are generated by reproduction and $N_{population} \times R_{adaptation}$ individuals are generated by adaptation, obviously taking into account rounding. On second and further generations only simulations associated to this new individuals (N_{new}) have to be run:

$$N_{new} = N_{population} \times (R_{mutation} + R_{reproduction} + R_{adaptation})\,. \tag{4.4}$$

Then, total number of simulations performed by the genetic algorithm is:

$$N_{total} = N_{population} + (N_{generations} - 1) \times N_{new}, \tag{4.5}$$

with $N_{generations}$ the number of generations of new entities. The individuals of the former population that obtained lower values in the evaluation function are replaced so that the best $N_{survival}$ individuals survive:

$$N_{survival} = N_{population} - N_{new}. \tag{4.6}$$

Furthermore, the ancestors to generate new individuals are chosen among the surviving population. Obviously, to have survival population, the following condition has to be enforced:

$$R_{mutation} + R_{reproduction} + R_{adaptation} < 1 \tag{4.7}$$

Calibrator uses a default aleatory criterion in Genetic, with a probability linearly decreasing with the ordinal in the ordered set of surviving individuals (see figure 4.8).

4.4.3. Mutation algorithm

In the mutation algorithm an identical copy of the parent genome is made except for a bit, randomly chosen with uniform probability, which is inverted. Figure 4.9 shows an example of the procedure.

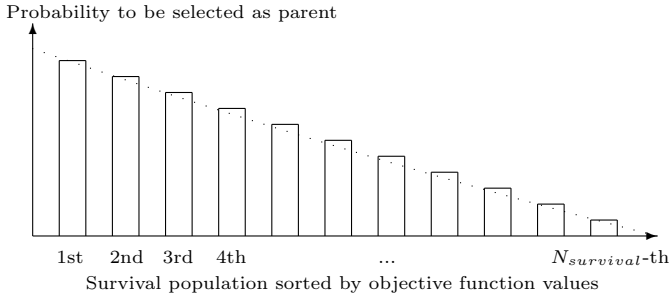

Figura 4.8: Probability of a survival entity to be selected as parent of the new entities generated by mutation, reproduction or adaptation algorithms.

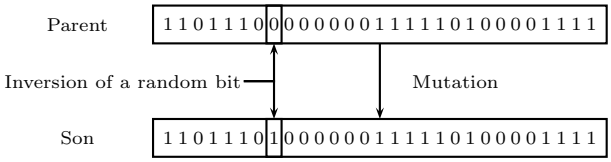

Figura 4.9: Diagram showing an example of the generation of a new entity by mutation.

4.4.4. Reproduction algorithm

The default algorithm in Genetic selects two different parents with one of the least errors after the complete simulation of one generation. A new individual is then generated by sharing the common bits of both parents and a random choice in the others. The new child has the same number of bits as the parents and different genome. Figure 4.10 shows a sketch of the algorithm.

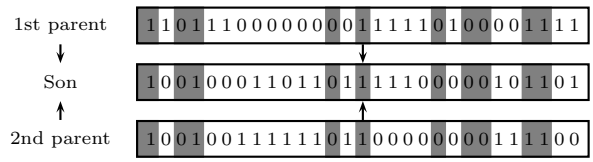

Figura 4.10: Example of the generation of a new entity by reproduction in the Genetic default algorithm. Note that the identical bits in both parents (in grey) are also present in their son. The rest of the bits are random.

4.4.5. Adaptation algorithm

Another algorithm is included in Genetic called .[a]daptation.[a]lthough, in the biological sense, it would be rather be a smooth mutation. First, one of the variables codified in the genome is randomly selected with uniform probability. Then, a bit is randomly chosen assuming a probability linearly decreasing with the significance of the bit. The new individual receives a copy of the parents genome with the selected bit inverted. Figure 4.11 contains an example.

This algorithm is rather similar to the mutation algorithm previously des-

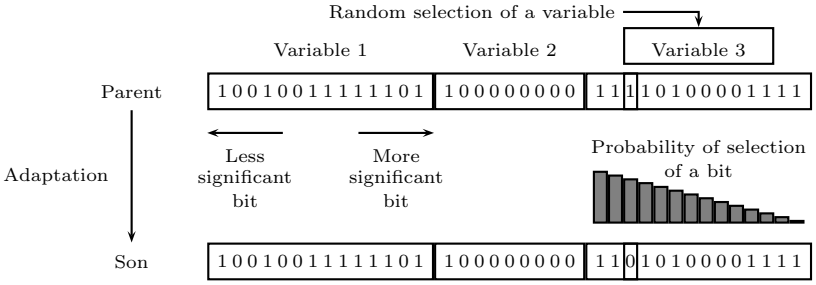

Figura 4.11: Example of the generation of a new individual from a parent by adaptation.

cribed but, since the probability to affect bits less significant is larger, so is the probability to produce smaller changes.

4.4.6. Parallelization

This method is also easily parallelizable following a similar scheme to the iterative algorithm, as it can be seen in the figure 4.12.

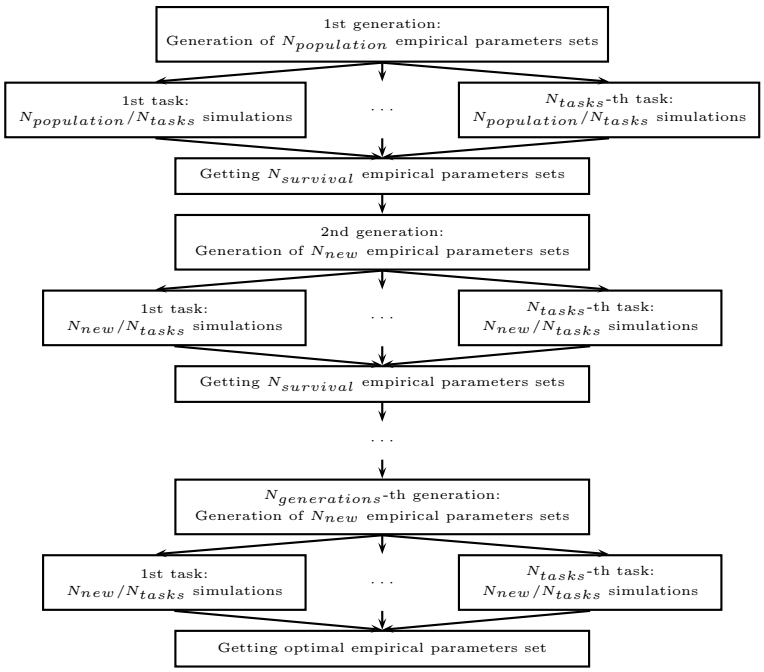

Figura 4.12: Flowchart of the parallelization scheme implemented in Genetic for the genetic method.

Bibliografía

Autoconf. Autoconf is an extensible package of M4 macros that produce shell scripts to automatically configure software source code packages. URL https://www.gnu.org/software/autoconf, 2015.

Automake. Automake is a tool for automatically generating Makefile.in files compliant with the GNU coding standards. URL https://www.gnu.org/software/automake, 2015.

Clang. A C language family frontend for LLVM. URL http://clang.llvm.org, 2015.

GAUL. The genetic algorithm utility library. URL http://gaul.sourceforge.net, 2015.

GCC. The GNU compiler collection. URL https://gcc.gnu.org, 2015.

Genetic. A simple genetic algorithm. URL https://github.com/jburguete/genetic, 2015.

GLib. A general-purpose utility library, which provides many useful data types, macros, type conversions, string utilities, file utilities, a mainloop abstraction, and so on. URL https://developer.gnome.org/glib, 2015.

GNU-Make. GNU Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files. URL http://www.gnu.org/software/make, 2015.

GSL. GNU scientific library. URL http://www.gnu.org/software/gsl, 2015.

GTK+. The GIMP Toolkit, a multi-platform toolkit for creating graphical user interfaces. URL http://www.gtk.org, 2015.

Install-UNIX. A set of makefiles to install some useful applications in several unix type systems. URL https://github.com/jburguete/install-unix, 2015.

Libxml. The XML C parser and toolkit of GNOME. URL http://xmlsoft.org, 2015.

MPICH. High-performance portable MPI. URL http://www.mpich.org, 2015.

OpenMPI. Open source high performance computing. URL http://www.open-mpi.org, 2015.