MPCOTool (the Multi-Purposes Calibration and Optimization Tool): an open source software to supply empirical parameters required in simulation models

Software authors: J. Burguete and B. Latorre
Manual authors: J. Burguete, P. García-Navarro and S. Ambroj

July 19, 2017

Contents

1 Building from the source code **2**

2 Interface **5**
2.1 Command line format 5
2.2 Using MPCOTool as an external library 6
2.3 Interactive graphical user interface application 6
2.4 Input files 7
2.4.1 Main input file 7
2.4.2 Template files 10
2.5 Output files 11
2.5.1 Results file 11
2.5.2 Variables file 11

3 Organization of MPCOTool **12**

4 Optimization methods **14**
4.1 Sweep brute force method 14
4.2 Monte-Carlo method 15
4.3 Iterative algorithm applied to brute force methods 15
4.4 Direction search method 17
4.4.1 Coordinates descent 19
4.4.2 Random 19
4.5 Genetic method 19
4.5.1 The genome 19
4.5.2 Survival of the best individuals 21
4.5.3 Mutation algorithm 21
4.5.4 Reproduction algorithm 22
4.5.5 Adaptation algorithm 22
4.5.6 Parallelization 23

References **24**

Chapter 1

Building from the source code

The source code in MPCOTool is written in C language. This software has been built and tested in the following operative systems:

- Debian Hurd, kFreeBSD and Linux 9;
- DragonFly BSD 4.8;
- Dyson Illumos;
- Fedora Linux 26;
- FreeBSD 11.0;
- Microsoft Windows 7[1], 8.1[1] and 10[1];
- NetBSD 7.0;
- OpenBSD 6.1;
- OpenIndiana Hipster;
- OpenSUSE Tumbleweed;
- and Ubuntu Linux 17.04.

Probably, this software can be built and it works in other operative systems, software distributions or versions but it has not been tested.

In order to build the executable file from the source code, a C compiler (GCC [2017] or Clang [2017]), the configuration systems Autoconf [2017], Automake [2017] and Pkg-config [2017], the executable creation control program GNU-Make [2017] and the following open source external libraries are required:

- Libxml [2017]: Library required to read the main input file in XML format.
- GSL [2017]: Scientific library required to generate the pseudo-random numbers used by the genetic and the Monte-Carlo algorithms.

[1]Windows 7 and Windows 8.1 are trademarks of Microsoft Corporation.

- GLib [2017]: Library required to parse the input file templates and to implement some data types and the routines used to parallelize the usage of the computer's processors.
- JSON-GLib [2017]: Library used to read the main input file in JSON format.
- GTK+3 [2017]: Optional library to build the interactive GUI application.
- OpenMPI [2017] or MPICH [2017]: Optional libraries. When installed, one of them is used to allow parallelization in multiple computers.

The indications provided in Install-UNIX [2017] can be followed in order to install all these utilities.

On OpenBSD 6.0, prior to build the code, you have to select adequate version of Autoconf and Automake doing on a terminal:

```
> export AUTOCONF_VERSION=2.69 AUTOMAKE_VERSION=1.15
```

On Window systems, you have to install MSYS2 (http://sourceforge.net/projects/msys2) and the required libraries and utilities. You can follow detailed instructions in https://github.com/jburguete/install-unix.

On Fedora Linux 26, in order to use OpenMPI compilation, do in a terminal (in 64 bits version):

```
> export PATH=$PATH:/usr/lib64/openmpi/bin
```

On FreeBSD 11.0, due to a wrong error in default gcc version, do in a terminal:

```
> export CC=gcc5 (or CC=clang)
```

Once all the tools installed, the Genetic source code must be downloaded and it must be compiled following on a terminal:

```
> git clone https://github.com/jburguete/genetic.git
> cd genetic/2.2.2
> ./build
```

The following step is to download the source code MPCOTool, to link it with Genetic and compile together by means of:

```
> git clone https://github.com/jburguete/mpcotool.git
> cd mpcotool/3.4.4
> ln -s ../../genetic/2.2.2 genetic
> ln -s genetic/libgenetic.so (or .dll on Windows systems)
> ./build
```

On servers or clusters, where no-GUI with MPI parallelization is desirable, replace the *build* script by:

```
> ./build_without_gui
```

Optionally, to compile the tests with the standard analytical optimization functions, you have to do (the executable files of test2, test3 and test4 use also the *Genetic* library):

```
> cd ../tests/test2
> ln -s ../../../genetic/2.2.2 genetic
> ln -s genetic/libgenetic.so (.dll on Windows systems)
> cd ../test3
> ln -s ../../../genetic/2.2.2 genetic
> ln -s genetic/libgenetic.so (.dll on Windows systems)
> cd ../test4
```

```
> ln -s ../../../genetic/2.2.2 genetic
> ln -s genetic/libgenetic.so (.dll on Windows systems)
> cd ../../3.4.4
> make tests
```

Finally, the next optional step build the PDF manuals:

```
> make manuals
```

Chapter 2

Interface

WARNING! Real numbers are represented according to the international standard, overriding locale settings, separating the integer and decimal parts by ".".

2.1 Command line format

In this section optional arguments are typed in square brackets.

- Command line in sequential mode (where X is the number of threads to execute and S is a seed for the pseudo-random numbers generator):

```
> ./mpcotoolbin [-nthreads X] [-seed S] input_file.xml
[result_file] [variables_file]
```

- Command line in parallelized mode (where X is the number of threads to open for every node and S is a seed for the pseudo-random numbers generator):

```
> mpirun [MPI options] ./mpcotoolbin [-nthreads X] [-seed S]
input_file.xml [result_file] [variables_file]
```

- The syntax of the simulator program has to be:

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

  There are two options for the output file. It can begin with a number indicating the objective function value or it can be a results file that has to be evaluated by an external program (the evaluator) comparing with an experimental data file.

- In the last option of the former point, the syntax of the program to evaluate the objective function has to be (where the results file has to begin with the objective function value):

```
> ./evaluator_name simulated_file experimental_file results_file
```

- On UNIX type systems the GUI application can be open doing on a terminal:

```
> ./mpcotool
```

2.2 Using MPCOTool as an external library

MPCOTool can also be used as an external library by doing:

1. Copy the dynamic library ("libmpcotool.so" on Unix systems or "libmpcotool.dll" on Windows systems) to your program directory.
2. Include the function header in your source code:
   ```
   > extern int mpcotool (int argn, char **argc);
   ```
3. Build the executable file with the linker flags:
   ```
   > $ gcc ... -L. -Wl,-rpath=. -lmpcotool ...
   ```
4. Calling to this function is equivalent to command line order (see previous section):
   - argn: number of arguments including the program name.
   - argc[0]: "mpcotool" (program name).
   - argc[1]: first command line argument.

     ...
   - argc[$argn - 1$]: last command line argument.

2.3 Interactive graphical user interface application

An alternative form to execute the software is to perform the interactive graphical user interface application, called *MPCOTool*. In this application the parallelization in multiple computers with OpenMPI or MPICH is deactivated, it can be only used in command line execution. In the figure 2.1 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.

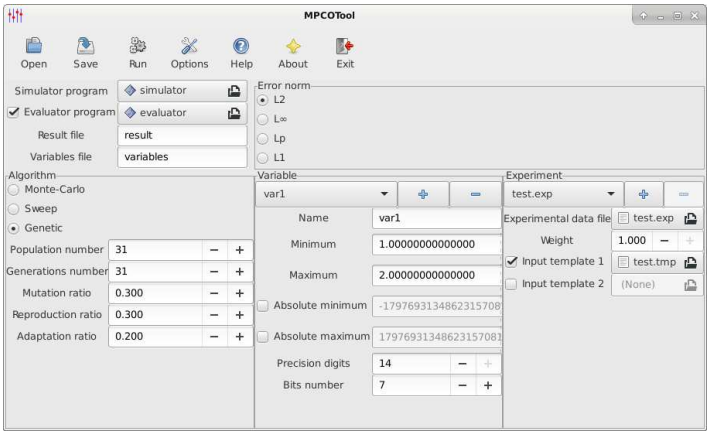

Figure 2.1: Main window of MPCOTool graphical user interface application.

Final optime results are presented in a dialog as the shown in the figure 2.2.

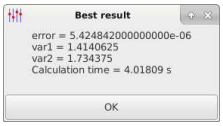

Figure 2.2: Results dialog of MPCOTool graphical user interface application.

2.4 Input files

2.4.1 Main input file

XML format

This file can be written in XML format with a tree type structure as the represented in figure 2.3.

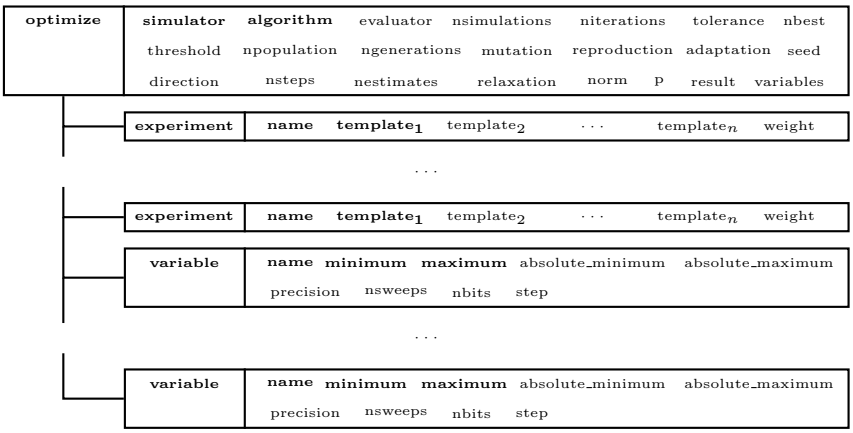

Figure 2.3: Structure of the main input file. Mandatory nodes and properties are in bold. Others properties can be also mandatory depending on the selected optimization algorithm.

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

simulator : to indicate the simulator program,

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

algorithm : to set the optimization algorithm. Three values 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),

threshold : threshold in the objective function to stop the optimization,

seed : seed of the pseudo-random numbers generator (default 7007),

direction : method to search the optimal direction (optional for sweep and Monte-Carlo algorithms). Two values are currently available:

coordinates : coordinates descent search,

random : random search. It requieres:

nestimates : number of random checks to search the optimal direction.

Both methods requires the following parameters:

nsteps : number of steps to perform the direction search method,

relaxation : relaxation parameter for the direction search method,

and for each variable:

step : initial step size for the direction search method,

norm : to set the error norm (default "euclidian"). Four values are currently available:

euclidian : euclidian error norm L_2, see (3.1),

maximum : maximum error norm L_∞, see (3.2),

p : P error norm L_p. It requires:

p : exponent of the P error norm, see (3.3),

taxicab : taxicab error norm L_1, see (3.4),

result : to set the result file name. Optional, the default file name is *"result"*,

variables : to set the variables file name. Optional, the default file name is *"variables"*.

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) to (3.4).

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.

In this format the file is written as:

```
<?xml version="1.0"?>
<optimize simulator="simulator_name" evaluator="evaluator_name"
    algorithm="algorithm_type" nsimulations="simulations_number"
    niterations="iterations_number" tolerance="tolerance_value"
    nbest="best_number" npopulation="population_number"
    ngenerations="generations_number" mutation="mutation_ratio"
    reproduction="reproduction_ratio" adaptation="adaptation_ratio"
    direction="direction_search_type" nsteps="steps_number"
    relaxation="relaxation_parameter" nestimates="estimates_number"
    threshold="threshold_parameter" norm="norm_type" p="p_parameter"
    seed="random_seed" result_file="result_file"
    variables_file="variables_file">
    <experiment name="data_file_1" template1="template_1_1"
        template2="template_1_2" ... weight="weight_1"/>
    ...
    <experiment name="data_file_N" template1="template_N_1"
        template2="template_N_2" ... weight="weight_N"/>
    <variable name="variable_1" minimum="min_value" maximum="max_value"
        precision="precision_digits" sweeps="sweeps_number"
        nbits="bits_number" step="step_size"/>
    ...
    <variable name="variable_M" minimum="min_value" maximum="max_value"
        precision="precision_digits" sweeps="sweeps_number"
        nbits="bits_number" step="step_size"/>
</optimize>
```

Alternatively, the main input file can be also written in JSON format:

```
{
    "simulator": "simulator_name",
    "evaluator": "evaluator_name",
    "algorithm": "algorithm_type",
```

```
    "nsimulations": "simulations_number",
    "niterations": "iterations_number",
    "tolerance": "tolerance_value",
    "nbest": "best_number",
    "npopulation": "population_number",
    "ngenerations": "generations_number",
    "mutation": "mutation_ratio",
    "reproduction": "reproduction_ratio",
    "adaptation": "adaptation_ratio",
    "direction": "direction_search_type",
    "nsteps": "steps_number",
    "relaxation": "relaxation_parameter",
    "nestimates": "estimates_number",
    "threshold": "threshold_parameter",
    "norm": "norm_type",
    "p": "p_parameter",
    "seed": "random_seed",
    "result_file": "result_file",
    "variables_file": "variables_file",
    "experiments":
    [
        {
            "name": "data_file_1",
            "template1": "template_1_1",
            "template2": "template_1_2",
            ...
            "weight": "weight_1",
        },
        ...
        {
            "name": "data_file_N",
            "template1": "template_N_1",
            "template2": "template_N_2",
            ...
            "weight": "weight_N",
        }
    ],
    "variables":
    [
        {

            "name": "variable_1",
            "minimum": "min_value",
            "maximum": "max_value",
            "precision": "precision_digits",
            "sweeps": "sweeps_number",
            "nbits": "bits_number",
            "step": "step_size",
        },
        ...
        {
            "name": "variable_M",
            "minimum": "min_value",
            "maximum": "max_value",
            "precision": "precision_digits",
            "sweeps": "sweeps_number",
            "nbits": "bits_number",
            "step": "step_size",
        }
    ]
}
```

2.4.2 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 MPCOTool 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 pa-

rameter 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 data defined in *main input file*;

2.5 Output files

2.5.1 Results file

MPCOTool generates a file where the best combination of variables and the corresponding calculated objective function, as well as the calculation time, are saved. The file name can be set ih the *result* property of the main input file. If not set the default file name is "result".

2.5.2 Variables file

The program generates also a file where all combinations of variables checked in the calibration are saved in columns. First columns correspond to the variables combination and the last column is the objective function value. The file name can be set in the *varaibles* property of the main input file. A default name "variables" is used if this property is not defined.

Chapter 3

Organization of MPCOTool

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 MPCOTool is summarized in the *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, MPCOTool reads the corresponding $N_{experiments} \times N_{inputs}$ 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 another external program, defined in the property *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 (J) associated with the experiments set can be calculated by four different error norms:

$$L_2: \quad J = \sqrt{\sum_{i=1}^{N_{experiments}} |w_i\, o_i|^2}, \tag{3.1}$$

$$L_\infty: \quad J = \max_{i=1}^{N_{experiments}} |w_i\, o_i|\,, \tag{3.2}$$

$$L_p: \quad J = \sqrt[p]{\sum_{i=1}^{N_{experiments}} |w_i\, o_i|^p}, \tag{3.3}$$

$$L_1: \quad J = \sum_{i=1}^{N_{experiments}} |w_i\, o_i|\,, \tag{3.4}$$

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, MPCOTool automatically parallelizes the simulations using all the available computing resources.

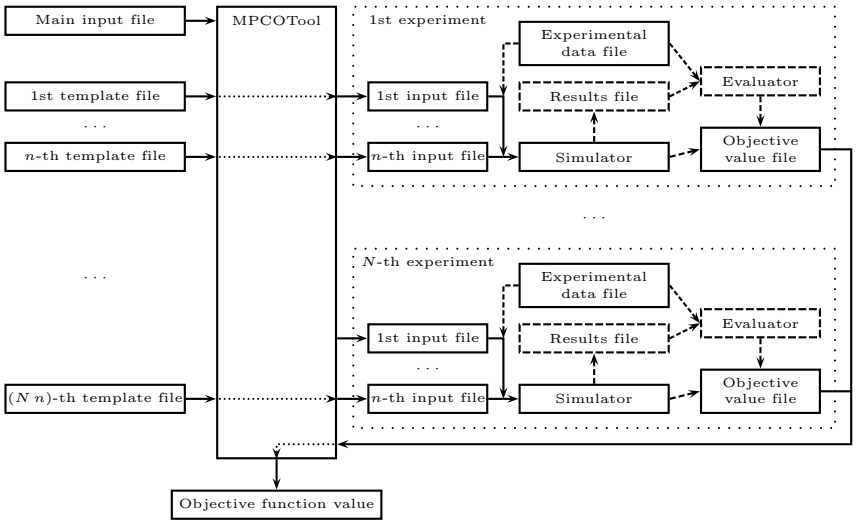

Figure 3.1: Flowchart of the interactions among MPCOTool, the input files and the simulation and evaluation programs to produce an objective function value for each empirical parameters combination generated by the optimization algorithm.

Chapter 4

Optimization methods

The optimization methods implemented in MPCOTool 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 MPCOTool is run.

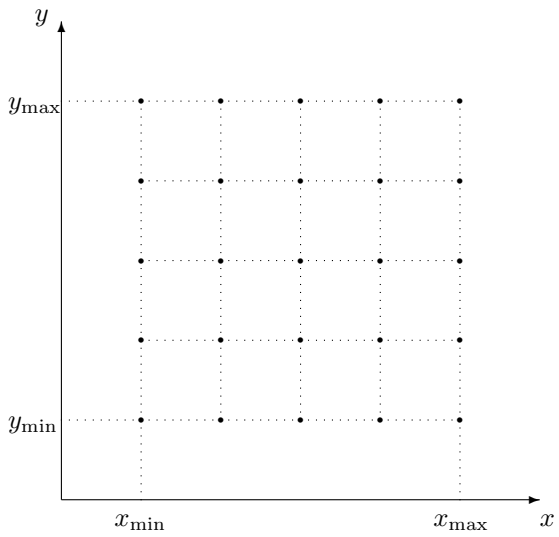

Figure 4.1: Diagram showing an example of application of the sweep brute force method with two variables for $N_x = N_y = 5$.

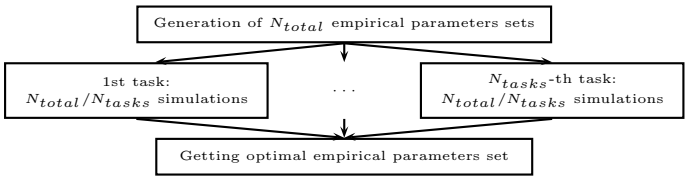

Figure 4.2: Flowchart of the parallelization scheme in MPCOTool for brute force methods (sweep and Monte-Carlo).

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

MPCOTool 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_{best}^j, the subset of the best simulation results in the j-th iteration, the following quantities are defined:

$x_{\max}^b = \max_{i \in N_{best}} x_i^j$: Maximum value of variable x in the subset of the best simulation results from the j-th iteration.

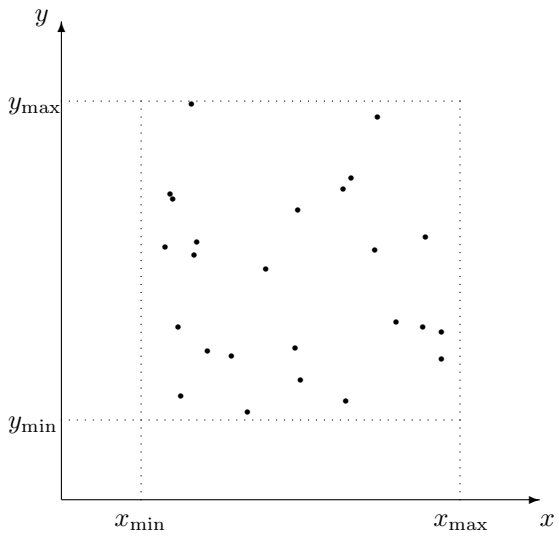

Figure 4.3: Diagram illustrating a Monte-Carlo brute force method with two variables and $N_{simulations} = 25$.

$x^b_{\min} = \max_{i \in N_{best}} x^j_i$: 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 $(j+1)$-th iteration so that:

$$x_i^{j+1} \in \left[x^{j+1}_{\min},\ x^{j+1}_{\max} \right], \tag{4.2}$$

with:

$$\text{Sweep} \ \Rightarrow \ \begin{cases} x^{j+1}_{\max} = x^b_{\max} + \dfrac{x^j_{\max} - x^j_{\min}}{N_x - 1}\, tolerance, \\ x^{j+1}_{\min} = x^b_{\min} - \dfrac{x^j_{\min} - x^j_{\min}}{N_x - 1}\, tolerance, \end{cases}$$

$$\text{Monte} - \text{Carlo} \ \Rightarrow \ \begin{cases} x^{j+1}_{\max} = \dfrac{x^b_{\max} + x^b_{\min} + \left(x^b_{\max} - x^b_{\min}\right)(1 + tolerance)}{2}, \\ x^{j+1}_{\min} = \dfrac{x^b_{\max} + x^b_{\min} - \left(x^b_{\max} - x^b_{\min}\right)(1 + tolerance)}{2}, \end{cases} \tag{4.3}$$

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).

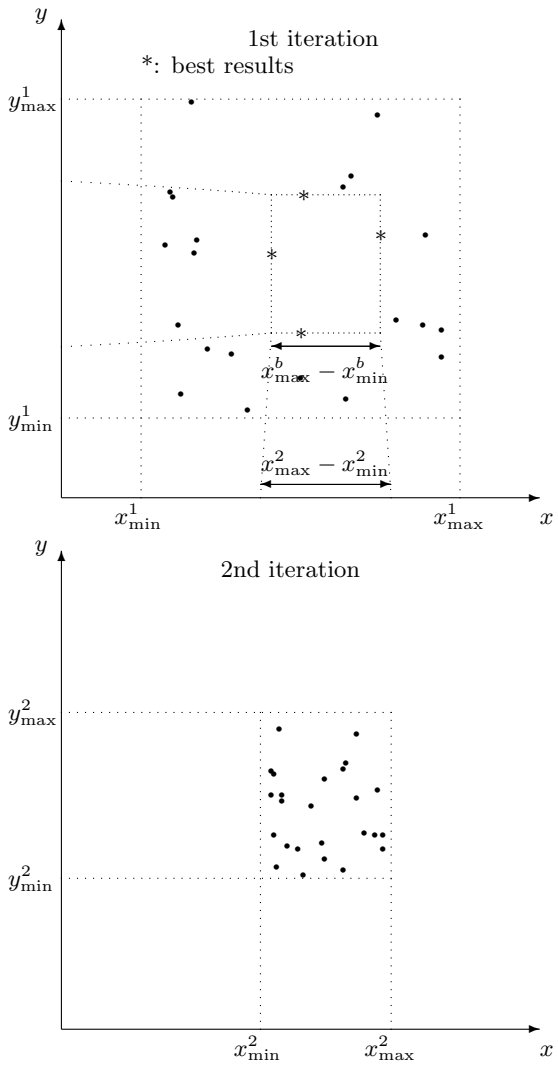

Figure 4.4: Diagram representing an example of the iterative algorithm applied to a Monte-Carlo brute force method with two variables for $N_{simulations} = 25$, $N_{best} = 4$ and two iterations.

4.4 Direction search method

Brute force optimization methods, sweep and Monte-Carlo, can be also combined with a direction search algorithm. Defining the vector $\vec{r}_i$ as the optime variables combination obtained in the i-th step, $\vec{r}_1$ as the optime varaibles combination vector obtained by the brute force method and defining the vector $\vec{s}_i$ as:

$$\vec{s}_1 = \vec{0}, \qquad \vec{s}_i = (1 - relaxation)\,\vec{s}_{i-1} + relaxation\,\Delta\vec{r}_{i-1}, \tag{4.4}$$

with $\Delta\vec{r}_{i-1} = \vec{r}_i + \vec{r}_{i-1}$ and $relaxtion$ the relaxation parameter, the direction search method checks $N_{estimates}$ variable combinations and choice the optimum

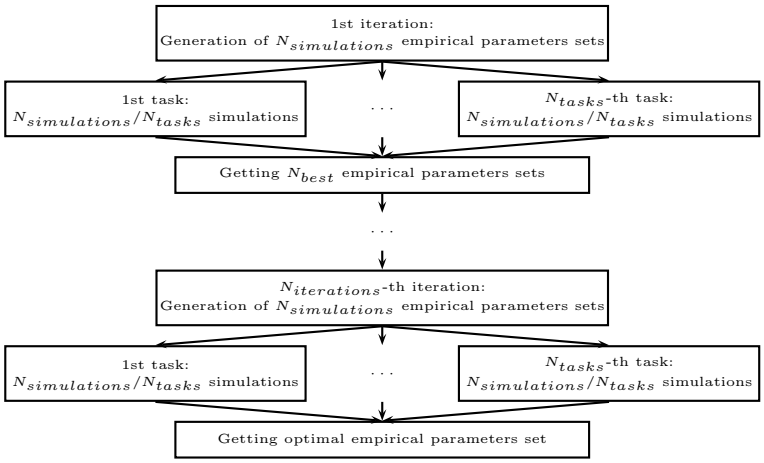

Figure 4.5: Flowchart of the parallelization scheme in MPCOTool for the iterative method.

as:

$$\vec{r}_{i+1} = \text{optime}\left(\vec{r}_i,\ \vec{r}_i + \vec{s}_i + \vec{t}_j\right),\ j = 1, \cdots, N_{estimates}. \tag{4.5}$$

If the step does not improve the optimum ($\vec{r}_i = \vec{r}_{i+1}$) then the direction step vectors $\vec{t}_j$ are divided by two and $\vec{s}_{i+1}$ is set to zero. The method is iterated N_{steps} times.

Although direction search method gets the fastest convergence, is the method in MPCOTool that obtains the least advantages of parallelization. The method is almost sequential and parallelization only can be performed for each step in the $N_{estimates}$ simulations to estimate the optimal direction. In the figure 4.6 a flowchart of the parallelization scheme for this method is shown.

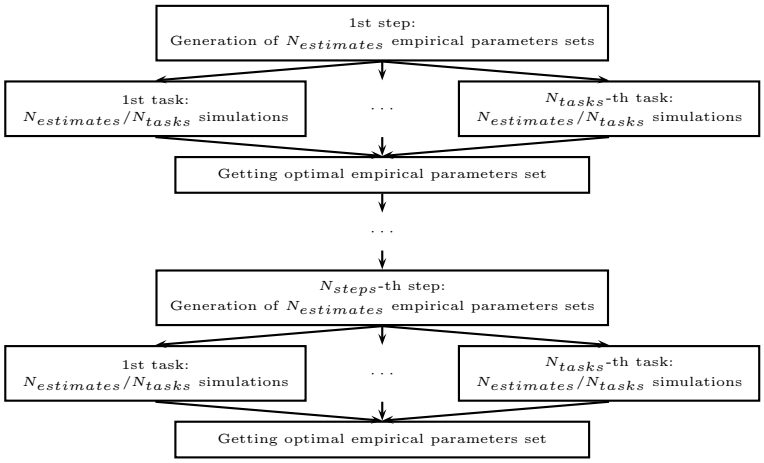

Figure 4.6: Flowchart of the parallelization scheme in MPCOTool for the direction search method.

MPCOTool uses two methods to build the $\vec{t}_j$ vectors:

4.4.1 Coordinates descent

This method builds the $\vec{t}_j$ vectors by increasing or decreasing only one variable:

$$\vec{t}_1 = \begin{pmatrix} step_1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \vec{t}_2 = \begin{pmatrix} -step_1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \vec{t}_3 = \begin{pmatrix} 0 \\ step_2 \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

$$\vec{t}_4 = \begin{pmatrix} 0 \\ -step_2 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \cdots \quad , \vec{t}_{N_{estimates}} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ -step_{N_{variables}} \end{pmatrix}, \tag{4.6}$$

being $step_j$ the initial step size for the j-th variable defined by the user in the main input file. The number of estimates in this method depends on the variables number:

$$N_{estimates} = 2\, N_{variables} \tag{4.7}$$

4.4.2 Random

The vectors $\vec{t}_j$ are built randomly as:

$$\vec{t}_j = \begin{pmatrix} (1 - 2\, r_{j,1})\ step_1 \\ \vdots \\ (1 - 2\, r_{j,k})\ step_k \\ \vdots \\ (1 - 2\, r_{j,N_{variables}})\ step_{N_{variables}} \end{pmatrix}, \tag{4.8}$$

with $r_{j,k} \in [0, 1)$ random numbers.

In the figure 4.7 a sketch for a system with two variables is presented to illustrate the working mode of coordinates descent and random algorithms.

4.5 Genetic method

MPCOTool also offers the use of a genetic method Genetic [2017] with its default algorithms. It is inspired on the ideas in GAUL [2017], but it has been fully reprogrammed involving more modern external libraries. The code in Genetic is also open source under BSD license. Figure 4.8 shows the flowchart of the genetic method implemented in Genetic.

4.5.1 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 4.9 shows an

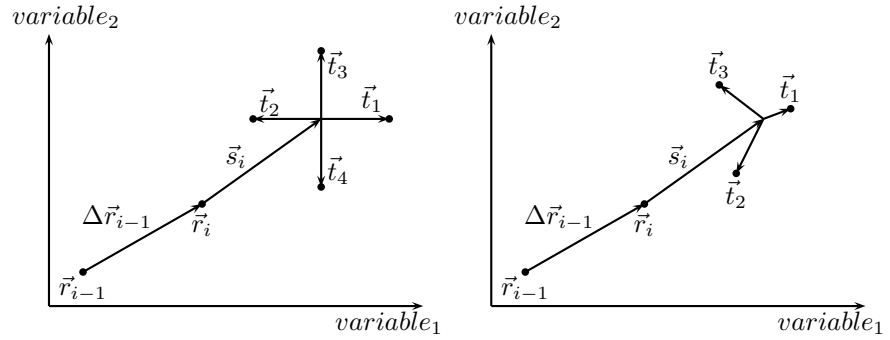

Figure 4.7: (Left) coordinates descent and (right) random with $N_{estimates} = 3$ direction search method checks of variables combination in a system with two variables.

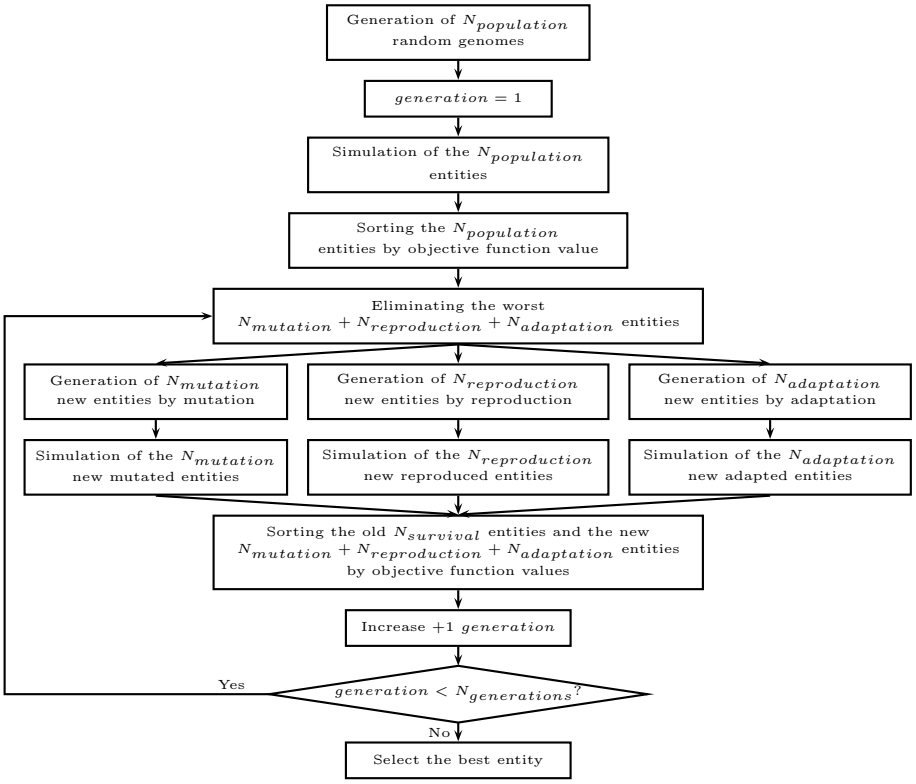

Figure 4.8: Flow diagram of the genetic method implemented in Genetic.

example for the coding of three variables. The value assigned to a variable x is determined by the allowed extreme values $x_{\min}$ and $x_{\max}$, the binary number assigned in the genome to variable I_x and by the number of bits assigned to variable N_x according to the following formula:

$$x = x_{\min} + \frac{I_x}{2^{N_x} - 1} \left(x_{\max} - x_{\min}\right). \tag{4.9}$$

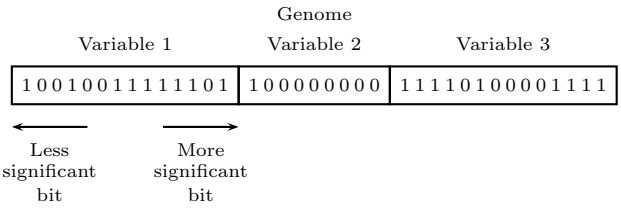

Figure 4.9: Example coding three variables to optimize into a genome. The first and third variables have been coded with 14 bits, and the second variable has been coded with 9 bits.

4.5.2 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:

$$N_{new} = N_{population} \times (R_{mutation} + R_{reproduction} + R_{adaptation}) . \tag{4.10}$$

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

$$N_{total} = N_{population} + (N_{generations} - 1) \times N_{new}, \tag{4.11}$$

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:

$$N_{survival} = N_{population} - N_{new}. \tag{4.12}$$

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:

$$R_{mutation} + R_{reproduction} + R_{adaptation} < 1 \tag{4.13}$$

MPCOTool uses a default aleatory criterion in Genetic, with a probability linearly decreasing with the ordinal in the ordered set of surviving individuals (see figure 4.10).

4.5.3 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 4.11 shows an example of the procedure.

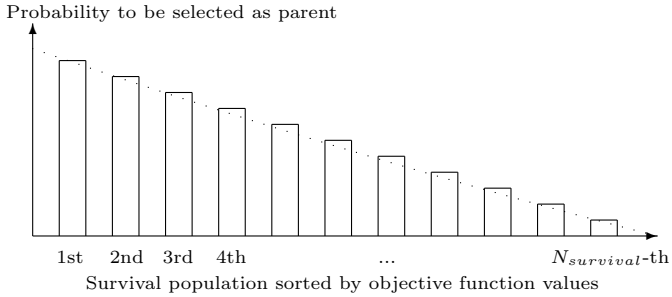

Figure 4.10: Probability of a survival entity to be selected as parent of the new entities generated by mutation, reproduction or adaptation algorithms.

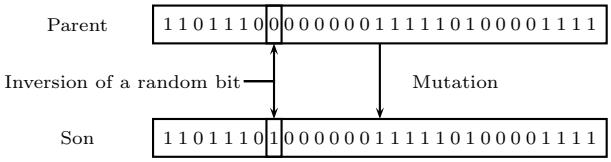

Figure 4.11: Diagram showing an example of the generation of a new entity by mutation.

4.5.4 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 4.12 shows a sketch of the algorithm.

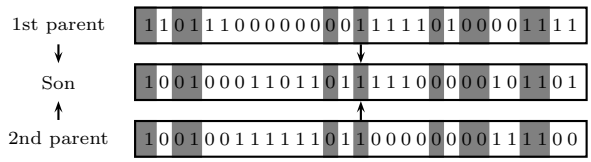

Figure 4.12: Example of the generation of a new entity by reproduction in the Genetic default algorithm. Note that the identical bits in both parents (in grey) are also present in their son. The rest of the bits are random.

4.5.5 Adaptation algorithm

Another algorithm is included in Genetic called "adaptation" although, 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 4.13 contains an example.

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

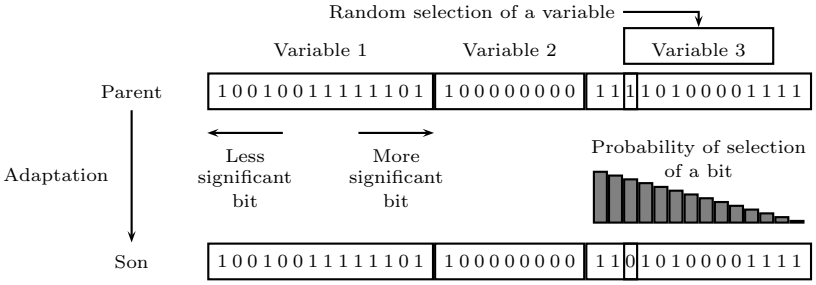

Figure 4.13: Example of the generation of a new individual from a parent by adaptation.

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

4.5.6 Parallelization

This method is also easily parallelizable following a similar scheme to the iterative algorithm, as it can be seen in the figure 4.14.

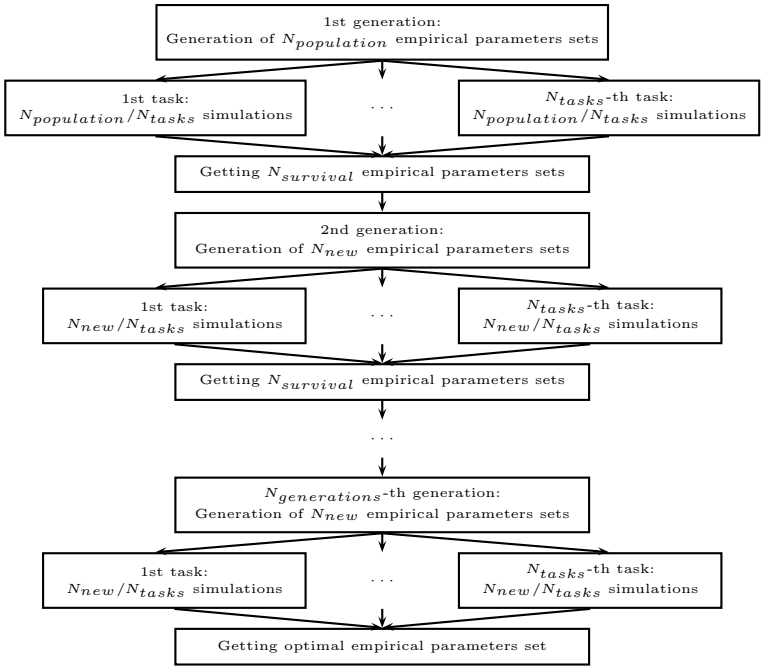

Figure 4.14: Flowchart of the parallelization scheme implemented in Genetic for the genetic method.

References

Autoconf. Autoconf is an extensible package of M4 macros that produce shell scripts to automatically configure software source code packages. `https://www.gnu.org/software/autoconf`, 2017.

Automake. Automake is a tool for automatically generating Makefile.in files compliant with the GNU coding standards. `https://www.gnu.org/software/automake`, 2017.

Clang. A C language family frontend for LLVM. `http://clang.llvm.org`, 2017.

GAUL. The genetic algorithm utility library. `http://gaul.sourceforge.net`, 2017.

GCC. The GNU compiler collection. `https://gcc.gnu.org`, 2017.

Genetic. Genetic: a simple genetic algorithm. `https://github.com/jburguete/genetic`, 2017.

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. `https://developer.gnome.org/glib`, 2017.

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. `http://www.gnu.org/software/make`, 2017.

GSL. GNU scientific library. `http://www.gnu.org/software/gsl`, 2017.

GTK+3. GTK+, or the GIMP toolkit, is a multi-platform toolkit for creating graphical user interfaces. `http://www.gtk.org`, 2017.

Install-UNIX. A set of Makefiles to install some useful applications on different UNIX type operative systems. `https://github.com/jburguete/install-unix`, 2017.

JSON-GLib. A JSON reader and writer library using GLib and GObject. `https://github.com/ebassi/json-glib`, 2017.

Libxml. The XML C parser and toolkit of GNOME. `http://xmlsoft.org`, 2017.

MPICH. High-performance portable MPI. `http://www.mpich.org`, 2017.

OpenMPI. Open source high performance computing. http://www.open-mpi.org, 2017.

Pkg-config. Pkg-config is a helper tool used when compiling applications and libraries. http://www.freedesktop.org/wiki/Software/pkg-config, 2017.