First steps to solve MINLP problems with Muriqui Optimizer

August 5, 2020

Wendel Melo
College of Computer Science
Federal University of Uberlandia, Brazil
wendelmelo@ufu.br

Marcia Fampa
Institute of Mathematics and COPPE
Federal University of Rio de Janeiro, Brazil
fampa@cos.ufrj.br

Fernanda Raupp
National Laboratory for Scientific Computing (LNCC)
Ministry of Science, Technology and Innovation, Brazil
fernanda@lncc.br

ABSTRACT

Muriqui Optimizer is a software to solve Mixed Integer Nonlinear Programming (MINLP) problems. Muriqui includes an executable file to be used with the AMPL and/or GAMS systems and a powerful C++ library for integrating with various software. The main differential of Muriqui is that it brings the possibility of using the main MINLP algorithms of the literature, with a series of customizable parameters. This text provides general instructions for obtaining, installing and using Muriqui.

1 What is the Muriqui Optimizer?

Muriqui Optimizer is a solver of convex Mixed Integer Nonlinear Programming (MINLP) problems. Algebraically, Muriqui Optimizer solves problems in the format:

$$
\begin{array}{lll}
(P) \quad z := \min\limits_{x} & f(x) + \frac{1}{2}x'Q_0x + c'x + d, & \\
\text{subject to:} \quad l_{c_i} \leq & g_i(x) + \frac{1}{2}x'Q_ix + a_ix & \leq u_{c_i}, \quad i = 1, \ldots, m, \\
l_x \leq & x & \leq u_x, \\
& x_j \in \mathbb{Z}, \text{ for } j \in \mathcal{I}, &
\end{array}
\tag{1}
$$

Note that x denotes the vector with the decision variables. The vectors l_x and u_x bring lower and upper bounds for x, and $\mathcal{I}$ is the set of indexes of variables constrained to integer

values. Concerning the objective function, $f(x)$ represents a nonlinear term, Q_0 is a symmetric matrix used to describe the quadratic term, c is a column vector used to describe the linear term and d is a constant term. For each constraint i, $i = 1, \ldots, m$, $g_i(x)$ represents a nonlinear term, Q_i is a symmetric matrix used to describe a quadratic term, a_i is a row vector used to describe a linear term and l_{c_i} and u_{c_i} are lower and upper bounds, respectively.

Certainly, linear and quadratic terms can be represented in the functions $f(x)$ and $g_i(x), i = 1, \ldots, m$. Nevertheless, the decomposition in linear, quadratic and general nonlinear terms aims to make possible a better use of the characteristics of problem (P). In addition, some available nonlinear programming (NLP) solvers can take advantage of this decomposition to solve relaxations of problem (P) more quickly.

Also, we point out that some of the elements in (P) may be "missing". For example, some components of the vectors l_c or l_x can be defined as $-\infty$, while some of the components of the vectors u_c or u_x can be defined as $+\infty$, some of the functions $g_i(x)$ or $f(x)$ can be null, as well as the matrices Q_i and the vectors c and a_i.

Muriqui has been developed to solve convex problems. However, non-convex problems can also be addressed, without the guarantee of obtaining optimal solution. Ideally, the functions $f(x)$ and $g_i(x)$ must be doubly differentiable and evaluated at any point x that satisfies the box constraints (1), but it may be possible, in some cases, to circumvent this last requirement with approximations for the derivatives.

Muriqui brings the idea that a generic configuration may not be the best choice in a specific application. Thus, a remarkable feature of Muriqui is the wide range of customizable options available. Therefore, users have a wide range of choices of applications including the choice of the algorithm to be used, the parameters setting of these algorithms, and the solvers used to handle the generated subproblems. From this perspective, users are encouraged to pass default settings and test different configurations to find the best option for each specific situation.

2 Who are we?

We are a group of researchers interested in the MINLP optimization area. As part of our work, we developed our own solver called Muriqui, whose initial goal was to only support our research. Nevertheless, we would be glad if our work had applicability to other MINLP researchers or practitioners. We are looking forward to receiving feedback, even if they are bad reports in other to improve it.

3 License and terms of use

For the time being, Muriqui is a totally free and open MINLP solver, and can be used through the MIT license. We kindly ask the users to cite our paper [12] describing the implemented algorithms on the documents where they report its use. We also ask the users to have some patience when using Muriqui, especially the first versions, that may contain some bugs. We will be very grateful if the users can report to us errors and bugs found.

It is important to note that, although Muriqui is a free software, it is not self-contained. This means that the installation of other software and libraries are required for its correct operation (see Section 4). Each of these software may have its own usage and license policies. For proper use of Muriqui, these software must be installed under the responsibility of the user.

4 Required software

If the user intends to install Muriqui using its source code, he/she will initially need a C++ 2011 compiler. On our tests, we have succeeded with distinct versions of compilers GCC and ICPC in Linux systems. In order to take full advantage of the features offered, it is recommended to install: the AMPL Solver Library (ASL), at least one solver for Mixed Linear Programming (MILP) problems, and at least one solver for Nonlinear Programming (NLP) problems. In the following, we include some instructions for these installations.

4.1 AMPL Solver Library

The AMPL Solver Library (ASL) is necessary to integrate Muriqui with the AMPL system. Once it is installed, it is possible to generate an executable file of Muriqui capable of reading and processing models in the nl format (compiled models generated from AMPL). However, if your goal is to use Muriqui with your C++ API, you do not need to install this library. Nevertheless, we strongly recommend the use of the built-in ASL executable whenever possible, since the AMPL language greatly facilitates the writing and manipulation of optimization models, besides avoiding derivative coding. But if you still have to opt for using the API, for example, to integrate another software project, you should be aware that you will need to correctly code the first and second order derivatives of each nonlinear term $f(x)$ and $g_i(x)$.

4.1.1 Installing the ASL

The ASL can be downloaded at `http://www.netlib.org/ampl/solvers.tgz`. After unzipping the downloaded file, go into the folder generated and run the configuration script with the command:

```
> configure
```

In order to enable function evaluations in parallel, go to the generated directory and run the command:

```
make arith.h
```

and add the following line to the file `arith.h`

```
#define ALLOW_OPENMP
```

Adding the line above is sufficient to enable ASL in multithreading mode. However, alternatively, you can define by your own the functions that should guarantee mutual exclusion in ASL (see [8] for more details).

Finally, run the "make" command. It is worth mentioning that it is always good to perform an optimized compilation for performance gain purposes. So if you can, add optimization flags for the compiler in the make file, such as "-O3" and/or "-march=native". If you are not familiar to these flags or cannot add them for some reason, please ignore this comment. Run:

```
> make
```

If the execution succeeded and you have not changed any default options, a file called "amplsolver.a" should be generated. To conform to the default library names, we recommend that you rename this file to "libamplsolver.a".

4.2 MILP Solvers

To run some MINLP algorithms in Muriqui, it is necessary to solve Mixed Integer Linear Programming (MILP) subproblems that can be generated. As solving MILP problems is outside the scope of Muriqui, the installation of at least one of the following supported MILP solvers is required:

- Cplex (most recommended);
- Gurobi (recommended);
- Xpress (midterm recommended);
- Mosek (little recommended);
- Cbc (little recommended);
- GLPK (less recommended).

In most of our tests, we have used Cplex to solve the MILP subproblems generated by Muriqui. For this reason, the chances of some interface bug with Cplex are lower. This is the main reason why this is the most recommended solver, besides the fact that a license for academic use is easy to obtain. However, if you believe that Gurobi or Xpress may work best for you, you are encouraged to try them out. If obtaining a commercial software license is out of your reach or desire, you still have, as a last alternative, the option of using GLPK, which is a free solver. However, GLPK is practically not recommended because it is not thread safe, which can hinder the execution in some contexts. If you are aware that GLPK has become thread safe, please let me know and we will upload its recommendation rating.

After choosing one of the supported MILP solvers, you should perform the installation according to the instructions given by its developers. After installation, you should be able to generate a C executable program using the API of the solver chosen to proceed with the requirements for the Muriqui installation (try compiling and running the codes in C/C++ of the examples distributed with the software chosen).

4.3 NLP Solvers

To run some MINLP algorithms in Muriqui, it is necessary to solve Nonlinear Programming (NLP) subproblems that can be possibly generated. As solving NLP problems is also outside the scope of Muriqui, the installation of at least one of the following supported NLP solvers is required:

- Mosek (if chosen, at least one of the following solvers should be chosen as well);
- Ipopt;
- Knitro;

We observe that we have a slight preference for Mosek, because it is able to handle more easily the separable terms of problem (P), and because it is easy to obtain an academic license for it. However, you are welcome to try out the other solvers, especially if you have a Knitro

license. For those who do not want or cannot use commercial software, the option indicated is the free solver Ipopt, which may, in some cases, even work better than the others. It is important to note that if you choose Mosek, we still recommend installing Ipopt, as Mosek do not address non-convex problems (not even to search for local optimal). So if the user is interested in addressing non-convex MINLP problems, Ipopt or Knitro are the proper choices.

After choosing one or more of the supported solvers, the user should install the solvers as instructed by their developers. After the installation, the user should be able to generate an executable program in C using the API of the solver chosen to proceed with the requirements for the Muriqui installation (try compiling and running the codes in C/C++ of the examples distributed with the software chosen).

5 Installing Muriqui Optimizer

5.1 Downloading Muriqui Optimizer

The source code of Muriqui can be downloaded from `http://wendelmelo.net/software`. Before proceeding, it is necessary to install the software required for the correct use of Muriqui, as explained in Section 4.

5.2 Compiling Muriqui Optimizer

After downloading and unzipping Muriqui into a folder of your preference, referred to here as $MURIQUIDIR, the next step is to configure the compilation to use the options and libraries chosen. This configuration is done by editing the files WAXM_config.h and make.inc, present in $MURIQUIDIR, in an integrated and consistent way, as explained below:

5.2.1 Editing the WAXM_config.h file

The WAXM_config.h file contains definitions for the correct compilation of Muriqui. The flags indicate the presence of specific libraries and functions to be used. For example, to compile Muriqui integrated to the AMPL system through the ASL library, after installing ASL following the steps in Section 4.1, the user should edit the line:

```
#define WAXM_HAVE_ASL                    0
```

changing it to:

```
#define WAXM_HAVE_ASL                    1
```

The user should also edit the lines corresponding to the MILP and NLP solvers used. For example, if the user wants to compile Muriqui with Cplex, Gurobi, Mosek and Ipopt, he/she should edit the lines:

```
#define WAXM_HAVE_CPLEX                  0
#define WAXM_HAVE_GUROBI                 0
#define WAXM_HAVE_MOSEK                  0
#define WAXM_HAVE_IPOPT                  0
```

changing them to:

```
#define WAXM_HAVE_CPLEX                    1
#define WAXM_HAVE_GUROBI                   1
#define WAXM_HAVE_MOSEK                    1
#define WAXM_HAVE_IPOPT                    1
```

Note that it is possible to compile Muriqui with several MILP and NLP solvers simultaneously, allowing the user to choose among these software only when applying the algorithms implemented in Muriqui.

Other flags are: `WAXM_HAVE_CLOCK_GETTIME`, which indicates the presence of the C function clock_gettime (usually present in Unix systems); `WAXM_HAVE_POSIX`, which indicates the presence of libraries described in the Posix standard (also present in Unix systems).

5.2.2 Editing the make.inc file

The make.inc file contains settings for the compilation of Muriqui, such as the compiler to be used, compilation flags, and the location of required inclusion files and libraries. Please note that special attention must be paid when editing this file so that it conforms to WAXM_config.h. For example, if WAXM_config.h contains the definition `#define WAXM_HAVE_CPLEX  1`, which means that Muriqui is configured to be compiled with Cplex. Therefore, the make.inc file should contain the definition of the CPLEXINC variable with the inclusion definitions and CPLEXLIB with the library link definitions. By default, make.inc contains commented lines with the definition of these variables, so it is possible to uncomment the respective lines and make the necessary changes in order to enable the correct compilation. For example, assuming that Cplex is installed in the $CPLEXDIR folder, a possible setting for these variables would be:

```
CPLEXINC = -I${CPLEXDIR}/include/ilcplex/
CPLEXLIB = -L${CPLEXDIR}/lib/x86-64_linux/static_pic/ -lcplex -lpthread -pthread
```

Note that the above definitions may vary depending on the system settings. The user should make similar definitions for each software library that is set to 1 in WAXM_config.h, including the ASL. In our example in Section 5.2.1, we configure Muriqui to be compiled also with Gurobi, Mosek and Ipopt. Therefore, the user would also need to define the variables $GUROBIINC, $GUROBILIB, $MOSEKINC, $MOSEKLIB, $IPOPTINC, and $IPOPTLIB, in the make.inc file (the user can uncomment the respective lines and make the necessary edits). In most cases, to integrate ASL in the Muriqui compilation, it should be sufficient to follow the steps described in Section 4.1, change the definition of `WAXM_HAVE_ASL` to 1 (Section 5.2.1), and change the line in make.inc with the definition of the $ASLDIR variable, with the value of the folder where the libamplsolver.a file is located.

5.2.3 Enabling integration with GAMS system

To enable integration with GAMS system, follow these steps:

- Download and install GAMS;
- Add GAMS installation directory to the system PATH and to the dynamic libraries PATH (LD_LIBRARY_PATH);

- In the WAXM_config.h file, edit the line:

  ```
  #define WAXM_HAVE_GAMS 0
  ```

  to:

  ```
  #define WAXM_HAVE_GAMS 1
  ```

- Edit the make.inc file, uncommenting the rows related to compiling with GAMS and changing the definition line of the GAMSDIR variable so that it contains the GAMS installation directory, for example:

  ```
  #GAMS variables to compilation.
  GAMSDIR = /opt/gams/
  GAMSINC = -I$(GAMSDIR)/apifiles/C/api/
  GAMSLIB = $(GAMSDIR)/apifiles/C/api/gmomcc.c
  ```

After performing the previous steps, the Muriqui compilation will enable integration with GAMS. However, to use Muriqui within GAMS, you must still configure it by following the steps in Section 6.3.

5.3 Running the Makefile

Once the user configured the compilation as described in the previous subsections, he/she is ready to run the program make in $MURIQUIDIR by typing:

```
> make
```

If the configuration was correctly done, the executable file muriqui and the library libmuriqui are generated. {a, lib}. The executable muriqui is built to be used in an integrated way to the AMPL system or to read .nl files (compiled AMPL models). Therefore, it only makes sense to use it if the ASL library has been included in the compilation. The library libmuriqui contains definitions and routines available in the Muriqui API. To make sure that the compilation has succeeded, the user may run the examples distributed with the source code of Muriqui (Section 6).

If the compilation was not successful, the user may try editing the variable definitions in make.inc or WAXM_config.h. The user may feel free to also ask for help on the Muriqui mailing list. We will try to assist him/her in this task as much as possible.

6 Running examples

6.1 Via AMPL system

The folder $MURIQUIDIR/examples contains examples of using of Muriqui with AMPL system, through files with extension .mod. You can navigate to this folder and choose one of the examples to investigate, for example, $MURIQUIDIR/examples/t1gross/t1gross.mod:

```
var x1 >= 0 <= 2;  # Continuous variable
var x2 >= 0 <= 2;  # Continuous variable
var x6 >= 0 <= 1; # Continuous variable
var y1 binary;  # Integer variable
var y2 binary;  # Integer variable
var y3 binary;  # Integer variable

param U;

minimize fcObj: 5*y1 +6*y2 +8*y3 +10*x1 -7*x6 -18*log(x2 +1) -19.2*log(x1 -x2 +1) +10;

subject to g1: 0.8*log(x2 + 1) + 0.96*log(x1 - x2 + 1) - 0.8*x6 >= 0;
subject to g2: x2 - x1 <= 0;
subject to g3: x2 - U*y1 <= 0;
subject to g4: x1 - x2 - U*y2 <= 0;
subject to g5: log(x2 + 1) + 1.2*log(x1 - x2 + 1) - x6 - U*y3 >= -2;
subject to g6: y1 + y2 <= 1;

data;
param U := 2;

#In this example, we solve this problem by several different algorithms.
#Note, you will probably need solve your models just once choosing one of the
#available algorithms.

option solver "muriqui";

#solving model by lp-nlp BB algorithm
options muriqui_alg_choice "str MRQ_LP_NLP_BB_OA_BASED_ALG";
#options muriqui_options "str in_milp_solver MRQ_GUROBI
#                         str in_nlp_solver MRQ_NLP_MOSEK
#                         dbl in_max_cpu_time 600
#                         int in_max_iterations 1000000000";
solve;

#solving model by outer approximation algorithm
options muriqui_alg_choice "str MRQ_OA_ALG";
solve;

#solving model by extended cutting plane algorithm
options muriqui_alg_choice "str MRQ_ECP_ALG";
solve;

#solving model by extended supporting hyperplane algorithm
options muriqui_alg_choice "str MRQ_ESH_ALG";
solve;

#solving model by bonmin hybrid algorithm
options muriqui_alg_choice "str MRQ_BONMIN_HYBRID_ALG";
solve;

#solving model by branch-and-bound algorithm (hybriding with outer approximation)
```

```
options muriqui_alg_choice "str MRQ_BB_ALG";
solve;

#solving model by pure branch-and-bound algorithm
options muriqui_alg_choice "str MRQ_BB_ALG";
options muriqui_options "int in_use_outer_app 0
int in_use_outer_app_as_heuristic  0";
solve;
```

The file t1gross.mod shows a MINLP model. After the definition of the model, we need to specify the use of Muriqui with the following line.

```
option solver "muriqui";
```

For AMPL to find the Muriqui executable file correctly, it is necessary to include $MURIQUIDIR in the path of the system. The next step is to specify one of the algorithms available in Muriqui to solve the model. This can be done adjusting the option *muriqui_alg_choice* to "str <algorithm>", where <algorithm> must be a valid constant that refers to an algorithm in Muriqui (see Section 7.3). For example, to use Outer Approximation, we use the following line.

```
options muriqui_alg_choice "str MRQ_OA_ALG";
```

Through the option *muriqui_options* one can pass parameters to the algorithm. The parameters are passed by a string composed of trios of the type "<type> <param name> <param value>", where <type> can be *int* (integer number), *dbl* (real number) or *str* (string). In a similar way, one can pass parameters to the solvers of MILP and NLP that were adopted through the options *muriqui_milp_options* and *muriqui_nlp_options*, respectively. The following AMPL code specifies parameters to be passed to the MILP solver (Cplex) and NLP solver (Ipopt) used by Muriqui.

```
#seting muriqui algorithm options, setting CPLEX as MILP solver
#and IPOPT as in_nlp_solver
option muriqui_options "str in_milp_solver MRQ_CPLEX
str in_nlp_solver MRQ_IPOPT ";

#seting options to milp solver (CPLEX)
option muriqui_milp_options "int CPX_PARAM_THREADS 1
dbl CPX_PARAM_TILIM 100";

#seting options to nilp solver (IPOPT)
option muriqui_nlp_options  "int max_cpu_time 5000
str linear_solver  pardiso";
```

6.2 Via AMPL nl files

The file t4gross.nl present in $MURIQUIDIR contains the compilation of an AMPL model. To run it, the user should call the executable muriqui compiled with ASL, passing the name of the file as a parameter:

```
> ./muriqui t4gross.nl
```

```
------------------------------------------------------------------------------
Muriqui MINLP solver, version 0.7.00 compiled at Dec  2 2017 20:40:39
by Wendel Melo, Computer Scientist, Federal University of Uberlandia, Brazil
collaborators: Marcia Fampa (UFRJ, Brazil), Fernanda Raupp (LNCC, Brazil)

if you use, please cite:
W Melo, M Fampa & F Raupp. Integrating nonlinear branch-and-bound and outer
approximation for convex Mixed Integer Nonlinear Programming. Journal of Global
Optimization, v. 60, p. 373-389, 2014.
------------------------------------------------------------------------------

muriqui: parameter 1: t4gross.nl
muriqui: input file: t4gross.nl
muriqui: Reading user model:
muriqui: Maximization problem addressed as minimization
muriqui: Done
muriqui: Reading user parameters:
muriqui: Trying reading algorithm choice file muriqui_algorithm.opt . Not found!
muriqui: Done
muriqui: Trying read input parameter file muriqui_params.opt
muriqui: Done
muriqui: preprocessing...

Starting Branch-and-Bound algorithm

muriqui: milp solver: cplex, nlp solver: mosek
iter lb  ub  gap nodes at open

1  -13.5901 -5.32374 8.26636 2 0
muriqui: Appliyng Outer Approximation on the root
cpu time: 1.03897
cpu time: 1.04079
------------------------------------------------------------------------------
Problem: t4gross.nl Algorithm: 1004 Return code: 0 obj function: -8.0641361104
time: 0.70 cpu time: 1.04 iters: 21
------------------------------------------------------------------------------
An optimal solution was found! Obj function: 8.064136. CPU Time: 1.040791
```

Note that the output of each execution may vary a little, depending on the system and on the configuration.

6.3 Via GAMS

Muriqui may also receive MINLP models from GAMS system. For this, it is first necessary to compile Muriqui by enabling integration with GAMS, as described in Section 5.2.3. After that, you must configure GAMS to recognize Muriqui as the MINLP solver. To do this, you need to perform the following steps (UNIX):

1. Copy the files gmsmq_us.run and gmsmq_ux.out of Muriqui directory to the GAMS installation directory and make them executable;

2. Edit the file gmscmpun.txt (in the GAMS installation directory) adding the following lines:

```
MURIQUI 2111 5 mq 1 0 1 MIP QCP MIQCP RMIQCP NLP RMINLP MINLP
gmsmq_us.run
gmsmq_ux.out
```

The above steps may be slightly different on non-UNIX systems. Please contact support of GAMS for any questions or difficulties. Having successfully completed the above steps, you can solve a MINLP model using GAMS by setting your MINLP variable with the value `muriqui`, for example:

```
* GAMS model file ex1223.gms from MINPLib

Variables  x1,x2,x3,x4,x5,x6,x7,b8,b9,b10,b11,objvar;
Positive Variables  x1,x2,x3,x4,x5,x6,x7;
Binary Variables  b8,b9,b10,b11;
Equations  e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11,e12,e13,e14;

e1..    x1 + x2 + x3 + b8 + b9 + b10 =L= 5;
e2.. sqr(x6) + sqr(x1) + sqr(x2) + sqr(x3) =L= 5.5;
e3..    x1 + b8 =L= 1.2;
e4..    x2 + b9 =L= 1.8;
e5..    x3 + b10 =L= 2.5;
e6..    x1 + b11 =L= 1.2;
e7.. sqr(x5) + sqr(x2) =L= 1.64;
e8.. sqr(x6) + sqr(x3) =L= 4.25;
e9.. sqr(x5) + sqr(x3) =L= 4.64;
e10..    x4 - b8 =E= 0;
e11..    x5 - b9 =E= 0;
e12..    x6 - b10 =E= 0;
e13..    x7 - b11 =E= 0;
e14.. -(sqr((-1) + x4) + sqr((-2) + x5) + sqr((-1) + x6) - log(1 + x7)
+ sqr((-1) + x1) + sqr((-2) + x2) + sqr((-3) + x3)) + objvar =E= 0;

* set non-default bounds
x1.up = 10; x2.up = 10; x3.up = 10; x4.up = 1;
x5.up = 1;  x6.up = 1;  x7.up = 1;

Model m / all /;

* HERE, WE DEFINE MURIQUI LIKE THE MINLP SOLVER
option minlp = muriqui;

$if NOT '%gams.u1%' == '' $include '%gams.u1%'

$if not set MINLP $set MINLP MINLP
Solve m using %MINLP% minimizing objvar;
```

The Muriqui parameter setting can be done through the following files that are present in the current directory in the execution of the GAMS interpreter:

- muriqui_algorithm.opt: defines the algorithm to be used by Muriqui;
- muriqui_params.opt: defines parameter values of Muriqui algorithms;
- muriqui_milp_params.opt: defines parameter values for the MILP solver used by Muriqui;
- muriqui_nlp_params.opt: defines parameter values for the NLP solver used by Muriqui;

For details on using these files, see Section 10.

6.4 Via API

The folder $MURIQUIDIR/examples contains examples of use of the Muriqui API in C++. The user may navigate to one of them, for example, $MURIQUIDIR/examples/t1gross and run make from this point. If the execution succeeds, the executable t1gross should be generated, which solves an example using several MINLP algorithms implemented in Muriqui:

```
> make
...
> ./t1gross
```

7 Implemented algorithms

For the time being, the following MINLP algorithms are implemented in Muriqui Optimizer and available for use:

7.1 Exact

- LP based Branch-and-Bound (LP-BB) [13]
- LP/NLP based Branch-and-Bound (LP/NLP-BB) [14];
- Hybrid Outer Approximation Branch-and-Bound (HOABB) [10];
- Outer Approximation (OA) [6, 7];
- Extended Cutting Plane (ECP) [15];
- Extended Supporting Hyperplane (ESH) [9];
- Bonmin Hybrid Branch-and-Bound (BHBB) [2].

We note that the application of a pure Branch-and-Bound algorithm can also be performed by choosing the HOABB algorithm and disabling the subprocedures of outer approximation.

7.2 Heuristics

- Feasibility Pump (FP) [4];
- Diving heuristic [4];
- Outer Approximation based Feasibility Pump (OAFP) [3];
- Relaxation Enforced Neighborhood Search (RENS) [1];
- Integrality Gap Minimization Heuristic 1 (IGMA1) [11];
- Integrality Gap Minimization Heuristic 2 (IGMA2), [11].

7.3 Constants and Classes of Algorithms

Table 1 shows the constant used in Muriqui to assign each algorithm, with the respective API class in C++ that implements it.

Table 1: Constants and C++ classes of algorithms in Muriqui

Initials	**Type**	**Muriqui Constant**	**API Class**
LP-BB	Exact	MRQ_LP_BB_ECP_BASED_ALG	MRQ_LPBBExtCutPlan
LP/NLP-BB	Exact	MRQ_LP_NLP_BB_OA_BASED_ALG	MRQ_LPNLPBBOuterApp
HOABB	Exact	MRQ_BB_ALG	MRQ_BranchAndBound
OA	Exact	MRQ_OA_ALG	MRQ_OuterApp
ECP	Exact	MRQ_ECP_ALG	MRQ_ExtCutPlan
ESH	Exact	MRQ_ESH_ALG	MRQ_ExtSupHypPlane
BHBB	Exact	MRQ_BONMIN_HYBRID_ALG	MRQ_BonminHybrid
FP	Heuristic	MRQ_FP_HEUR_ALG	MRQ_FeasibilityPump
Diving	Heuristic	MRQ_DIVE_HEUR_ALG	MRQ_Diving
OAFP	Heuristic	MRQ_OA_FP_HEUR_ALG	MRQ_OAFeasibilityPump
IGMA1	Exact/Heuristic	MRQ_IGMA1_ALG	MRQ_IGMA1
IGMA2	Heuristic	MRQ_IGMA2_ALG	MRQ_IGMA2
RENS	Heuristic	MRQ_RENS_HEUR_ALG	MRQ_RENS
CR	Continuous Relax	MRQ_CONT_RELAX_ALG	MRQ_ContinuousRelax

8 Which algorithm should I use?

Since Muriqui provides several MINLP algorithms to be applied, a natural question is how to choose the algorithm to be applied. Although the user is encouraged to perform various tests in order to determine the best approach for his/her specific application, we know that this may be impractical in some situations. Choosing the right algorithm for each situation can take a number of factors into account. For example, if the user only wants to quickly obtain a feasible solution, he/she should prefer to apply a feasibility heuristic (e.g. IGMA2, if the problem is binary, OAFP if the problem is convex, or FP for a more general problem).

If a problem needs to be solved to optimality, a lot of information can be taken into account. We have observed, for example, that linear approximation algorithms usually do well on convex problems. So, if you know in advance that your problem is in this category, an algorithm in that class would be the most natural choice. Otherwise, the best option may be HOABB. We

discuss below some characteristics of the algorithms and their implementations that can help the user in this decision.

8.1 Linear approximation algorithms

8.2 LP based Branch-and-Bound (LP-BB)

In our tests [13], the LP-BB algorithm [13] has shown the best practical performance on convex problems. For now, this algorithm can only be applied if the Cplex or Gurobi API is available in the execution environment and one of them is set as the MILP solver. This is the Muriqui's default algorithm, which can make use of all CPU processors available on the hardware.

8.2.1 LP/NLP based Branch-and-Bound (LP/NLP-BB)

In our tests [12], the LP/NLP-BB algorithm [14] has shown very good practical performance on convex problems. For now, this algorithm can only be applied if the Cplex or Gurobi API is available in the execution environment and one of them is set as the MILP solver. Our implementation can make use of all CPU processors available on the hardware.

8.2.2 Outer Approximation

The OA algorithm [6, 7] may be a good option in general situations where the user wishes to solve a convex problem on a computer with multi-processors (CPU). However, the OA algorithm solves, at each iteration, one or two NLP problems, which are subproblems constructed from (P). If for some reason the user suspects that solving continuous relaxation or any of the (P) subproblems requires too much effort, it may be a good idea to test algorithms such as ECP or ESH.

8.2.3 Extended Cutting Plane

The ECP algorithm [15] is characterized by not solving any NLP subproblems during its execution. This algorithm also has the friendly feature of being a totally first order method, which means that it does not use any information from second derivatives or even from approximations of them. If the user suspects that his/her (convex) MINLP problem has many computationally expensive second-order derivatives to be calculated, or if NLP packages have difficulty in solving their continuous relaxation, this may be the most appropriate algorithm. If the user is using Muriqui through the API, where the derivatives need to be coded, the user does not need to build callbacks for the calculation of the second order derivatives, in case this algorithm is chosen. As OA, the ECP algorithm is also enabled to use all CPU processors available in the hardware.

8.2.4 Extended Supporting Hyperplane

The ESH algorithm [9] can be seen as a good alternative to the ECP algorithm. However, the current implementation in Muriqui needs to solve an NLP subproblem at the beginning of the algorithm, which may require the computation of second order derivatives or their approximations. From this point, the ESH algorithm proceeds as a first-order method as well. Preliminary tests showed some superiority of ESH when compared to ECP, on problems with low percentage

of linear constraints. As OA and ECP, ESH is also enabled to use all CPU processors available in the hardware.

8.2.5 Bonmin Hybrid Branch-and-Bound

This algorithm is the hybrid Branch-and-Bound implemented in the solver Bonmin [2]. It is based on the application of LP/NLP-BB with the use of OA and the resolution of continuous NLP relaxations of subproblems through the enumeration tree. In our tests, our implementation of this algorithm failed to outperform the pure LP/NLP-BB. However, if you know in advance that the Bonmin implementation performs well on your (convex) MINLP problem, this algorithm may be a good choice (in any way, we recommend the user to also test pure LP/NLP-BB in this case). Since this algorithm is based on the execution of the LP/NLP-BB iterations, its implementation in Muriqui has the same usage restrictions as the latter (see Subsection 8.2.1).

8.3 Hybrid Outer Approximation Branch-and-Bound (HOABB)

The HOABB algorithm [10] seems to be the most suitable for handling non-convex problems, since linear approximation algorithms are strongly based on convexity, and not rarely, declare infeasibility for non-convex feasible problems. It should be noted, however, that this algorithm also does not guarantee optimality when solving non-convex problems, and can only be used as an heuristic. This algorithm is based on the application of the non-linear Branch-and-Bound with the recurrent application of OA. By adjusting its parameters, this OA application can be disabled, thus making the algorithm a pure non-linear Branch-and-Bound. The difficulty in solving NLP relaxations results in many of the cases observed, in linear approximation algorithms to perform better than HOABB. However, if a situation appears where linear approximation algorithms present poor performance, adopting HOABB may be a good choice. This algorithm can make use of all CPU processors available on the hardware.

9 Understanding C++ API

9.1 Evaluating nonlinear terms

To use the Muriqui API to solve a model, it is necessary to code procedures (class methods) for calculations involving nonlinear terms and their first and second order derivatives (function evaluations). Depending on the context of use, some procedures to calculate derivatives may be dispensed with. For example, the ECP algorithm does not use second-order derivatives. It is also possible to use Muriqui's pure Branch-and-Bound algorithm with some NLP solver, such as Ipopt, to approximate the derivatives. In this case, the procedures for calculating the respective derivatives being approximated could also be omitted. However, for a better use of the Muriqui functions as a whole, it is strongly recommended to code the procedures to calculate the derivatives.

The subfolder $MURIQUIDIR/examples contains examples of using Muriqui Optimizer. The first example, t1gross, was taken from [6], and its formulation is given by:

$$
\begin{aligned}
(E_1) \quad \min_{x} \quad & 5x_3 + 6x_4 + 8x_5 + 10x_0 - 7x_2 - 18\ln(x_1 + 1) - 19.2\ln(x_0 - x_1 + 1) + 10 && (2)\\
\text{s. t} \quad & 0.8\ln(x_1 + 1) + 0.96\ln(x_0 - x_1 + 1) - 0.8x_2 \geq 0, && (3)\\
& x_1 - x_0 \leq 0, && (4)\\
& x_1 - Ux_3 \leq 0, && (5)\\
& x_0 - x_1 - Ux_4 \leq 0, && (6)\\
& \ln(x_1 + 1) + 1.2\ln(x_0 - x_1 + 1) - x_2 - Ux_5 \geq -2, && (7)\\
& x_3 + x_4 \leq 1, && (8)\\
& 0 \leq x_0 \leq 2, && (9)\\
& 0 \leq x_1 \leq 2, && (10)\\
& 0 \leq x_2 \leq 1, && (11)\\
& x_3, x_4, x_5 \in \{0, 1\} && (12)\\
& U = 2. && (13)
\end{aligned}
$$

The $MURIQUIDIR/examples/t1gross/t1gross.hpp file contains a code for calculating the derivatives of problem (E_1). The first step is to declare a class specifically for calculations of the nonlinear terms of the problem and their derivatives. This class must be derived from the class MRQ_NonLinearEval, whose declaration can be made available by including the library "muriqui.hpp", available in the folder $MURIQUIDIR/includes:

```
#include <cmath>
#include "muriqui.hpp"

using namespace minlpproblem;
using namespace muriqui;

//we have to define a class to perform nonlinear function evaluations
class MyEval : public MRQ_NonLinearEval
{
   public:

   //that method is called before other functions evaluations in a algorithm
   virtual int initialize(const int nthreads, const int n, const int m, const
       int nzNLJac, const int nzNLLagHess) override;

   //to evaluate objective function nonlinear part
   virtual int eval_nl_obj_part(const int threadnumber, const int n, const bool
       newx, const double *x, double &value) override;

   //to evaluate nonlinear constraints part
   virtual int eval_nl_constrs_part(const int threadnumber, const int n, const
       int m, const bool newx, const bool *constrEval, const double *x, double *
       values) override;

   //to evaluate nonlinear objective gradient part
   virtual int eval_grad_nl_obj_part(const int threadnumber, const int n, const
       bool newx, const double* x, double* grad) override;

   //to evaluate nonlinear jacobian part
```

```
    virtual int eval_grad_nl_constrs_part(const int threadnumber, const int n,
        const int m, const int nz, const bool newx, const bool *constrEval, const
        double *x, MIP_SparseMatrix& jacobian) override;

    //to evaluate nonlinear hessian of lagrangian (only lower triangle)
    // lagrangian:  objFactor*f(x,y) + lambda*g(x,y)
    virtual int eval_hessian_nl_lagran_part(const int threadnumber, const int n,
        const int m, const int nz, const bool newx, const double *x, const double
        objFactor, const double *lambda, MIP_SparseMatrix& hessian) override;

    //that method is called after all functions evaluations in a algorithm
    virtual void finalize(const int nthreads, const int n, const int m, const int
         nzNLJac, const int nzNLLagHess) override;

    virtual ~MyEval(){};
};
```

At first, after including the library muriqui.hpp, we can notice the use of two distinct namespaces. In addition to the namespace muriqui, there is also the namespace minlpproblem, which is a sub-package of Muriqui Optimizer responsible for managing the reading, representation and writing of MINLP problems. Note that our class responsible for calculating nonlinear terms and their derivatives is called MyEval. The first method of the class is the *initialize* method, whose definition in this example is given by:

```
int MyEval::initialize(const int nthreads, const int n, const int m, const int
    nzNLJac, const int nzNLLagHess)
{
    return 0;
}
```

The *initialize* method is called once per algorithm execution with the *run* method, before any function evaluation. Thus, any initialization operations required for calculations, such as defining internal parameters and allocating auxiliary memory structures, can be performed in this method. On the other side, each algorithm when finished will call the *finalize* method, where shutdown procedures, such as free memory allocation, can be made. Note that special care should be taken if your class needs to allocate memory to perform function evaluations: some of the Muriqui algorithms are multithreads, which means that multiple calls to the evaluation procedures can be made simultaneously. Therefore, it is necessary to ensure that its procedures will work correctly on the allocated memory structures, without any thread incorrectly interfering in the calculation of another. An alternative to avoid these difficulties, especially in the first tests, is to restrict the number of threads to 1. This can be done by setting the input parameter *in_number_of_threads* in any class that represents some algorithm.

The parameters that the *initialize* method receives are: *nthreads*: number of concurrent execution threads to be created by the algorithm for function evaluation; n: total number of decision variables in the MINLP problem; m: total number of constraints in the MINLP problem; *nzNLJac*: declared number of non-zeros in the Jacobian of nonlinear terms; *nzNLLagHess*: declared number of non-zeros in the lower triangle of the Lagrangian Hessian. As no initialization procedure is required in our example, we only return the value 0 to indicate to Muriqui the success in executing the method. Whenever a method of the considered class returns a non-zero value, Muriqui assumes that an error has occurred in the execution of the respective procedure, and then the running algorithm can be aborted if the required value is crucial for

its continuation.

The next method of the class MyEval is *eval_nl_obj_part*, which is the method responsible for calculating the value of the nonlinear part of the objective function $f(x)$ at a given solution x. Notice that the variable value is an output parameter and must contain at the end of the execution of this method the value of the calculated objective function at the given solution x:

```
//to evaluate objective function nonlinear part
int MyEval::eval_nl_obj_part(const int threadnumber, const int n, const bool
    newx, const double *x, double &value)
{
   value = 5*x[3] + 6*x[4] + 8*x[5] + 10*x[0] - 7*x[2] - 18*log(x[1]+1) -19.2*
       log(x[0] -x[1] + 1) + 10;
   return 0;
}
```

In this method, the input parameter *threadnumber* specifies which thread is requesting the calculation (remember that for multithreads algorithms, more than one thread can request computations simultaneously). The parameter *newx* indicates whether the current solution x being passed to the method was the same one passed in the last calculation involving function evaluations for the thread considered. It is important to note here that only nonlinear terms need to be effectively calculated in this method. Although we have coded the entire objective function in this example, we could have specified the linear and constant terms of the objective function in the MINLP problem definition itself, leaving only the logarithms, as in example t2gross ($MURIQUIDIR/examples/t2gross). See this last example for more details on how to accomplish this separation of terms in the objective function and constraints of the problems. In general, performing this separation of terms makes the function evaluation simpler and can increase the efficiency of Muriqui, since it can take advantage of the decomposition of the problem in its procedures. Even though we have made this first example without this separation to demonstrate the general use of the API in a more simplified way, we recommend separating the linear and quadratic terms from the problem for a better efficiency as performed in example t2gross.

The next method of the MyEval class is the *eval_nl_constrs_part* method, which is responsible for evaluating the terms in the nonlinear constraints ($g_i(x)$) at a given solution x. Note that here, *values* is an output parameter that contains an array with the values of nonlinear terms:

```
//to evaluate nonlinear constraints part
int MyEval::eval_nl_constrs_part(const int threadnumber, const int n, const int
    m, const bool newx, const bool *constrEval, const double *x, double *values)
{
   const double U = 2;
   const double *y = &x[3];

   //linear and quadratic constraints must not be evaluated

   if( !constrEval || constrEval[0] )
      values[0] = 0.8*log(x[1] + 1) + 0.96*log(x[0] - x[1] + 1) - 0.8*x[2];

   if( !constrEval || constrEval[4] )
      values[4] = log(x[1] + 1) + 1.2*log(x[0] - x[1] + 1) -x[2] - U*y[2];

   return 0;
```

```
}
```

In this method, the boolean vector *construcEval* indicates what constraints need to have their nonlinear terms computed. If this pointer is null, it means that all constraints with nonlinear terms must be evaluated at the solution x passed as argument. We draw attention to the fact that linear and quadratic constraints should not be evaluated in this method! Any linear or quadratic term can be specified in the definition of the MINLP problem and therefore should not be calculated by the methods of class MyEval. For this reason, only the values for the first and fifth constraints are computed by the *eval_nl_constrs_part* method, since these are the only ones that contain nonlinear terms. We observe again that the linear and quadratic terms present in these constraints need not be coded in this method, they are mentioned here just for didactic matters. Linear or quadratic terms stated in the definition of the MINLP problem are automatically calculated by Muriqui when some function evaluation is required.

Next, we have the *eval_grad_nl_obj_part* method, which computes the gradient vector with respect to the nonlinear terms of the objective function ($\nabla f(x)$) at a given solution x:

```
//to evaluate nonlinear objective gradient part
int MyEval::eval_grad_nl_obj_part(const int threadnumber, const int n, const
    bool newx, const double *x, double *grad)
{
   grad[0] = 10 - 19.2/(x[0] -x[1] + 1);
   grad[1] = -18/(x[1]+1) + 19.2/(x[0] -x[1] + 1);
   grad[2] = -7;
   grad[3] = 5;
   grad[4] = 6;
   grad[5] = 8;

   return 0;
}
```

where the output parameter *grad* is a full (nonsparse) array of n positions, with one position for each partial derivative of the n decision variables. It is necessary to fill all the n positions of the vector, even if some variable does not appear in the objective function (in that case, its respective position will be filled with zero). Again, only derivatives of the nonlinear terms could be considered here, if the linear and quadratic terms of the objective function are specified in the definition of the MINLP problem (which is not the case in this example).

We have now the *eval_grad_nl_constrs_part* method, responsible for the calculation of the Jacobian (first order derivatives) of the nonlinear terms of the constraints ($\nabla g_i(x)$):

```
//to evaluate nonlinear jacobian part
int MyEval::eval_grad_nl_constrs_part(const int threadnumber, const int n, const
    int m, const int nz, const bool newx, const bool *constrEval, const double *
   x, MIP_SparseMatrix& jacobian)
{
   const double U = 2.0;

   if( !constrEval || constrEval[0] )
   {
      jacobian.setElement(0, 0, 0.96/(x[0] - x[1] + 1)); //line 0, column 0
      jacobian.setElement(0, 1, ( 0.8/(x[1] + 1) - 0.96/(x[0] - x[1] + 1) )); //
          line 0, column 1
      jacobian.setElement(0, 2, -0.8); //line 0, column 2
   }
```

```
   if( !constrEval || constrEval[4] )
   {
      jacobian.setElement(4, 0, 1.2/( x[0] - x[1] + 1 )); //line 4, column 0
      jacobian.setElement(4, 1, ( 1/(x[1] + 1) - 1.2/(x[0] - x[1] + 1) )); //
          line 4, column 1
      jacobian.setElement(4, 2, -1); //line 4, column 2
      jacobian.setElement(4, 5, -U); //line 4, column 5
   }

   return 0;
}
```

In this method, *nz* returns the number of non null elements in the nonlinear Jacobian specified in the problem definition. The output parameter *jacobian* represents a sparse array for storing nonzero elements in the Jacobian. The respective positions of this array that will contain the non null values must be specified in the MINLP problem definition. Thus, this method must calculate Jacobian values for **all** positions (of the constraints required in *constructval*) defined as nonzero in the MINLP problem specification. Note again that only nonlinear constraints have been considered.

Next, the *eval_hessian_nl_lagran_part* method appears with the purpose of evaluating the Hessian matrix of the Lagrangian (only the lower triangle). The Lagrangian here is defined as:

$$L(x, \lambda) = \alpha f(x) + \sum_{i=1}^{m} \lambda_i g_i(x),$$

and its Hessian will be defined by:

$$\nabla^2 L(x, \lambda) = \alpha \nabla^2 f(x) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x).$$

Thus, the *eval_hessian_nl_lagran_part* method is defined as:

```
//to evaluate nonlinear hessian of lagrangian (only lower triangle)
// lagrangian:  objFactor*f(x,y) + lambda*g(x,y)
int MyEval::eval_hessian_nl_lagran_part(const int threadnumber, const int n,
    const int m, const int nz, const bool newx, const double *x, const double
    objFactor, const double *lambda, MIP_SparseMatrix& hessian)
{
   double aux;
   hessian.setAllSparseMatrix(0.0); //set all sparse matrix to 0.0


   //first we set the objective part
   if( objFactor != 0.0 )
   {
   aux = objFactor * 19.2/pow(x[0] -x[1] + 1, 2);

   hessian.setElement(0, 0, aux ); //setting value to position 0,0
   hessian.setElement(1, 0, -aux ); //setting value to position 1,0
   hessian.setElement(1, 1, objFactor * ( 18.0/pow( x[1] + 1, 2) + 19.2/pow(x[0]
       - x[1] + 1, 2) ) ); //setting value to position 1,1
   }
```

```
    //now, the constraints

    aux = lambda[0]*( -0.96/pow( x[0] - x[1] + 1, 2) ) + lambda[4]*( -1.2/pow( x
        [0] - x[1] + 1, 2) );

    hessian.addToElement(0, 0, aux); //adding value to position 0,0
    hessian.addToElement(1, 0, -aux); //adding value to position 1,0


    aux = lambda[0]*( -0.8/pow( x[1] + 1, 2) - 0.96/pow( x[0] - x[1] + 1, 2) ) +
        lambda[4]*( -1/pow(x[1] + 1, 2) - 1.2/pow(x[0] - x[1] + 1, 2) );

    hessian.addToElement(1, 1, aux); //adding value to position 1,1

    return 0;
}
```

The input parameter *objFactor* represents the factor α, and the output parameter *hessian* represents a sparse matrix with the lower triangle of the Lagrangian Hessian. As this matrix accumulates a sum of other matrices, it is opportune to zero out all of its positions before the calculations start with the *setAllSparseMatrix* method:

```
void MIP_SparseMatrix::setAllSparseMatrix(double value);
```

To properly adjust the positions of the matrix, we use the methods for the sparse Hessian *setElement* and *addToElement*, whose prototype can be checked in the following:

```
int MIP_SparseMatrix::setElement(unsigned int rowIndex, unsigned int colIndex,
    double value);
int MIP_SparseMatrix::addToElement(unsigned int rowIndex, unsigned int colIndex,
     double value);
```

Finally, the *finalize* method is called when the running algorithm finishes all its calculations and no other function evaluation is needed. This method is useful for ending possible initializations made in the *initialize* method. Since, in our example, we do not need to do any operation or memory allocation in the *initialize* method, we do not need any operations on the *finalize* method, as well. In fact, in this example, this method does not even need to be specified. Note that this method returns no value.

```
void MyEval::finalize(const int nthreads, const int n, const int m, const int
    nzNLJac, const int nzNLLagHess)
{
}
```

9.2 Verifying the evaluation of the nonlinear terms

The occurrence of errors during the coding of evaluations of nonlinear terms is very common, which may cause the functions to be incorrectly evaluated and compromise the execution of the algorithms that depend on them. For this reason, Muriqui incorporates a simple derivative verifier based on finite differences. This verifier compares the coded derivatives with values obtained by finite differences, warning about any inconsistency. The file $MURIQUIDIR/examples/t1gross/t1gross.cpp provides an example of using this verifier, called through *checkFisrtDerivatives* and *checkSecondDerivatives* methods, whose signatures are given by:

```
int MRQ_MINLPProb::checkFisrtDerivatives(bool checkObj, bool checkConstr, const
    double *x, const double step, const double *steps, const double tolerance,
    bool &answer);

int MRQ_MINLPProb::checkSecondDerivatives(bool checkObj, bool checkConstr, const
     double* x, const double objFactor, const double* lambda, const double step,
    const double* steps, const double tolerance, bool& answer);
```

that are present in the MRQ_MINLPProblem class, responsible for managing the representation of the MINLP problem.

9.3 Specifying characteristics of the problem

The namespace minlpproblem brings the definition of the MRQ_MINLPProblem class (MIP_MINLPProblem), which has the purpose of managing the representation of the MINLP problem. This class has several methods that allow the specification of the problem addressed. Examples $MURIQUIDIR/examples/t1gross/t1gross.cpp and $MURIQUIDIR/examples/t2gross/t2gross.cpp are good starting points for understanding how the methods in the class work.

9.4 Creating your own Makefile for compilation

Since Muriqui can make use of several external libraries, it may not be trivial to compile a new application that incorporates their API. Our recommendation is to create a Makefile for the compilation of new projects based on the Makefile and make.inc files in the examples. Just by changing a few lines in the Makefile of the examples, it should be possible to build Makefiles suitable for new applications. Special attention should be given to setting the MYMURIQUIDIR variable in the Makefile file, which should point to the $MURIQUIDIR directory.

9.5 Setting parameters of supported solvers

For the execution of the algorithms implemented, Muriqui works with external solvers to solve MILP and NLP subproblems, for example. In some specific situations, it may be necessary to adjust specific parameters of these solvers. To allow an easy adjustment of these parameters, Muriqui implements a class called MRQ_GeneralSolverParams, which stores an arbitrary number of parameters of the types *integer*, *double* and *string*. In general, the execution of the MINLP algorithms implemented in Muriqui can receive two objects of class MRQ_GeneralSolverParams through the *run* method. The former usually specifies parameters to be passed to the MILP solver, as the latter usually specifies parameters to the NLP solver.

Let us assume, for example, that we are interested in executing the Outer Approximation (OA) algorithm using Cplex and Ipopt as MILP and NLP solvers, respectively. The following code is used to store parameters to be passed to Cplex.

```
MRQ_GeneralSolverParams milpsParams;

milpsParam.storeDoubleParameter("CPX_PARAM_EPGAP", 1e-3); //adjusts tolerance of
    the relative integrality gap   of cplex to 1e-3

milpsParam.storeIntegerParameter("CPX_PARAM_SCRIND", 0); //disables cplex
    messages with output print
```

Now, the following code stores parameters for Ipopt

```
MRQ_GeneralSolverParams nlpsParams;

nlpsParams.storeIntegerParameter("max_iter", 1000); //sets the maximum number of
    iterations of Ipopt to 1000

nlpsParams.storeIntegerParameter("print_level", 4); //sets the print level of
    Ipopt to 4

nlpsParams.storeStringParameter("linear_solver", "ma27"); //sets the linear
    solver used by Ipopt to ma27
```

You can then call the *run* method to execute an algorithm by passing pointers to the MRQ_GeneralSolverParams objects responsible for storing the parameters to be adjusted:

```
MRQ_OuterApp oa;
oa.run(problem, &milpsParams, &nlpsParams);
```

Note that, in some cases, MRQ_GeneralSolverParams objects may be ignored. For example, the ECP algorithm does not make use of an NLP solver, so it ignores the object with the parameters for this type of solver. The heuristic Diving, on the other hand, does not use an MILP solver, and, therefore, does not make use of the corresponding parameters.

9.6 Organization of the classes of algorithms in API C++

Muriqui defines a general class for algorithms representation, called MRQ_Algorithm. The algorithms are implemented in Muriqui in classes derived from this one. This means that definitions made for the MRQ_Algorithm class, such as general parameters and methods, are valid for all classes that implement algorithms. Other general (sub)classes below MRQ_Algorithm, are MRQ_LinearApproxAlgorithm, which generalizes definitions of linear approximation algorithms, and MRQ_Heuristic, which brings general definitions to heuristics. Figure **??** shows the organization chart of the Muriqui algorithms classes. The definitions made in the upper classes are valid for the lower classes in the given organizational chart flow. This is especially useful for understanding the operation of the general input and output methods and parameters used by Muriqui. Finally, we have pointed out the existence of the MRQ_ContinuousRelax class, which has the purpose of allowing the resolution of the continuous relaxation of a MINLP problem by means of some of the supported NLP *solvers*. If the solver supports the definition of integer variables, such as Knitro, it is possible to use this class to effectively solve the MINLP problem with this solver using the Muriqui API as an intermediary.

10 Input parameters

The input parameters control aspects of the behavior of the algorithms. Generally, they are divided into 3 categories:

- Parameters adjusted to integers (*int*) (for some practical purposes, boolean parameters are also included here, which are then treated as binary integers, i.e., treated as 0 and 1);
- Parameters adjusted to real numbers (*dbl*);

- Parameters adjusted to enumerated constants (inside the API) or text strings (*str*) (outside the API).

Note that to perform a correct adjustment, it is necessary to know the type of parameter to be adjusted. Parameter setting by the AMPL system is exemplified in Section **??**, where it is necessary to describe a triplet, specifying the type of parameter (*int*, *dbl* or *str*), its name, and its value. For example, the command:

```
options muriqui_options "   str in_nlp_solver MRQ_NLP_MOSEK
dbl in_max_cpu_time 600.0
int in_max_iterations 1000";
```

establishes that

1. the parameter *in_nlp_solver* (of type *str*) must be adjusted to MRQ_NLP_MOSEK, which will set Mosek as the NLP solver;

2. the parameter *in_max_cpu_time* (of type *dbl*) must be adjusted to 600.0, which means that the algorithm considered can only be processed for a maximum of 600.0 seconds;

3. the parameter *in_max_iterations* (of type *int*) must be adjusted to 1000, which means that the algorithm considered can only execute at most 1000 iterations;

For running Muriqui as an executable program, one can specify parameters through a special file called muriqui_params.opt. Thus, these same parameters could be adjusted by defining this file in the directory of execution with the following content:

```
str in_nlp_solver MRQ_NLP_MOSEK
dbl in_max_cpu_time 600.0
int in_max_iterations 1000
```

The choice of the algorithm, which is made before the parameter adjustments, can be made by specifying a special file called muriqui_algorithm.opt, for example, with the content:

```
MRQ_OA_ALG
```

which specifies that the algorithm to be used is the Outer Approximation. Here, any algorithm constant of Table 1 could have been used. Note that, in this case only, in particular, there is no definition of type or name of the parameter, only its value.

It should be noted that input parameter names are always prefixed by *in_*, while output parameter names are always prefixed by *out_*.

Muriqui also allows user to adjust parameters of the MILP and NLP solvers used in the algorithms. For this, you can specify the muriqui_milp_params.opt and muriqui_nlp_params.opt files respectively with the desired parameters following the same schema as the muriqui_params.opt file. Note that the set of parameters available for adjustment varies according to each solver, and it is necessary to consult the respective documentation for more details. For example, for Cplex, we could define a muriqui_milp_params.opt file with the lines

```
dbl  CPX_PARAM_EPGAP     1e-5
int  CPX_PARAM_NODELIM   9999999
```

Whereas, for Ipopt, we could define a muriqui_nlp_params.opt file as follows:

```
str   linear_solver     ma27
int   print_level       2
dbl   constr_viol_tol   1e-6
int   max_iter          5000
```

For convenience, lines of parameter definition files whose first character is # will be treated as commentary lines, that is, they will be ignored by Muriqui. Note that at most one parameter must be set per line in these files.

The adjustment of input parameters through the API can be done in two different ways. The first one is by accessing the parameter directly as an attribute of some of the algorithm classes (see examples in $MURIQUIDIR/examples), as in the following code:

```
MRQ_OuterApp oa;
oa.in_nlp_solver = MRQ_NLP_MOSEK;
oa.in_max_cpu_time = 600.0;
oa.in_max_iterations = 1000;
```

The second way consists of using the methods *setDoubleParameter*, *setIntegerParameter* and *setStringParameter*, which adjust parameters of type *double*, *integer* e *string*, respectively, as in the following example:

```
int MRQ_Algorithm::setDoubleParameter(const char *name, double value);

int MRQ_Algorithm::setIntegerParameter(const char *name, long int value);

int MRQ_Algorithm::setStringParameter(const char *name, const char *value);
```

As they are defined in the MRQ_Algorithm class, these methods apply to any algorithm implemented in Muriqui, as in the following example:

```
MRQ_OuterApp oa;
oa.setStringParameter(  "in_nlp_solver",      "MRQ_NLP_MOSEK");
oa.setDoubleParameter(  "in_max_cpu_time",    600.0);
oa.setIntegerParameter( "in_max_iterations", 1000);
```

10.1 Parameters defined in class MRQ_Algorithm

The parameters defined in the MRQ_Algorithm class apply to the algorithms implemented in Muriqui in general. Parameters specific to each approach are defined in the derived classes.

10.1.1 Real parameters *dbl*

- **in_absolute_convergence_tol**: absolute convergence tolerance. If the difference between the lower and upper bounds becomes less than or equal to this value, the algorithm stops declaring optimality if any feasible solution is known, or declares infeasibility, otherwise. Default value: $1.0e-3$;
- **in_absolute_feasibility_tol**: absolute feasibility tolerance, used to determine if solutions meet the general constraints of (P). Default value: $1.0e-6$;
- **in_integer_tol**: integrality tolerance. A solution is considered integer if the largest integrality gap among the integer variables is less than or equal to this value. Default value: $1.0e-4$;

- **in_lower_bound**: initial lower bound for the execution of the algorithms. In some situations, a valid lower bound for the optimal value of the objective function of (P) is known a priori. This value can be passed to the algorithms with this parameter, in an attempt to accelerate the optimality detection. Default value: -MRQ_INFINITY;

- **in_max_time**: maximum real time (clock time, in seconds) of execution of the algorithm. This parameter disregards the number of processors of the hardware and the execution time of the algorithm in each processor, letting the adjustment of this last one to the parameter *in_max_cpu_time*. Default value: ∞;

- **in_max_cpu_time**: maximum execution time (in seconds) in the processors of the algorithm. This parameter takes into account the execution time on all processors in the hardware. Default value: ∞;

- **in_relative_convergence_tol**: relative convergence tolerance. If the relative difference between the best upper and lower bounds is less than or equal to this value, the algorithm stops declaring optimality, if any feasible solution is known, or declares infeasibility, otherwise.

- **in_relative_feasibility_tol**: relative feasibility tolerance, used to determine if solutions satisfy the general constraints of (P). Default value: $1.0e-6$;

- **in_upper_bound**: initial upper bound. In some situations, an upper bound is already known for the optimal value of the objective function of (P). This value can be passed to the algorithms with this parameter, in an attempt to accelerate the optimality detection. Default value: MRQ_INFINITY;

10.1.2 Integer parameters *int*

- **in_assume_convexity**: flag that specifies whether the input problem should be treated as convex or not. Currently, this parameter has no practical use. Default value: 1 (*true*);

- **in_call_end_of_iteration_callback**: flag that specifies whether the callback of end of iteration should be called or not. If it is set to 1 (*true*), this callback allows the user to monitor the execution of the algorithm at each iteration. This parameter can only be used by the API, and a follow-up callback must be provided through the *in_user_callbacks* parameter. Default value: 0 (*false*);

- **in_call_update_best_sol_callback**: flag that specifies whether the solution-update callback should be called or not. If set to 1 (*true*), this callback allows the user to track the evolution of the best solution found by the algorithm. This parameter can only be used by the API, and a follow-up callback must be provided through the *in_user_callbacks* parameter. Default value: 0 (*false*);

- **in_max_iterations**: maximum number of iterations of the algorithm. Default value: ULONG_MAX;

- **in_number_of_threads**: number of threads used by the algorithm. In practice, this number specifies the number of processors to be used simultaneously to solve the problem, when it is applicable. Note that some algorithms do not allow parallel execution. The

adjustment of this parameter also affects the performance of the supported solvers, since this value is also passed to them. When this parameter is set to 0, Muriqui assumes the number of threads to be equal to the number of processors available in the hardware. We do not recommend adjusting this value with a bigger number than the number of processors available. Default value: 0;

- **in_preprocess_lin_constr**: flag that specifies whether to preprocess the linear constraints of the problem addressed or not. Default value: 1 (*true*);
- **in_preprocess_obj_function**: flag that specifies whether to preprocess the objective function of the problem addressed or not. Default value: 1 (*true*);
- **in_preprocess_quad_constrs**:flag that specifies whether to preprocess the quadratic constraints in the problem addressed or not. Default value: 1 (*true*);
- **in_print_level**: level of information of the algorithm printed during execution. More information is displayed for higher levels. Default value: 3;
- **in_print_parameters_values**: flag that specifies whether the value of all parameters of the current algorithm must be printed before its execution or not. It is useful to verify if the parameters setting is correct. Default value: 0 (*false*);
- **in_printing_frequency**: frequency of printing the iteration summary. At each *in_printing_frequency* iteration, Muriqui prints information regarding the current iteration. Default value: 1 (the default value of this parameter may be different in derived classes);
- **in_set_special_nlp_solver_params**: flag that specifies whether special parameters of the NLP solver should be set or not. In general, these special parameters are a subset of the solver parameters that are adjusted in an attempt to accelerate the convergence. However, this automatic adjustment can cause numeric instabilities in some cases. If your application is having difficulties in solving NLP subproblems, try to disable this flag. Default value: 0 (*false*) (the default value of this parameter may be different in derived classes, especially Branch-and-Bound-related algorithms);
- **in_store_history_solutions**: flag that specifies if a history of solutions should be constructed with the feasible solutions found along the iterations. This functionality is useful when it is desired to check, at the end of execution, all feasible solutions found by the algorithm. If this flag is enabled, these solutions will be stored in the output parameter *out_sol_hist*. Default value: 0 (*false*);
- **in_use_dynamic_constraint_set**: flag that specifies whether the Dynamic Restriction Set (DCS) should be adopted or not. This feature is only implemented in the Branch-and-Bound algorithm and allows to disregard some constraints of (P) if certain binary variables are integer, similar to the use of Big-M. For correct operation, you need to provide specific problem-information using the *setDCS0Array* and *setDCS1Array* methods. Default value: 0 (*false*);
- **in_use_initial_solution**: flag that specifies if the initial solution provided by the user should be used by the algorithm, for example, as a starting point for the NLP solver. Default value: 0 (*false*);

10.1.3 Enumerated parameters *str*

- **in_milp_solver**: determines the MILP solver adopted by the algorithm, when applicable. Make sure Muriqui has been correctly compiled to use the solver selected (see Section 4). The options for this parameter are listed in Table 2. Default value: MRQ_CPLEX.

Table 2: Constants of MILP solvers used in Muriqui

Solver	Constant	Priority Order
Cbc	MRQ_CBC	6
Cplex	MRQ_CPLEX	1
Glpk	MRQ_GLPK	7
Gurobi	MRQ_GUROBI	2
Knitro	MRQ_MILP_KNITRO	4
Mosek	MRQ_MILP_MOSEK	5
Xpress	MRQ_XPRESS	3

- **in_nlp_solver**: determines the NLP solver adopted by the algorithm, when applicable. Make sure Muriqui has been correctly compiled to use the solver selected (see Section 4). The options for this parameter are listed in Table 3. Default value: MRQ_NLP_KNITRO.

Table 3: Constants of NLP solvers used in Muriqui

Solver	Constant	Priority Order
Ipopt	MRQ_IPOPT	2
Knitro	MRQ_NLP_KNITRO	1
Mosek	MRQ_NLP_MOSEK	3

10.2 Parameters defined in class MRQ_LinearApproxAlgorithm

The MRQ_LinearApproxAlgorithm class is derived from MRQ_Algorithm, thus inheriting its parameters and methods. The parameters defined here apply to linear approximation algorithms in general.

10.2.1 Real Parameters *dbl*

- **in_eps_to_active_constr_to_linearisation**: absolute and relative tolerance used to consider active constraint. This value will be used if the *in_constr_linearisation_strategy* parameter is defined as
  MRQ_CLS_ONLY_INFEAS_AND_ACTIVE_MASTER_SOL_ALSO.

  Default value: $1.0e-4$.

10.2.2 Integer parameters *int*

- **in_measure_nlp_time**: flag that specifies the measurement of the time spent in solving NLP problems in the algorithm. The measured time (in seconds) will be available, at the end of the execution, in the output parameters *out_clock_time_of_nlp_solving* (clock time) and *out_cpu_time_of_nlp_solving* (processing time). Default value: 0 (*false*);

- **in_set_obj_lower_bound_on_master_problem**: flag that determines whether the current lower bound should be explicitly passed to the master problem. Default value: 0 (*false*);

- **in_set_quadratics_in_master_problem**: flag that specifies if objective function and quadratic constraints (if any) will be directly added to the master problem in its original quadratic form. If this flag is enabled, you will need to use a MILP solver that also solves mixed integer quadratic problems. Default value: 0 (*false*);

10.2.3 Enumerated parameters *str*

- **in_constr_linearisation_strategy**: specifies the constraint linearization strategy. Possible values for this parameter are:

  - MRQ_CLS_ALL_CONSTRS: linearizes all constraints at each linearization point;
  - MRQ_CLS_ONLY_INFEAS_AND_ACTIVE: linearizes only violated and active constraints in the solution of the master problem;

  Default value: MRQ_CLS_ALL_CONSTRS;

- **in_obj_linearisation_strategy**: determines the linearization strategy for the objective function. Possible values for this parameter are:

  - MRQ_OLS_ALL_POINTS: linearizes objective function at all points of linearization;
  - MRQ_OLS_NON_OBJ_CUT_POINTS: linearizes objective function at non-dominated (cut) points. This option is still undergoing experimentation;

  Default value: MRQ_OLS_ALL_POINTS;

- **in_quad_app_master_strategy**: determines the strategy of using quadratic approximation in the master problem. This parameter is in an experimental phase. Possible values for this parameter are:

  - MRQ_QAMS_NO_QUAD_APP: no use of quadratic approximation;
  - MRQ_QAMS_ON_BEST_POINT: quadratic approximation built on the best solution;
  - MRQ_QAMS_ON_LAST_POINT: quadratic approximation built on the point obtained in the previous iteration;

  Default value: MRQ_QAMS_NO_QUAD_APP.

10.3 Parameters defined in the MRQ_Heuristic class

The MRQ_Heuristic class is derived from MRQ_Algorithm, thus inheriting its parameters and methods. The parameters defined here apply to heuristic algorithms in general.

10.3.1 *Integer* parameters *int*

- **in_solve_nlp_as_local_search_at_end**: flag that specifies whether a local search NLP problem should be solved if the algorithm finds any workable solution. Default value: 1 (*true*);
- **in_seed_to_random_numbers**: seed for generating pseudo-random numbers, used if the parameter *in_use_random_seed_to_random_numbers* value is *false*. Default value: 1986;
- **in_use_random_seed_to_random_numbers**: flag that determines whether a "random" seed will be used to generate pseudo-random numbers. In practice, this "random" seed is generated from the current system time. Default value: 0 (*false*).

10.4 Parameters defined in MRQ_ExtCutPlan class (ECP)

The class MRQ_ExtCutPlan is derived from MRQ_LinearApproxAlgorithm, thus inheriting its parameters and methods. The parameters defined here are specific for the Extended Cutting Plane algorithm.

10.4.1 *Integer* parameter *int*

- **in_refine_final_solution_using_nlp**: flag that activates refinement of the solution found by solving a nonlinear programming problem where the integer variables are fixed in the values corresponding to the best solution found. The purpose of this refinement is to reduce possible numerical errors present in the best solution found. Default value: 1 (*true*).

10.5 Parameters defined in MRQ_OuterApp class (OA)

The class MRQ_OuterApp is derived from MRQ_LinearApproxAlgorithm, thus inheriting its parameters and methods. The parameters defined here are specific for the Outer Approximation algorithm.

10.5.1 *Integer* parameters *int*

- **in_binarie_cut_when_nlp_infeasible**: flag that determines the adoption of a binary cut when the NLP problem obtained by fixing the integer variables is not feasible. This strategy can only be adopted in binary problems. In this case, the NLP feasibility problem will not be solved. Default value: 0 (*false*);
- **in_round_first_nlp_relaxation_solution**: flag that specifies the use of rounding on the solution of the NLP problem of a continuous relaxation for possible linearization. Default value: 0 (*false*);

- **in_use_first_nlp_relaxation**: flag that determines whether the solution of continuous relaxation will be used as a linearization point. This parameter can only be disabled if the user has provided one or more alternative points of linearization through the method *addPointsToLinearisation*. Default value: 1 (*true*);

10.6 Parameters defined in MRQ_LPNLPBBOuterApp class (LP/NLP-BB)

The class MRQ_LPNLPBBOuterApp is derived from MRQ_LinearApproxAlgorithm, thus inheriting its parameters and methods. The parameters defined here are specific for the LP/NLP based Branch-and-Bound algorithm.

10.6.1 *Integer* parameters *int*

- **in_binarie_cut_when_nlp_infeasible**: flag that determines the adoption of binary cut when the NLP problem obtained by fixing the integer variables is not feasible. This strategy can only be adopted in binary problems. In this case, the NLP feasibility problem will not be solved. Default value: 0 (*false*);

- **in_use_first_nlp_relaxation**: flag that determines whether the solution of continuous relaxation will be used as a linearization point. This parameter can only be disabled if the user has provided one or more alternative points of linearization through the method *addPointsToLinearisation*. Default value: 1 (*true*);

- **in_linearize_obj_in_nl_feas_solutions**: flag that determines if a possible nonlinear objective function must be linearized in solutions from solving feasibility problems. Default value: 1 (*true*).

10.7 Parameters defined in MRQ_BonminHybrid class (BHBB)

The class MRQ_BonminHybrid is derived from MRQ_LPNLPBBOuterApp, thus inheriting its parameters and methods. The parameters defined here are specific for the Bonmin Hybrid Branch-and-Bound algorithm.

10.7.1 *Double* parameters *dbl*

- **in_outer_app_max_cpu_time**: maximum processing time, in seconds, for the initial execution of the External Approximation algorithm. Other options related to the External Approach can be adjusted using the *in_outer_app* parameter. Default value: 30.0;

- **in_outer_app_max_time**: maximum clock time, in seconds, for initial execution of the External Approximation algorithm. Other options related to External Approach can be adjusted using the *in_outer_app* parameter. Default value: ∞;

10.7.2 *Integer* parameters *int*

- **in_out_app_max_iterations**: maximum number of iterations at each application of the OA algorithm. Default value: ULONG_MAX (largest unsigned long integer representable in the current system);

- **in_nlp_relaxation_solving_frequence**: frequency of NLP relaxation resolutions along the tree from BB to MILP. Default value: 10.

10.8 Parameters defined in MRQ_ExtSupHypPlane class (ESH)

The class MRQ_ExtSupHypPlane is derived from MRQ_LinearApproxAlgorithm, thus inheriting its parameters and methods. The parameters defined here are specific for the Extended Supporting Hyperplane algorithm.

10.8.1 *Double* parameters *dbl*

- **in_absolute_tol_to_check_previous_sol**: absolute tolerance to enable checking solution repetition between iterations. Due to numerical errors, in some cases, the solution to the previous iteration may repeat itself. This parameter controls when two solutions must be tested component by component if the objective function value of both is close enough. Default value: $1.0e-6$;

- **in_cont_relax_absolute_convergence_tol**: absolute convergence tolerance in the resolution of the continuous relaxation of (P). This algorithm solves the continuous relaxation explicitly through ESH, and this tolerance is used as the criterion to stop this resolution. Default value: $1.0e-3$;

- **in_cont_relax_relative_convergence_tol**: relative convergence tolerance in the resolution of the continuous relaxation of (P). This algorithm solves the continuous relaxation explicitly through ESH, and this tolerance is used as the criterion to stop this resolution. Default value: $1.0e-3$;

- **in_delta_to_inc_eps_to_active_constraints_to_linearization**: value for tolerance increment of active constraints if solution repetition is detected. Default value: 0.05;

- **in_eps_to_enforce_interior_sol_on_cont_relax_sol**: epsilon value to be used as relative and absolute distance to force obtaining interior solution in the resolution of continuous relaxation. This parameter is used if the value of the parameter *in_starting_point_strategy* is MRQ_ESHP_SPS_CLOSED_TO_CONT_RELAX_SOL. Default value: 0.025;

- **in_eps_to_interior_sol**: epsilon value used to consider a solution as interior. Default value: 0.01;

- **in_eps_to_line_search**: tolerance used for zero value in the linear search procedure. Default value: $1.0e-6$;

- **in_relative_tol_to_check_previous_sol**: tolerance to enable checking solution repetition between iterations. Due to numerical errors, in some cases, the solution of

the previous iteration may repeat itself. This parameter controls when two solutions must be tested component by component if the objective function value of both is close enough. Default value: $1.0e-6$;

10.8.2 *Integer* parameters *int*

- **in_linearize_on_interior_sol**: flag that specifies whether the interior solution should also be used as a linearization point. Default value: 0 (*false*);
- **in_max_lp_subiters**: maximum number of sub-iterations using LP (for solving the continuous relaxation of (P)). Default value: 500;
- **in_max_lp_subiter_to_improve_obj_app**: maximum number of sub-iterations using LP (for solving the continuous relaxation of (P)) without improving the objective function value. Default value: 50;
- **in_try_solve_interior_problem_if_cont_relax_fail**: flag that determines whether the problem for obtaining an interior solution should be solved if an interior solution could not be obtained from the modified continuous relaxation of (P). This parameter is only used if the parameter value *in_starting_point_strategy* is MRQ_ESHP_SPS_CLOSED_TO_CONT_RELAX_SOL. Default value: 1 (*true*).

10.8.3 Enumerated parameters *str*

- **in_interior_point_strategy**: strategy for obtaining interior solution. The possible values for this parameter are:
  - MRQ_ESHP_IPS_MOST_INTERIOR: search for the most interior solution;
  - MRQ_ESHP_IPS_CLOSED_TO_CONT_RELAX_SOL: search for a solution that is close to the one of the continuous relaxation of (P).

  Default value: MRQ_ESHP_IPS_MOST_INTERIOR;
- **in_lp_constr_linearisation_strategy**: strategy for linearization of constraints in LP-based sub-iterations. The possible values for this parameter are the same for the parameter *in_constr_linearisation_strategy* in the MRQ_LinearApproxAlgorithm class. Default value: MRQ_CLS_ONLY_INFEAS_AND_ACTIVE.

10.9 Parameters defined in MRQ_BranchAndBound class (HOABB)

The class MRQ_BranchAndBound is derived from MRQ_Algorithm, thus inheriting its parameters and methods. The parameters defined here are specific for the Hybrid Outer Approximation Branch-and-Bound algorithm.

10.9.1 *Double* parameters *dbl*

- **in_feas_heuristic_max_time**: maximum time (clock), in seconds, for each feasibility heuristic application. Default value: 30.0;

- **in_igma2_factor_to_max_dist_constr**: factor (between 0 and 1) for neighborhood determination of the IGMA2 heuristic. Default value: 0.05;
- **in_igma2_factor_to_max_dist_constr_on_bin_vars**: establishes a factor (between 0 and 1) for neighborhood determination based on binary variables of the IGMA2 heuristic. This parameter is only used if the parameter *in_igma2_set_max_dist_constr_on_bin_vars* is set to 1 (*true*). Default value: 0.05;
- **in_outer_app_subprob_time**: maximum time (clock), in seconds, for each OA application in subproblems in the BB tree. Default value: 10.0;
- **in_outer_app_time**: maximum time (clock), in seconds, for each OA application in problem (P). Default value: 10.0;
- **in_pseudo_cost_mu**: factor (between 0 and 1) to calculate pseudo-costs. When the branching strategy is based on pseudo-costs, this factor balances the pseudo-cost calculations of each variable. The value 0 favors variables with a higher lower bound increase on both sides of the branch, while the value 1 privileges variables with the greatest lower bound increase on only one side of the branch. Default value: 0.2.

10.9.2 *Integer* parameters *int*

- **in_call_after_solving_relax_callback**: flag that enables calling the user callback after_solving_relax, which can be specified via the *in_user_callbacks* parameter. This callback is called after the resolution of each relaxation in each BB node and allows to modify the default behavior of BB, for example, by enabling additional pruning (user pruning), executing specialized heuristics formulated by the user, or even modifying the solution indicated as optimal for the relaxations. Default value: 0 (*false*);
- **in_call_before_solving_relax_callback**: flag that enables calling the user callback before_solving_relax, which can be specified via the *in_user_callbacks* parameter. This callback is called before the resolution of each relaxation in each BB node and allows to modify the default behavior of BB, for example, by enabling additional pruning (user pruning) or modifying the relaxation to be solved. Default value: 0 (*false*);
- **in_call_new_best_solution_callback**: flag that enables calling the user callback new_best_solution, which can be specified via the *in_user_callbacks* parameter. This callback is called after each update of the best solution known by the algorithm, and allows, for example, to modify the solution being updated or to perform user-specific local search procedures on that solution. Default value: 0 (*false*);
- **in_consider_relax_infeas_if_solver_fail**: this flag brings the following behavior to BB: if it is enabled and the NLP solver fails to solve some relaxation, that relaxation will be considered infeasible. If the flag is disabled and this same fault occurs, the algorithm will terminate its execution returning an error. Default value: 1 (*true*);
- **in_count_total_prunes**: flag that enables counting the number of prunings performed by the algorithm, discriminating prunings by infeasibility, limit, optimality and pruning of the user (performed by means of callbacks). If this flag is active, the pruning count is available in the output parameter *out_prune_counter*. Default value: 0 (*false*);

- **in_feas_heuristic_frequency**: frequency of application of feasibility heuristics. Enabled feasibility heuristics are applied alternately to each iteration, and this application can be interrupted after obtaining the first feasible solution if the *in_int_feas_heurs_strategy* is set to the value MRQ_BB_IHS_UNTIL_FIRST_FEAS_SOL. Default value: 50000;

- **in_igma2_frequency**: frequency of application of the IGMA2 heuristic. IGMA2, if enabled, is applied to every *in_igma2_frequency* iterations, and this application can be interrupted after obtaining the first feasible solution if the parameter *in_igma2_strategy* is set to the value MRQ_BB_IHS_UNTIL_FIRST_FEAS_SOL. Default value: 1;

- **in_igma2_set_max_dist_constr_on_bin_vars**: flag that enables the neighborhood on binary variables of the IGMA2 heuristic. Default value: 0 (*false*);

- **in_igma2_set_special_gap_min_solver_params**: flag that enables the adjustment of special parameters for the solver used in the resolution of the IGMA2 integrality gap minimization problems. This adjustment tries to speed up the execution of the solver. Default value: 1 (*true*);

- **in_igma2_solve_local_search_problem_even_on_non_int_sol**: flag that enables the resolution of the local search problem of IGMA2 even on non-integer solutions. Default value: 1 (*true*);

- **in_lists_reorganization_frequency**: reorganization frequency of storage data structure of the open node list. The reorganization of this structure occurs at each *in_lists_reorganization_frequency* iterations. Default value: 10000;

- **in_number_of_branching_vars**: maximum number of variables adopted in the branching process. At each branch, at most *in_number_of_branching_vars* variables can be chosen to partition the space. Note, however, that if this parameter is set to $\bar{p}$, $2\bar{p}$ partitions of the current partition will be generated. Default value: 1;

- **in_number_of_node_sublists**: number of sublists in the data structure that stores the list of open nodes. Default value: 100;

- **in_max_tree_level_to_count_prunes_by_level**: this parameter defines pruning count in the BB tree by level. Prunings will be counted up to the level determined by this parameter. After executing the algorithm, prune counting by level becomes available in the output parameter array *out_prune_counter_by_level*, where the i-th position of the array refers to the count of the i-th level. Default value: 0;

- **in_outer_app_frequence**: frequency of application of the OA algorithm. OA, if enabled, will be applied to the (P) problem every *in_outer_app_frequence* iterations. Default value: 10000;

- **in_outer_app_subprob_frequence**: frequency of application of OA algorithm as heuristic in subproblems. OA will be applied in subproblems, if enabled, to every *in_outer_app_subprob_fr* iterations. Default value: 100;

- **in_rounding_heuristic_call_iter_frequence**: frequency of application of the rounding heuristic. This heuristic, if enabled, is applied to every *in_rounding_heuristic_call_iter_frequence* iterations. Default value: 10000;

- **in_seed_to_random_numbers**: seed for generating pseudo-random numbers. Default value: 1986;
- **in_stop_multibranch_after_first_bound_prune**: this flag, if enabled, is responsible for disabling branching over multiple variables after the first pruning by limit. Default value: 1 (*true*);
- **in_use_dual_obj_to_bound_prunning**: this flag, if enabled, causes the dual objective function value to be used for pruning by limit. If it is not active, the primal value is used. Default value: 0 (*false*);
- **in_use_feas_heuristic_diving**: flag that enables the use of Diving heuristics. Default value: 1 (*true*);
- **in_use_feas_heuristic_fp**: flag that enables the use of Feasibility Pump heuristics. Default value: 1 (*true*);
- **in_use_feas_heuristic_oafp**: flag that enables the use of OA Feasibility Pump heuristics. Default value: 1 (*true*);
- **in_use_feas_heuristic_rens**: flag that enables the use of RENS heuristics. Default value: 1 (*true*);
- **in_use_outer_app**: flag that enables the use of the OA algorithm in BB. If this flag is disabled, together with the flag *in_use_outer_app_as_heuristic*, the algorithm becomes a pure BB. Default value: 1 (*true*);
- **in_use_outer_app_as_heuristic**: flag that enables the use of the OA algorithm as heuristics in subproblems. If this flag is disabled, together with the flag *in_use_outer_app*, the algorithm works as a pure BB. Default value: 1 (*true*).

10.9.3 Enumerated parameters *str*

- **in_branching_strategy**: determines branching strategy. Sets the variables selected for branching. Possible values for this parameter are:
  - MRQ_BB_BS_HIGHEST_INT_GAP: chooses variables with greater integrality gap;
  - MRQ_BB_BS_BIN_FIRST_HIGHEST_INT_GAP: selects variables with greater integrality gap with priority for binary variables. This means that non-binary variables are only selected when all binary variables have integer values in the solution of the current relaxation;
  - MRQ_BB_BS_STBRANCH_PSEUDO_COSTS: selects variables using strategy based on strong branching and pseudocosts according to [5];
  - MRQ_BB_BS_VAR_PRIORITIES: chooses variables according to user-defined priorities before executing the algorithm (not yet implemented);
  - MRQ_BB_BS_USER_INDEX_CHOICE: calls the user callback to dynamically determine the choice of variables for branching. This callback must be provided through parameter *in_user_callbacks*;

– MRQ_BB_BS_USER_NODE_GENERATION: calls the user callback to dynamically partition space. Here, the new BB enumeration nodes are generated directly by the user. This callback must be provided through parameter *in_user_callbacks*;

Default value: MRQ_BB_BS_STBRANCH_PSEUDO_COSTS;

- **in_exp_strategy**: establishes exploration strategy. Possible values for this parameter are listed in Table 4: The strategy depth/best limit uses depth until the first feasible solution to (P) is found, moving on to the best limit strategy. Default value: MRQ_BB_ES_BEST_LIMIT;

Table 4: Constants of Exploration Strategies for Branch-and-Bound

Strategy	**Constant**
depth	MRQ_BB_ES_DEPTH
breadth	MRQ_BB_ES_WIDTH
best limit	MRQ_BB_ES_BEST_LIMIT
depth/best limit	MRQ_BB_ES_DEPTH_BEST_LIMIT

- **in_int_feas_heurs_strategy**: specifies strategy for using feasibility heuristics. The IGMA2 heuristics is not affected by this parameter, having its use regulated by the *in_igma2_strategy* parameter. Possible values for this parameter are:
  - MRQ_BB_IHS_NO_HEURISTICS: does not use feasibility heuristics;
  - MRQ_BB_IHS_UNTIL_FIRST_FEAS_SOL: uses feasibility heuristics until the first feasible solution is obtained for (P);
  - MRQ_BB_IHS_ALWAYS: uses feasibility heuristics periodically until the end of the algorithm execution.

  Default value: MRQ_BB_IHS_UNTIL_FIRST_FEAS_SOL;

- **in_igma2_gap_min_solver**: determines the NLP solver for use in the IGMA2 feasibility heuristic. The possible values for this parameter are provided in Table 3. Default value: MRQ_NLP_KNITRO;

- **in_igma2_strategy**: specifies execution strategy for the IGMA2 feasibility heuristic. The possible values are the same as the parameter *in_int_feas_heurs_strategy*. Default value: MRQ_BB_IHS_UNTIL_FIRST_FEAS_SOL;

- **in_parent_sol_storing_strategy**: establishes the storage strategy of the parent solution of each node in the enumeration tree. Storing the parent solution at each node allows the use of warm start techniques at the cost of increasing memory demand. If the pseudo-cost strategy is being used as a branching strategy, the algorithm will need to store at least the primal solution of the parent of each node. The possible values for this parameter are:
  - MRQ_BB_PSSS_NO_STORING: does not store the parent solution;

  - MRQ_BB_PSSS_ONLY_PRIMAL: stores only the parent's primal solution. This option already allows using warm start in the resolution of each relaxation, however, a better result is expected with the simultaneous use of the primal and dual solution;
  - MRQ_BB_PSSS_PRIMAL_AND_DUAL: stores only the parent's dual solution.

  Default value: MRQ_BB_PSSS_NO_STORING;

- **in_rounding_heuristic_strategy**: defines the strategy of using rounding heuristics. The possible values for this parameter are:

  - MRQ_RS_NO_ROUNDING: does not use rounding;
  - MRQ_RS_NEAREST_INTEGER: rounds each integer variable to the nearest integer value;
  - MRQ_RS_PROBABILISTIC: rounds each integer variable in a random-probabilistic way, where the fractional part of the value of the variable will give the probability of it being rounded up. For example, if an integer variable gets 3.25 in the solution of the current relaxation, then this variable will have 25% chance to be rounded up and 75% chance to be rounded down;

  Default value: MRQ_RS_PROBABILISTIC.

10.10 Parameters defined in class MRQ_IGMA2 (IGMA2)

The MRQ_IGMA2 class is derived from MRQ_BranchAndBound, thus inheriting its parameters and methods. The parameters defined here are specific to the algorithm Integrality Gap Minimization Heuristic 2.

10.10.1 *Double* parameters *dbl*

- **in_factor_to_max_dist_constr**: factor (between 0 and 1) for neighborhood determination. Default value: 0.05;

- **in_factor_to_max_dist_constr_on_bin_vars**: factor (between 0 and 1) for neighborhood determination based on binary variables. This parameter is only used if the parameter *in_set_max_dist_constr_on_bin_vars* is set to value 1 (*true*). Default value: 0.05;

- **in_percentual_to_rectangular_neighborhood**: percentage of boxes of variables for building a rectangular neighborhood. This parameter is only used if the parameter *in_neighborhood_strategy* is set to the value MRQ_IGMA2_NS_RECTANGULAR. Default value: 0.05;

10.10.2 *Integer* parameters *int*

- **in_set_max_dist_constr_on_bin_vars**: flag that enables neighborhood constraint construction only on binary variables. Default value: 0 (*false*);

- **in_set_special_gap_min_solver_params**: flag that enables the adjustment of special parameters for the solver for the integrality gap minimization problems. This adjustment aims to try to speed up the execution of the solver. Default value: 1 (*true*);

- **in_solve_local_search_problem_even_on_non_int_sol**: flag that enables the resolution of the local search problem even on non-whole solutions. Default value: 1 (*true*);

10.10.3 Enumerated parameters *str*

- **in_gap_min_solver**: establishes the NLP solver for the gap minimization problems. The admissible values for this parameter are presented in Table 3.
  Default value: MRQ_NLP_KNITRO;

- **in_neighborhood_strategy**: neighborhood formulation strategy in the problem of minimizing the integrality gap. The possible values for this parameter are:

  - MRQ_IGMA2_NS_RECTANGULAR: (hyper) rectangular neighborhood around the current solution;
  - MRQ_IGMA2_NS_SPHERIC: (hyper) spherical neighborhood around the current solution;

10.11 Parameters defined in class MRQ_FeasibilityPump (FP)

The MRQ_FeasibilityPump class is derived from MRQ_Heuristic, thus inheriting its parameters and methods. The parameters defined here are specific to the Feasibility Pump algorithm.

10.11.1 *Double* Parameters *dbl*

- **in_lower_bound_to_pi**: lower bound for the parameter π used to flip a solution when cycle is detected. Default value: -0.3;

- **in_upper_bound_to_pi**: upper bound for the parameter π used to flip a solution when cycle is detected. Default value: 0.7;

10.11.2 *Integer* parameters *int*

- **in_last_iters_considered_to_cycles**: number of previous iterations considered in checking solution cycling. Default value: 5;

- **in_max_cycle_subiters**: maximum number of consecutive cycle break attempts. If a number of attempts to break consecutive cycles is greater than *in_max_cycle_subiters*, the algorithm will stop, returning an error. Default value: 20;

- **in_set_linear_obj_term_on_bin_vars_at_nlp**: flag that enables the use of a linear term in the NLP feasibility problem for binary variables. This linear term calculates the distance between the solution of the problem and the input solution. If this flag is not active, the binary variables will have terms for calculating the distance like the other variables, which may involve heavier elements in the NLP problem. Default value: 1 (*true*);

- **in_set_norm1_on_nlp**: flag that enables the use of norm 1 instead of norm 2 in the NLP feasibility problem. Default value: 1 (*true*).

10.12 Parameters defined in class MRQ_OAFeasibilityPump (OAFP)

The MRQ_OAFeasibilityPump class is derived from MRQ_LinearApproxAlgorithm, thus inheriting its parameters and methods. The parameters defined here are specific to the Outer Approximation Feasibility Pump algorithm.

10.12.1 *Integers* parameters *int*

- **in_set_linear_obj_term_on_bin_vars_at_nlp**: flag that enables the use of a linear term in the NLP feasibility problem for binary variables. This linear term calculates the distance between the solution of the problem and the input solution. If this flag is not activated, the binary variables will have terms for calculating the distance like the other variables, which may involve heavier elements in the NLP problem. Default value: 1 (*true*);
- **in_set_norm1_on_nlp**: flag that enables the use of norm 1 instead of norm 2 in the NLP feasibility problem. Default value: 1 (*true*);
- **in_solve_nlp_as_local_search_at_end**: flag that specifies whether a local search NLP problem should be solved if the algorithm finds a feasible solution. Default value: 1 (*true*).

10.13 Parameters defined in class MRQ_Diving (Diving)

The MRQ_Diving class is derived from MRQ_Heuristic, thus inheriting its parameters and methods. The parameters defined here are specific to the Diving algorithm.

10.13.1 *Double* parameters *dbl*

- **in_percentual_of_add_var_fixing**: percentage of additional variables being fixed in the branching simulation. Default value:: 0.2.

10.13.2 *Integer* parameters *int*

- **in_consider_relax_infeas_if_solver_fail**: when this flag is active, problems where the NLP solver fails to provide an answer are considered infeasible. If this same situation occurs with this flag inactive, the algorithm will be stopped and an error code will be returned. Default value: 1 (*true*).

10.13.3 Enumerated parameters *str*

- **in_dive_selec_strategy**: variable selection strategy for branching simulation. The possible values for this parameter are:
  - MRQ_DIVE_SS_FRACTIONAL: choice for the variable with the largest integrality gap;
  - MRQ_DIVE_SS_VECTORLENGHT: choice for the vector length, which seeks to take into account the variation in the objective function and the number of restrictions being affected.

Valor default: MRQ_DIVE_SS_FRACTIONAL.

10.14 Parameters defined in class MRQ_RENS (RENS)

The MRQ_RENS class is derived from MRQ_Heuristic, thus inheriting its parameters and methods. The parameters defined here are specific to the Relaxation Enforced Neighborhood Search algorithm.

10.14.1 *Double* parameters *dbl*

- **in_continuous_neighborhood_factor**: factor for defining neighborhood of continuous variables. Default value: 0.25;
- **in_integer_neighborhood_factor**: factor for defining neighborhood of integer variables. Default value: 0.25.

10.14.2 *Integer* parameters *int*

- **in_apply_only_heuristics_on_subproblems**: flag that restricts the application of feasibility heuristics only to the MINLP subproblem. Default value: 0 (*false*).

10.14.3 Enumerated parameters *str*

- **in_algorithm_to_solve_subproblem**: specifies algorithm for solving MINLP subproblem. The possible values for this parameter are those described in Table 1 (except for MRQ_RENS, of course).

11 Output parameters (output from algorithms)

The output parameters include the response provided by the application of an algorithm (execution status, solution, objective value) to a MINLP problem, as well as data regarding the execution (number of iterations, CPU time, etc). As the input parameters have the prefix *in_* in their name, the names of the output parameters are prefixed by *out_*. The output parameters are adjusted by applying some of Muriqui's algorithms (run method in the C++ API).

11.1 Output parameters defined in class MRQ_Algorithm

The output parameters defined in the MRQ_Algorithm class apply to the algorithms implemented by Muriqui in general. Specific parameters for each approach are defined in the derived classes. The output parameters defined in this class are:

- **out_feasible_solution**: flag set to *true*, if the execution of the algorithm was able to find a feasible solution for (P), and *false*, otherwise;
- **out_number_of_feas_sols**: number of feasible solutions found in the execution of the algorithm;
- **out_number_of_iterations**: total number of iterations performed by the algorithm;

- **out_number_of_threads**: number of threads of execution used directly by the algorithm. It is worth noting that it is possible that an algorithm, such as OA or ECP, uses a single thread of direct execution, but the process of solving MILP subproblems uses more than one thread. The total number of threads used directly or indirectly can be controlled using the input parameter *in_number_of_threads*.

- **out_return_code**: return code of the algorithm. The possible values for this enumerated parameter, together with their respective meanings are:

  - MRQ_OPTIMAL_SOLUTION: optimal solution found;
  - MRQ_INFEASIBLE_PROBLEM: infeasible problem;
  - MRQ_UNBOUNDED_PROBLEM: unlimited problem;
  - MRQ_ALG_NOT_APPLICABLE: algorithm not applicable;
  - MRQ_BAD_DEFINITIONS: inconsistent definitions in the problem or input parameters;
  - MRQ_BAD_PARAMETER_VALUES: inadequate values for one or more parameters;
  - MRQ_INDEX_FAULT: use of invalid index for some data structure;
  - MRQ_MEMORY_ERROR: memory error;
  - MRQ_UNDEFINED_ERROR: undefined error;
  - MRQ_MAX_TIME_STOP: stop by maximum time limit (clock, or cpu) reached;
  - MRQ_MAX_ITERATIONS_STOP: stop by maximum iterations limit reached;
  - MRQ_CALLBACK_FUNCTION_ERROR: error returned by some user callback function to evaluate nonlinear functions;
  - MRQ_STOP_REQUIRED_BY_USER: algorithm tracking callback function requested to stop it;
  - MRQ_MILP_SOLVER_ERROR: error when executing MILP solver;
  - MRQ_NLP_SOLVER_ERROR: error when executing NLP solver;
  - MRQ_LIBRARY_NOT_AVAILABLE: library not available. This error occurs when an algorithm is configured to use a library that was not available in Muriqui's compilation. For example, if Muriqui is compiled using only Cplex as a MILP solver, this error will occur if the user tries to execute any algorithm that demands the resolution of MILP problems and a solver other than Cplex is specified;
  - MRQ_NAME_ERROR: name error. Used, for example, in functions that perform parameter adjustment by name and value;
  - MRQ_VALUE_ERROR: value error. Used, for example, in functions that perform parameter adjustment by name and value;
  - MRQ_MINLP_SOLVER_ERROR: error while executing MINLP solver. To date, this return code is unused;
  - MRQ_INITIAL_SOLUTION_ERROR: initial solution error;
  - MRQ_HEURISTIC_FAIL: return code used by heuristic algorithms to designate a failure in the search process for a feasible solution for (P);

  - MRQ_LACK_OF_PROGRESS_ERROR: error of lack of progress;
  - MRQ_NLP_NO_FEASIBLE_SOLUTION: it was not possible to obtain a feasible solution for a continuous relaxation of (P);
  - MRQ_NONIMPLEMENTED_ERROR: functionality not yet implemented;
  - MRQ_HEURISTIC_SUCCESS: return code used by heuristic algorithms to designate success in the process of searching for a feasible solution for (P);
  - MRQ_CONT_RELAX_OPTIMAL_SOLUTION: optimal solution in the resolution of continuous relaxation of (P).

  To obtain a string from the return code one can use the function

```
std::string MRQ_getStatus(int returnCode);
```

  present in API;

- **out_algorithm**: code of the algorithm used to solve the problem. The possible values for this parameter are those described in Table 1;
- **out_clock_time**: clock time elapsed during the execution of the algorithm;
- **out_cpu_time**: total processing time used by the algorithm in all threads of execution;
- **out_cpu_time_to_fisrt_feas_sol**: CPU time to obtain the first feasible solution;
- **out_lower_bound**: best lower bound explicitly calculated by the algorithm;
- **out_upper_bound**: best obtained upper bound;
- **out_obj_opt_at_continuous_relax**: optimal objective function value of the continuous relaxation of (P);
- **out_sol_hist**: solution history. If the input parameter *in_store_history_solutions* is set to *true*, this object will contain a history with all the improved feasible solutions found by the algorithm during its execution;
- **out_best_sol**: array with the best solution found. The array dimension is the number of (P) variables. The value of this parameter is only considered valid if the output parameter *out_feasible_solution* is set to the value *true*, and therefore we recommend testing this last parameter before attempting to access *out_best_sol*. Note that this parameter refers to a pointer that can be null, in case of memory allocation error, or the run method has not been called the first time;
- **out_best_obj**: the objective function value at the best solution found. The value of this parameter is only considered valid if the output parameter *out_feasible_solution* is set to the value *true*.

References

[1] Timo Berthold. Rens. *Mathematical Programming Computation*, 6(1):33–54, Mar 2014.

[2] Pierre Bonami, Lorenz T. Biegler, Andrew R. Conn, Gérard Cornuéjols, Ignacio E. Grossmann, Carl D. Laird, Jon Lee, Andrea Lodi, François Margot, and Nicolas Sawaya. An algorithmic framework for convex mixed integer nonlinear programs. *Discrete Optimization*, 5(2):186–204, May 2008.

[3] Pierre Bonami, Gérard Cornuéjols, Andrea Lodi, and François Margot. A feasibility pump for mixed integer nonlinear programs. *Mathematical Programming*, 119:331–352, 2009. 10.1007/s10107-008-0212-2.

[4] Pierre Bonami and João Gonçalves. Heuristics for convex mixed integer nonlinear programs. *Computational Optimization and Applications*, pages 1–19, 2008. 10.1007/s10589-010-9350-6.

[5] Pierre Bonami, Jon Lee, Sven Leyffer, and Andreas Wächter. On branching rules for convex mixed-integer nonlinear optimization. *J. Exp. Algorithmics*, 18:2.6:2.1–2.6:2.31, November 2013.

[6] Marco Duran and Ignacio Grossmann. An outer-approximation algorithm for a class of mixed-integer nonlinear programs. *Mathematical Programming*, 36:307–339, 1986. 10.1007/BF02592064.

[7] Roger Fletcher and Sven Leyffer. Solving mixed integer nonlinear programs by outer approximation. *Mathematical Programming*, 66:327–349, 1994. 10.1007/BF01581153.

[8] David M. Gay. Hooking your solver to ampl. Technical report, 1997.

[9] Jan Kronqvist, Andreas Lundell, and Tapio Westerlund. The extended supporting hyperplane algorithm for convex mixed-integer nonlinear programming. *Journal of Global Optimization*, 64(2):249–272, 2016.

[10] Wendel Melo, Marcia Fampa, and Fernanda Raupp. Integrating nonlinear branch-and-bound and outer approximation for convex mixed integer nonlinear programming. *Journal of Global Optimization*, 60(2):373–389, 2014.

[11] Wendel Melo, Marcia Fampa, and Fernanda Raupp. Integrality gap minimization heuristics for binary mixed integer nonlinear programming. *Journal of Global Optimization*, 71(3):593–612, Jul 2018.

[12] Wendel Melo, Marcia Fampa, and Fernanda Raupp. An overview of minlp algorithms and their implementation in muriqui optimizer. *Annals of Operations Research*, May 2018.

[13] Wendel Melo, Marcia Fampa, and Fernanda Raupp. Two linear approximation algorithms for convex mixed integer nonlinear programming. *Annals of Operations Research*, 2020.

[14] Ignacio Quesada and Ignacio E. Grossmann. An lp/nlp based branch and bound algorithm for convex minlp optimization problems. *Computers & Chemical Engineering*, 16(10-11):937 – 947, 1992. An International Journal of Computer Applications in Chemical Engineering.

[15] Tapio Westerlund and Frank Pettersson. An extended cutting plane method for solving convex minlp problems. *Computers & Chemical Engineering*, 19, Supplement 1(0):131 – 136, 1995. European Symposium on Computer Aided Process Engineering.