COUENNE: a user's manual

Pietro Belotti[⋆]

Dept. of Industrial & Systems Engineering, Lehigh University
200 W. Packer Ave, Bethlehem PA 18015.

Abstract. This is a short user's manual for the COUENNE open-source software for global optimization. It provides downloading and installation instructions, an explanation to all options available in COUENNE, and suggestions on when to use some of its features.

1 Introduction

COUENNE is an Open Source code for solving Global Optimization problems, i.e., problems of the form

$$\begin{array}{lll}(\mathbf{P}) \;\; \min & f(x) & \\ \quad\quad s.t. & g_j(x) \leq 0 & \forall j \in M \\ & x_i^l \leq x_i \leq x_i^u & \forall i \in N_0 \\ & x_i \in \mathbb{Z} & \forall i \in N_0^I \subseteq N_0,\end{array}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and, for all $j \in M$, $g_j : \mathbb{R}^n \to \mathbb{R}$ are multivariate (possibly nonconvex) functions, $n = |N_0|$ is the number of variables, and $x = (x_i)_{i \in N_0}$ is the n-vector of variables. We assume that f and all g_j's are *factorable*, i.e., they are expressed as $\sum_h \prod_k \eta_{hk}(x)$, where all functions $\eta_{hk}(x)$ are univariate.

COUENNE is part of the COIN-OR infrastructure for Operations Research software[1]. It is a reformulation-based *spatial branch&bound* (sBB) [10,11,12], and it implements:

- linearization;
- branching;
- heuristics to find feasible solutions;
- bound reduction.

Its main purpose is to provide the Optimization community with an Open-Source tool that can be specialized to handle specific classes of MINLP problems, or improved by adding more efficient linearization, branching, heuristic, and bound reduction techniques.

[⋆] Email: `belotti@lehigh.edu`

[1] See `http://www.coin-or.org`

Web resources. The homepage of COUENNE is on the COIN-OR website:

`http://www.coin-or.org/Couenne`

and shows a brief description of COUENNE and some useful links. The COUENNE project page can be found at `https://projects.coin-or.org/Couenne`. It provides quick installation instructions and other useful information on the project.

Mailing lists. Questions, suggestions, complaints, and wish lists about COUENNE can be sent to the mailing list address `couenne@lists.coin-or.org`. Visit `http://list.coin-or.org/mailman/listinfo/couenne` for subscription options.

In order to submit bug reports, the COIN-OR framework uses a *ticket* system[2]. If you think you have found a bug in COUENNE, you are encouraged to register at COIN-OR and submit a ticket on the project webpage at

`https://projects.coin-or.org/Couenne/report`

Tickets can also be submitted discussed on the COUENNE ticket mailing list:

`http://list.coin-or.org/mailman/listinfo/couenne-tickets`

Licensing. COUENNE is Open-Source and is subject to the Common Public License [7]. As a user, you are responsible for checking that the use you make of COUENNE abides by the CPL rules.

Remark. As for all software, COUENNE evolves. This manual is periodically updated when new features are added to COUENNE, therefore the version you are reading may not be the latest one. Check `http://www.coin-or.org/Couenne` for the updated version.

2 Download and installation

As for many packages in the Coin-OR infrastructure, COUENNE has a "stable" and a "trunk" version. The former is recommended for almost all users, as it has been thoroughly tested and provides an optimal solution to all instances tested — its only disadvantage is that it is not the cutting edge version. The "trunk" version, instead, contains the most recent features that are, however, still being tested. Therefore, it does not guarantee its results: it may (although in some rare cases) return a sub-optimal solution, an infeasible solution, or may just crash.

COUENNE can be downloaded in three ways: as a Subversion repository, as a source code tarball, or as a binary tarball.

[2] See `http://www.coin-or.org/faqs.html#q10`

Subversion[3]**.** On *nix operating systems, simply run the following command:

```
svn co https://project.coin-or.org/svn/Couenne/stable/0.1 Couenne-0.1
```

if you wish to download the stable version, otherwise

```
svn co https://project.coin-or.org/svn/Couenne/trunk Couenne-trunk
```

will download the trunk version. Both commands create a local Subversion repository, which is a copy of what is currently being developed in any of the two versions. Although a copy of the source code will require that you build the source, the process is usually straightforward and requires only a few commands. The other advantage of an `svn` copy is that, in order to *update* your copy with the latest features or bugfixes, you simply have to run the command

```
svn update
```

from the directory of your copy. For Windows[4] systems, several `svn` clients are available, such as TortoiseSVN[5].

Source tarball. as a source tarball, directly from the Coin-OR project website:

`http://www.coin-or.org/download/source/Couenne`

Binary tarball. COUENNE is also available as a binary tarball, again from the Coin-OR website:

`http://www.coin-or.org/download/binary/Couenne`

While the last option gives you a pre-compiled version of COUENNE, the first two options allow to download and modify its source code.

2.1 Installing COUENNE

These instructions are necessary when downloading the source code, i.e. through `svn` or as a tarball. We refer to general installation instructions for COIN-OR projects at `https://projects.coin-or.org/BuildTools`. In short, first run the following command (assuming you have checked out the stable version):

```
cd Couenne-0.1/ThirdParty
```

and download all the *third party* software tools that will be used in building COUENNE: the Ampl Solver Library (ASL), Blas, Lapack, and MUMPS or HSL. Instructions on how to download them are included in the `INSTALL.*` file in each directory. Then, in order to build COUENNE proper you need to run the following commands (assuming you have gone back to the `Couenne-0.1` directory)

[3] See `http://subversion.tigris.org` for details.
[4] Windows is a trademark of Microsoft Corporation.
[5] See `http://tortoisesvn.tigris.org`

```
mkdir build
cd build
../configure -C
make
make install
```

The above commands place Couenne in the `Couenne/build/bin/` directory, libraries in `Couenne/build/lib/`, and include files in `Couenne/build/include/`. An alternative directory can be specified with the `--prefix` option of configure. For instance, when replacing "`../configure -C`" above with

```
../configure -C --prefix=/usr/local
```

the Couenne executable will be installed in `/usr/local/bin/`, the libraries in `/usr/local/lib/`, and the include files in `/usr/local/include/`, assuming you have writing permission in these directories. Couenne is run as follows:

```
couenne instance.nl
```

where `instance.nl` is an AMPL stub (`.nl`) file. Such files can be generated from AMPL with the command "`write gfilename;`" (notice the "`g`" before the file name), for example.

You may also specify a set of options to tweak the performance of Couenne. These are found in the couenne.opt option file. A sample option file is given in the `Couenne/src/` directory.

3 How COUENNE works

Couenne is a reformulation-based branch-and-bound. The initial problem is reformulated by introducing a new set of variables, called *auxiliary* variables. After reformulation, the problem looks as follows:

$$\begin{array}{lll} (\mathbf{P}') \ \min & w_{n+q} & \\ s.t. & w_i = \vartheta_i(x, w_{n+1}, w_{n+2} \dots, w_{i-1}) & i \in Q \\ & w_i^l \leq w_i \leq w_i^u & i \in Q \\ & x_i^l \leq x_i \leq x_i^u & i \in N_0 \\ & x_i \in \mathbb{Z} & i \in N_0^I \subseteq N_0 \\ & w_i \in \mathbb{Z} & i \in Q_I \subseteq Q. \end{array}$$

Reformulation does not make the problem easier to solve, as it simply creates a bunch of new variables. It is easier, however, to obtain a lower bound on the optimal solution of $(\mathbf{P}')$ than it is for $(\mathbf{P})$. Whenever this manual mentions an auxiliary variable, it means a variable that has been created during this reformulation phase.

After reformulation, a linearization step allows to obtain a Linear Programming relaxation of $(\mathbf{P}')$ — and hence of $(\mathbf{P})$, which can be easily embedded in a branch-and-bound framework. COUENNE adds the following components:

– **bound tightening** techniques: these are used to infer better bounds on all variables (both original and auxiliary), in order to get a tighter lower bound;
– **heuristics** to obtain a feasible solution;
– **branching techniques** for partitioning the set of solutions.

The options discussed below allow to tweak the performance of COUENNE on these three components. For a thorough discussion on their meaning, we refer to the introductory work on COUENNE by Belotti et al. [2].

4 Using COUENNE

While we are working to make COUENNE available from several modeling languages, it currently accepts files in AMPL nonlinear format, i.e., files with a `.nl` extension. Therefore, there are currently two ways to use COUENNE:

– as a solver from the AMPL modeling language [5,6];
– as a standalone solver that reads files with extension `.nl`.

Suppose you want to use COUENNE from AMPL and you have a problem described in a `mymodel.mod` file as follows:

```
var x1 >= 1 <= 2;
var x2 >= 0 integer;
minimize obj: x1^2.23 + 2*x2;
subject to c1: x1^2 + x2^2 <= 4;
subject to c2: x1 + x2 >= 0;
```

Then you can issue the following commands to AMPL:

```
ampl: model mymodel.mod
ampl: option solver couenne;
ampl: solve;
```

If, instead, you wish to run COUENNE on your own `.nl` files, just run

```
$ couenne myproblem.nl
```

at the command line. Notice that the `couenne` executable must be visible from the command line, i.e., the directory where the executable is stored must be in some `PATH` environment variable.

Option file. Couenne reads its run-time parameters from option file `couenne.opt`. All parameters that are *not* specified in the option file are set to their default value. All options are discussed in Section 6.

5 Output of COUENNE

Suppose we are solving the *ex1221* instance of the `minlplib` library of MINLP problems [4], translated into AMPL format through the `convert` utility of the GAMS modeling language [3]. The original problem is as follows:

$$
\begin{array}{ll}
\min 2x_0 + 3x_1 + 1.5y_2 + 2y_3 - 0.5y_4 & \\
s.t.\ x_0^2 + y_2 & = 1.25 \\
\quad x_1^{1.5} + 1.5y_3 & = 3 \\
\quad y_2 + x_0 & \leq 1.6 \\
\quad y_3 + 1.333x_1 & \leq 3 \\
\quad y_4 - y_3 - y_2 & \leq 0 \\
\quad 0 \leq x_0 \leq 10 & \\
\quad 0 \leq x_1 \leq 10 & \\
\quad y_2, y_3, y_4 \in \{0, 1\}. &
\end{array}
$$

Without any `couenne.opt` option file, the output of COUENNE run on *ex1221* is as follows:

```
objectives:
min ( +2*x_0 +3*x_1 +1.5*y_2 +2*y_3 -0.5*y_4)
constraints:
((x_0^2) +1*y_2) = 1.25
((x_1^1.5) +1.5*y_3) = 3
( +1*y_2 +1*x_0) <= 1.6
( +1*y_3 +1.333*x_1) <= 3
( +1*y_4 -1*y_3 -1*y_2) <= 0
variables:
x_0 [ 0 , 10 ]
x_1 [ 0 , 10 ]
y_2 binary
y_3 binary
y_4 binary
end
Problem size before reformulation: 5 variables (3 integer), 5 constraints.
Problem size after  reformulation: 13 variables (4 integer), 3 constraints.
NLP0012I
              Num      Status      Obj             It       time
NLP0013I     1        OPT         7.667180052199679       9        0.012
Cbc0012I Integer solution of 7.66718 found by Init Rounding NLP
         after 0 iterations and 0 nodes (0.00 seconds)
NLP0013I     2        OPT         7.667180051588896       3        0.004
Cbc0001I Search completed - best objective 7.667180068813135, took 0 iterations
         and 0 nodes (0.01 seconds)
Cbc0035I Maximum depth 0, 0 variables fixed on reduced cost

couenne: Optimal

        "Finished"
```

The first part of the output is a visualization of the problem itself. Variable names may be different from the AMPL version. COUENNE uses a similar indexing standard to AMPL (the first variable indexed by zero), but a different naming standard, which helps in case you print out the reformulated version of the problem. For example, you may see the following variable names:

- `x_3` is the fourth variable. It is an original variable of the problem, and is continuous;
- `y_5` is the sixth variable of the problem, an original variable, and constrained to be integer;
- `w_34` is the 35-th variable; it is auxiliary, that is, it was introduced by reformulation, and is continuous (or at least COUENNE couldn't understand if it is constrained to be integer);
- `z_43` is the 44-th variable; it is auxiliary and associated with a function that can only take on integer values.

The visualization of the problem is rather straightforward, and is closed by the keyword "`end`." The following line shows that the original problem has five variables, three of which are integer, and five constraints. The line following it shows that, after reformulation, there are 13 variables (including original and auxiliary), four of which are integer (so there is an integer auxiliary), and three constraints. Two constraints (the first two) have been eliminated from the original problem because they are of the form

$$a_k x_k + f(x_1, x_2 \ldots, x_{k-1}, x_{k+1} \ldots, x_n) = c$$

so COUENNE figured out it might just change x_k into an auxiliary associated with the function $-\frac{1}{a}\left(f(x_1, x_2 \ldots, x_{k-1}, x_{k+1} \ldots, x_n) + c\right)$.

The following output comes from the three fundamental COIN-OR software packages that COUENNE uses: `Cbc`, `Clp`, and `Ipopt`. Lines starting with `Cbc` are output from the branch-and-bound interface, and show number of total nodes created, number of unvisited nodes, and current upper/lower bound, or show a new feasible solution for the MINLP whenever one is found, or again a summary line when the optimization has ended. Output by `Clp` is switched off by default, while output by `Ipopt` consists in the objective value of a solution found.

6 Options

Options are specified in a file named `couenne.opt`, which has to be in the same directory from which COUENNE is run. Each option is specified with the format

option_name value

and anything between a "#" and the end of the line is ignored. There is a sample `couenne.opt` file in the `Couenne/src` directory of the source code, with comments and hints on how to use/change these options. The following is a list of options that allow to tweak the result of COUENNE, and it is expected to change as new features are introduced. Next to each option name, its default value is given in brackets.

6.1 Output options

These options control the amount and type of output of COUENNE. Although most of these options are for debugging purposes, some of them provide useful (and limited) output. By default, all but `problem_print_level` are zero as they produce a lot of output that is in general not needed.

- `problem_print_level` (4): this is probably the most useful. An option value of 4 prints out the initial problem, while a value of 7 prints the reformulated problem as well.
- `branching_print_level` (0): branching rules.
- `boundtightening_print_level`: bound tightening.
- `convexifying_print_level` (0): generation of linearization cuts.
- `disjcuts_print_level` (0): (verbose) output on the generation of disjunctive cuts.
- `nlpheur_print_level`: heuristics.
- `reformulate_print_level` (0): output on the reformulation phase.

Print levels for other parts of COUENNE are as follows:

- `lp_log_level` (0): `Clp` output level;
- `mip_log_level` (1): `Cbc` output level;
- `nlp_log_level` (1): `Ipopt` output level.

Finally, if the option `display_stats` is set to “yes,” a line at the end of execution is printed with some data about the optimization (lower-upper bound, branch-and-bound time, separation time, etc.).

6.2 Linearization options

- `convexification_cuts` (4): number of rounds of cuts applied at each branch-and-bound node.
- `convexification_points` (3): for the lower envelopes of convex functions, this is the number of points where a supporting hyperplane is generated. This only holds for the initial linearization, as all other linearizations only add at most one cut per expression.
- `delete_redundant` (yes): some problems are reformulated in such a way that some auxiliary variables are associated with functions that are other variables, or $w_i = x_j$. By default, COUENNE gets rid of w_i, but by setting this option to “no” it will keep it as a variable.
- `use_quadratic` (no): this is still under testing. Quadratic form are not reformulated and therefore decomposed as a sum of auxiliary variables, each associated with a bilinear term, but rather taken as a whole expression. Envelopes for these expressions are generated through α-convexification [1].
- `violated_cuts_only` (yes): only add those linearization cuts that are violated.

6.3 Branching options

- branch_conv_cuts (no): after applying a branching rule and before re-solving the subproblem, generate a round of linearization cuts with the new bounds enforced by the rule (currently not working).
- branch_fbbt (yes): after applying a branching rule and before re-solving the subproblem, apply Bound Tightening.
- branching_object (var_object): the source of infeasibility of a problem. This option can take one of the following values: "var_obj", "expr_obj", and "vt_obj". The first indicates that infeasibility is associated with each variable. For example, suppose the optimal solution to the LP relaxation is denoted by $x^\star$. If two auxiliary variables $w_2 = f_2(x_1)$ and $w_3 = f_3(x_1)$ have an optimal value in the LP solution such that $w_2^\star \neq f_2(x_1^\star)$ and $w_3^\star \neq f_3(x_1^\star)$, then x_2 will be associated an "infeasibility" measure dependent on $|w_2^\star - f_2(x_1^\star)|$ and $|w_3^\star - f_3(x_1^\star)|$. With option "expr_obj," the infeasibility is attributed to auxiliaries w_2 and w_3, but this is not recommended. Finally, using "vt_obj" allows to use *Violation Transfer*, a branching variable selection technique presented by Tawarmalani and Sahinidis [14]. A better treatment of the branching process is given by Belotti et al. [2].
- cont_var_priority (1000): branching priority of continuous variables. When branching, this is compared to the priority of integer variables, whose priority is fixed to 1000. Higher values mean smaller priority, so if this parameter is set to 1001 or higher, if a branch-and-bound node has at least one integer variable whose value is fractional branching will be performed on that variable.
- enable_sos (no): use SOS branching objects, that are generated from constraints of the form $\sum_{h=1}^{k} y_h = 1$ with all variables y_h binary (still testing).

6.4 Bound tightening options

Bound tightening is a very important part of COUENNE. It allows to substantially reduce the feasible set, and therefore the total running time.

- feasibility_bt (yes): apply Feasibility-Based Bound Tightening (FBBT), a pre-processing technique to reduce the bounding box, before the generation of linearization cuts. This is a quick and effective way to reduce the solution set, and it is highly recommended to keep it active.
- redcost_bt (yes): Use reduced costs of the LP in order to infer better variable bounds.
- aggressive_fbbt (yes): apply Aggressive FBBT [2], a version of probing [8,13] that also allow to reduce the solution set, although it is not as quick as FBBT. It can be applied up to a certain depth of the branch&bound tree — see below. In general, this option is useful but can be switched off if a problem is too large and seems not to benefit from it.
- optimality_bt (yes): apply Optimality-Based Bound Tightening (OBBT) [9], which is another bound reduction technique aiming at reducing the solution set by looking at the initial LP relaxation. This technique is computationally expensive, and should be used only when necessary.

- `log_num_abt_per_level` (3): depth of the branch&bound tree at which usage of Aggressive FBBT should be reduced. As mentioned above, this bound tightening technique is rather time consuming, so it is useful to limit its application to all branch&bound nodes up to a certain depth d, which is by default equal to 3. At depths $t > d$, Aggressive FBBT will be applied with probability 2^{d-t}.
- `log_num_obbt_per_level` (4): Similarly to the previous option, this allow to limit the number of times OBBT is used by setting a branch&bound depth below which the usage of OBBT is reduced.

6.5 Disjunctive cut options

A recently added feature is the generation of disjunctive cuts. Cuts that are guarantee to eliminate suboptimal solutions are generated by using disjunctions naturally arising within a nonconvex MINLP. Their generation reflects the separation of similar cuts for Mixed-Integer Linear Programming and, in this initial version, is rather time consuming.

- `minlp_disj_cuts` (0): how often to generate disjunctive cuts. A "0" means they are never generates, while any positive number n instructs COUENNE to generate them at every n nodes of the branch&bound tree. A negative number $-n$ means that generation should be attempted at the root node, and if successful it can be repeated at every n nodes, otherwise it is stopped altogether.
- `disj_depth_level` (3): at what depth of the branch&bound tree generation of disjunctive cuts should be reduced. This has a similar behavior as `log_num_obbt_per_level`. A value of -1 means generation can be done at all nodes.
- `disj_depth_stop` (10): at what depth of the branch&bound tree generation of disjunctive cuts should be stopped. A value of -1 means generation is not stopped
- `disj_active_cols` (yes): only consider violated variable bounds when creating the Linear Programming problem (the so-called CGLP, Cut Generating LP); this reduces the size of the CGLP, but may produce less efficient cuts.
- `disj_active_rows` (yes): only consider violated linear inequalities when creating the CGLP. Similar considerations apply.
- `disj_cumulative` (no): when generating disjunctive cuts on a set of disjunctions $1, 2 \ldots, k$, introduce the cut relative to the previous disjunction $i - 1$ in the CGLP used for disjunction i. Notice that, although this makes the cut generated more efficient, it increases the rank of the disjunctive cut generated.
- `disj_init_number` (10): maximum number of disjunctions to be used at every round of disjunctive cuts.
- `disj_init_perc` (0.2): maximum percentage of all disjunctions currently violated by the problem used for generating cuts (to be compared with the parameter above to decide how many disjunctions to actually use for generating cuts).

6.6 Nonlinear solver options

– `local_optimization_heuristic` (yes): use a heuristic with `Ipopt` to find feasible solutions for the problem. It is highly recommended that this option be set to active, as it would be difficult to find feasible solutions otherwise.
– `log_num_local_optimization_per_level` (2): at what depth of the branch&bound tree should the calls to the heuristic be reduced. Behavior similar to `log_num_obbt_per_level`.

6.7 Tolerance options

– `art_cutoff` ($+\infty$): cutoff for the problem. Useful to fathom all nodes known to have a lower bound above this cutoff value, it reduces the total CPU time. Notice that it is useful in practice only when a solution whose value is equal to the cutoff is known.
– `art_lower` ($-\infty$): value of a lower bound to the optimal solution.
– `feas_tolerance` (10^{-6}): tolerance to be applied when checking for the feasibility of a MINLP solution. If a constraint $g_i(x) \leq 0$ within this tolerance, it is deemed satisfied.

6.8 Debug options

The general COUENNE user won't need these options. They can be rather useful when testing new components of COUENNE in order to check their validity.

A first debugging feature of COUENNE is the possibility to store an optimal solution in a file. The file must be named as the `.nl` instance file, but with extension `.txt` instead, and must be stored in the same directory where COUENNE is called. For instance, if the current directory is `~/Couenne/build/` and we want to debug COUENNE on an instance that has 12 variables and that is stored in `~/myInstances/trouble.nl`, we may write a file `~/Couenne/build/trouble.txt` containing 13 lines: the first line contains the value of the optimal solution, and the remaining 12 contain the value of the variables in an optimal solution.

In order to check whether COUENNE is correct, set option `problem_print_level` to 7. If a branch&bound node that is supposed to contain that optimal solution is fathomed, or an inequality is added that cuts off the optimal solution, a warning message will be displayed. Other options are below:

– `check_lp` (no): saves the `.lp` file of any LP generated to obtain a lower bound. Unless the problem is small, this will create a lot of files, slow down COUENNE significantly, and occupy a lot of hard disk space.
– `couenne_check` ($+\infty$): the value of a known optimal solution for the problem. Used to check whether COUENNE indeed finds an optimal solution.
– `opt_window` ($+\infty$): if an optimal solution x^{opt} is provided as described above, this value allows to restrict the lower and upper bound of each variable x_i to $[x_i^{\mathrm{opt}} - \texttt{opt\_window} \cdot (1 + |x_i^{\mathrm{opt}}|), x_i^{\mathrm{opt}} + \texttt{opt\_window} \cdot (1 + |x_i^{\mathrm{opt}}|)]$.
– `test_mode` (no): avoid pop-up windows reporting a memory access violation (which happens when debugging).
– `time_limit` (7200): CPU time limit in seconds.

Options of `Bonmin`, `Cbc`, `Clp`, and `Ipopt`. The option file can also contain option from the other components of COUENNE, namely `Bonmin`, `Cbc`, `Clp`, and `Ipopt`. These have to be specified with a prefix equal to the corresponding name, so if you want to specify the `Bonmin` option to use a B-BB algorithm, add a line

```
bonmin.algorithm B-BB
```

while all COUENNE options do not need any "`couenne.`" prefix.

References

1. I. P. Androulakis, C. D. Maranas, and C. A. Floudas. αBB: A global optimization method for general constrained nonconvex problems. *Journal of Global Optimization*, 7(4):337–363, December 1995.
2. P. Belotti, J. Lee, L. Liberti, F. Margot, and A. Wächter. Branching and bounds tightening techniques for non-convex MINLP. *Optimization Online*, 2008/08. http://www.optimization-online.org/DB_HTML/2008/08/2059.html.
3. A. Brooke, D. Kendrick, and A. Meeraus. *GAMS: A User's Guide, Release 2.25*. The Scientific Press, 1992. `http://citeseer.ist.psu.edu/brooke92gams.html`.
4. M.R. Bussieck, A.S. Drud, and A. Meeraus. MINLPLib – a collection of test models for mixed-integer nonlinear programming. *INFORMS Journal of Computing*, 15(1):114–119, 2003. `http://www.gamsworld.org/minlp/minlplib/minlpstat.htm`.
5. R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: a modeling language for mathematical programming*. Boyd and Fraser Publishing Company, 1993.
6. D.M. Gay. Hooking your solver to AMPL. Technical Report 93-10, AT&T Bell Laboratories, Murray Hill, NJ, 1993, revised 1997.
7. Open Source Initiative OSI. Common public license version 1.0. `http://www.opensource.org/licenses/cpl1.0.php`.
8. M.W.P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6(4):445–454, 1994.
9. J.P. Shectman and N.V. Sahinidis. A finite algorithm for global minimization of separable concave programs. *Journal of Global Optimization*, 12:1–36, 1998.
10. E.M.B. Smith. *On the Optimal Design of Continuous Processes*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, October 1996.
11. E.M.B. Smith and C.C. Pantelides. Global optimisation of nonconvex MINLPs. *Computers & Chem. Engineering*, 21:S791–S796, 1997.
12. E.M.B. Smith and C.C. Pantelides. A symbolic reformulation/spatial branch-and-bound algorithm for the global optimisation of nonconvex MINLPs. *Computers & Chem. Eng.*, 23:457–478, 1999.
13. M. Tawarmalani and N.V. Sahinidis. *Convexification and global optimization in continuous and mixed-integer nonlinear programming: Theory, algorithms, software and applications*, volume 65 of *Nonconvex Optimization and Its Applications*. Kluwer Academic Publishers, Dordrecht, 2002.
14. M. Tawarmalani and N.V. Sahinidis. Global optimization of mixed-integer nonlinear programs: A theoretical and computational study. *Mathematical Programming*, 99(3):563–591, 2004.