GALINI: An extensible mixed-integer quadratically-constrained optimization solver

F. Ceccon[a], R. Baltean-Lugojan[a], M. L. Bynum[b], C. Li[a] and R. Misener[a]

[a]Department of Computing, Imperial College London, 180 Queens Gate, SW7 2AZ, UK
[b]Center for Computing Research, Sandia National Laboratories P.O. 5800, Albuquerque, NM, 87185, USA

ARTICLE HISTORY
Compiled January 14, 2021

ABSTRACT
We present GALINI, an open source solver for nonconvex mixed-integer quadratically-constrained quadratic programs formulated with the Python algebraic modeling library Pyomo. GALINI uses Pyomo to represent optimization problems and leverages the existing library ecosystem to implement different parts of the solver. GALINI includes a generic branch & bound algorithm that can be use develop new solvers. The GALINI branch & cut algorithm can be extended at runtime with new: (i) cutting planes, (ii) branching strategies, (iii) node selection strategies, (iv) primal heuristics, and (v) relaxations. We present computational studies to show GALINI performs comparably to existing open source solvers.

KEYWORDS
mixed-integer nonlinear optimization; solver software; deterministic global optimization; cutting planes

1. Introduction

GALINI is a new solver for nonconvex Mixed-Integer Quadratically Constrained Quadratic Problems (MIQCQP) [18, 23, 33] that is easy to extend. This paper describes GALINI version 1.0.0. GALINI is available through a permissive license (Apache 2.0) at `https://github.com/cog-imperial/galini` and can be installed using `pip install galini`. GALINI solves problems in the form:

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{x}^T Q_0 \boldsymbol{x} + a_0 \boldsymbol{x} \\ \text{s.t.} \quad & C_i^L \leq \boldsymbol{x}^T Q_i \boldsymbol{x} + a_i \boldsymbol{x} \leq C_i^U \qquad \forall i \in \{1, \dots, M\} \\ & \boldsymbol{x} \in \{\mathbb{R}^C, \mathbb{Z}^I\} \end{aligned}$$

where Q_0 is the matrix of quadratic objective coefficients, Q_i the matrix of quadratic constraint i coefficients, a_0 the vector of linear objective coefficients, a_i the vector of linear constraint i coefficients, $C_i^L \in \{\mathbb{R}, -\infty\}$ the constraint i lower bound, and $C_i^U \in \{\mathbb{R}, +\infty\}$ the constraint i upper bound. We assume the $N = C+I$ variables $\boldsymbol{x}$ are

CONTACT R. Misener. Email: r.misener@imperial.ac.uk

Table 1. Notation used to develop the GALINI solver

Symbol	Description
$\mathbb{S}^N$	$\{ \mathbb{R}^C, \mathbb{Z}^I \}$
C, I	Number of continuous (C) and integer (I) variables
N	Total number of variables, $N = C + I$
M	Number of constraints
$\boldsymbol{x} \in \mathbb{S}^N$	A vector of variables
x_i	The i-th element of $\boldsymbol{x}$
x_i^L, x_i^U	Lower and upper bound of x_i
$\hat{\boldsymbol{x}}$	Value of the linear relaxation solution
w_{ij}	Auxiliary variable representing $x_i x_j$
w_f	Auxiliary variable representing $f(\boldsymbol{x})$
Q_0, Q_i	Matrix of quadratic coefficients in the objective and i-th constraint
a_0, a_i	Vector of linear coefficients in the objective and i-th constraint
C_i^L, C_i^U	Lower and upper bound of the i-th constraint
c^L, c^U	Lower and upper bound of an expression $f(\boldsymbol{x})$
z^L, z^U	Objective value of the best feasible (z^U) and best possible (z^L) solutions

bounded ($x_i \in [x_i^L, x_i^U], x_i^L \neq -\infty, x_i^U \neq +\infty$). If the user model contains unbounded variables, GALINI will try to deduce variable bounds but, if GALINI cannot deduce variable bounds, then GALINI may not converge. The base GALINI distribution does not artificially impose bounds on unbounded variables, so there is no guarantee of algorithm convergence if there are missing variable bounds. If a user-installed plug-in artificially imposes variable bounds, GALINI may converge to a solution that is feasible but not globally optimal.

We assume that all optimization problems minimize the objective function. GALINI handles maximization problems internally by disabling the original objective function $f(\boldsymbol{x})$ and adding a new objective function $-f(\boldsymbol{x})$. The user only experiences interaction with the original objective, but GALINI works with the modified objective function.

Applications of MIQCQP include: pooling problems [6, 7, 13, 70, 72] and crude oil scheduling [26, 55, 57, 58] in petrochemicals, heat exchange networks in energy efficiency, [38, 66, 67, 78], water distribution networks in environmental engineering [35, 36, 42, 51], and other engineering challenges [18, 23, 33, 44, 45]. Existing software addressing MIQCQP includes: αBB [5], αECP [94], Alpine [75, 76], ANTIGONE [71, 73], AOA [50], BARON [86], Bonmin [21], Couenne [16, 62], CPLEX [20], DICOPT [92], EAGO [95], FilMINT [2], Gurobi, Juniper [53], Knitro [25], LaGO [79], LINDOGlobal [43, 60], MAiNGO [22, 77], MILANO [19], MINLPbb [37], MINOTAUR [65], Mosek, Muriqui [69], NOMAD [54], Octeract, Pajarito.jl [29] Pavito [29], SBB GAMS, SCIP [3, 90] and SHOT [64]. Figure 1 illustrates a timeline of solver releases.

GALINI targets two types of users:

- **Developers** invent new algorithms or improve existing algorithms to solve MIQCQP. Developers may be (i) *algorithm developers* who create specific solver components, e.g. a new class of cutting planes, and (ii) *solver developers* who engineer new solvers by combining different components together.
- **End Users** solve MIQCQP application instances.

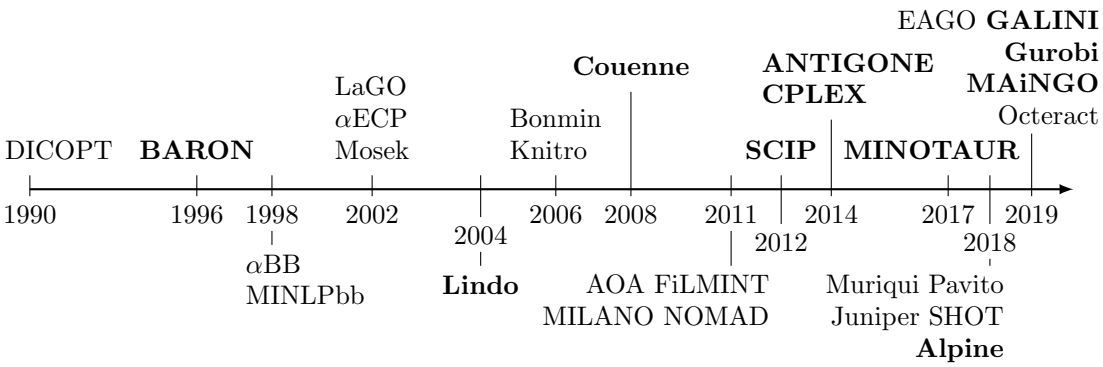

Figure 1. Timeline of Mixed-Integer Non Linear Problems (MINLP) solvers and when they were first released. Solvers that appear in bold can solve nonconvex problems to global optimality.

All of the Figure 1 solvers target end users. SCIP and MINOTAUR additionally target developers: SCIP provides a plug-in architecture and MINOTAUR provides a toolkit to build new solvers. Unlike SCIP and MINOTAUR, GALINI is distributed as a standalone package that can be extended by loading new plug-ins at runtime. Solver developers can use GALINI as a solver toolkit similar to MINOTAUR, while algorithm developers can extend GALINI dynamically without modifying the underlying solver. The advantage of this approach is that end users can experiment with plug-ins developed by different developers and only need to install GALINI once.

GALINI can be used in two modes:

- The `galini` **Python Package** provides classes and functions for developing optimization algorithms. It can also run GALINI as part of more complex scripts.
- The `galini` **Command Line Tool** solves optimization problems (defined as Python or OSiL files) using GALINI. A configuration file may specify non-default values for different GALINI options.

GALINI plug-ins can be developed without modifying GALINI source code and can be distributed independently from GALINI. Users can decide which plug-ins to install. Users can also decide which plug-ins to activate by updating the configuration file passed to the GALINI command line tool. As Section 5.1 describes, plug-ins are registered on the user machine using a system-wide registry of Python classes, also known as *entry points*. The Section 5 branch & cut algorithm provides several extension points: cutting planes, primal heuristic, node selection strategy, branching strategy, and initial feasible solution search strategy.

2. GALINI Foundation

2.1. Ecosystem of Python libraries

GALINI builds on top of existing mixed-integer nonlinear optimization (MINLP) packages for Python, as shown in Figure 2.

- The **Pyomo** [46, 47] modeling library represents the optimization problem. Pyomo also interfaces GALINI to the mixed-integer linear optimization (MIP) and nonlinear optimization (NLP) solvers in a solver-independent way.
- **Coramin** [30] performs Optimization Based Bounds Tightening and computes

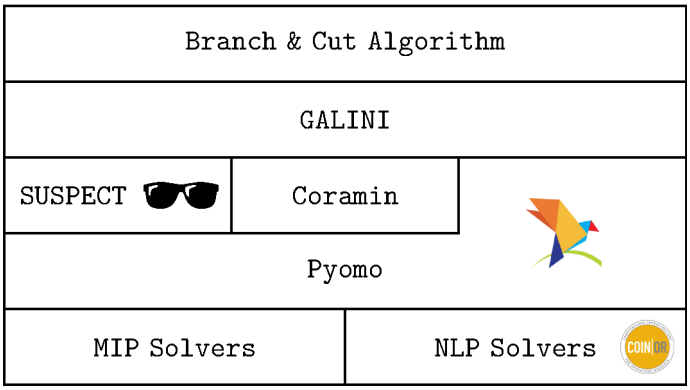

Figure 2. GALINI sits on top of a solid ecosystem of Python libraries. GALINI uses Pyomo to represent optimization problems, SUSPECT to find convexity information and perform bounds tightening, and Coramin for bounds tightening and building relaxations. Linear and nonlinear problems are solved using the Pyomo interface to existing MIP and NLP solvers. Solver developers can build algorithms on top of GALINI.

rigorous relaxations of the Pyomo model.

- **SUSPECT** [28] performs Feasibility Based Bounds Tightening (FBBT) and computes convexity and monotonicity information.
- GALINI is agnostic to MIP and NLP sub-solvers because of its integration with Pyomo. For developers interested in a persistent NLP interface, our new Pyomo solver pypopt[1] uses the **Ipopt** [93] C++ interface to access Ipopt directly.

2.2. Coramin & SUSPECT: Variable Bound Inference & Tightening

GALINI uses Coramin for *optimality-based bounds tightening* (OBBT) [16]. OBBT is expensive, so GALINI only uses it at the root node of the branch & bound tree.

GALINI uses the less expensive *feasibility-based bounds tightening* (FBBT) to reduce variable bounds at the beginning of each node [16, 28, 40]. GALINI uses SUSPECT for FBBT [28]. We're using SUSPECT 2.1.1 to infer variable bounds on quadratic terms like $q_{ii}x_i^2 + a_i x_i$ [31, 91, 89]. Because of computational expense, GALINI skips the FBBT step on expressions with a large number of terms (`fbbt_max_quadratic_size`, default = 100; `fbbt_max_expr_arguments`, default = 100).

If GALINI cannot infer a variable's bounds, it maintains the bounds at infinity. These infinite bounds are passed to third party solvers. The base GALINI code does not impose artificial variable bounds, but user-installed plug-ins requiring bounded variables may decide to use the user configurable constant `infinity` (see Section 3.1).

2.3. SUSPECT: Special Structure Detection

After FBBT at each node, GALINI detects convexity and monotonicity using SUSPECT. The resulting special structure information is accessible to all parts of the Section 5 branch & cut algorithm. The convexity information is used, e.g. when computing the convex relaxation. Time spent computing special structure is negligible compared to FBBT, so GALINI recomputes special structure at every node without any code optimization.

[1] https://github.com/cog-imperial/pypopt

2.4. Coramin: Convex Relaxations

GALINI uses Coramin[2] to include classes and functions that develop linear relaxations from Pyomo models. Coramin is a Python package built on Pyomo which contains functions and classes for developing global optimization algorithms. Coramin supports the development of both branch & bound and multitree algorithms [18, 61]. Coramin provides classes that form the building blocks for developing convex relaxations of nonconvex Pyomo models. The goal of these classes is threefold:

(1) aid in the refinement of convex relaxations as algorithms proceed,
(2) provide a mechanism for developing custom relaxations for specific applications,
(3) provide building blocks for relaxing general Pyomo models through factorable programming, i.e. the auxiliary variable method.

Coramin has classes for relaxations (overestimators and underestimators) of univariate, convex or concave functions, e.g. natural logarithm, bilinear terms, and some trigonometric functions. Although most, more complex, constraints can be relaxed using these building blocks through reformulation and factorable programming, Coramin also has a class for generating αBB-based relaxations [5]. Finally, Coramin contains a class for underestimators of convex, multivariate functions or overestimators of concave, multivariate functions. These classes include methods for refining the relaxations as variable bounds change, adding outer-approximation cuts for convex or concave functions, and generating piecewise convex relaxations. These classes inherit from Pyomo Blocks, allowing seamless integration into Pyomo models. These classes also have explicit support for Pyomo's persistent solver interfaces, providing an efficient mechanism for resolving models as relaxations are updated. Appendix A shows how these Coramin classes can be used to build and refine convex relaxations.

3. GALINI Initialization & Pre-processing

3.1. The Solver Object

Since GALINI can be used in larger scripts, GALINI does not use global variables to track the solver configuration. The GALINI package defines a `Galini` class with:

- A **Current Configuration** obtained by modifying the default configuration with any end user's configuration, e.g. any parameter changes from the default.
- A **Telemetry** object to log information about the solving process, e.g. the time spent solving relaxations.
- A **Logger** passes textual information to the user.
- The **Time Limit** remaining.
- The mathematical constants **Math Context** used in the solving process.
- Developers can activate the **Paranoid Mode Flag** for extra checks in GALINI. This flag is turned off by default since it impacts performance.

The math context, which tracks constants that can be optionally changed by users, is a property of the current `Galini` object instance and is passed as an argument to all numerical functions inside GALINI. The math context has the following constants:

- `epsilon`: a small quantity, defaults to 10^{-5}

[2] https://github.com/Coramin/Coramin

- `infinity`: a large quantity, numbers greater than this constant are considered ∞, defaults to 10^{20}.
- `constraint_violation_tol`: absolute tolerance used to consider a constraint violated.

We define as the *relative difference* $d(a, b)$ between two numbers a and b:

$$d(a,b) = \begin{cases} |a-b| / \max\{|a|,|b|\} & \text{if } a \neq 0, b \neq 0 \\ \infty & \text{if } |a| = \infty \text{ or } |b| = \infty \\ |a-b|/\epsilon & \text{otherwise,} \end{cases} \tag{1}$$

where ϵ is the `epsilon` from the math context and 0 is defined as being within `epsilon` from the math context.

3.2. GALINI Problem Representation

GALINI uses Pyomo [46, 47] expression types to represent operations. By default, Pyomo represents models with a separate expression tree for each constraint and objective. GALINI provides a function to modify the Pyomo model so that common sub-expressions are grouped together [28, 89, 91] to form a directed acyclic graph.

GALINI extends the Pyomo expression types to include a `QuadraticExpression` type. The quadratic expression implementation is transparent to Pyomo; code that does not know how to handle it will treat it as a `SumExpression` of bilinear terms. In pre-processing, GALINI detects sums of bilinear terms and converts them to `QuadraticExpression`. The quadratic expression type provides convenient methods to get the bilinear terms and provides sparse representation in coordinate format.

4. Branch & Bound

GALINI implements generic `BranchAndBoundAlgorithm` that solver developers extend to implement concrete algorithms. Specifically, solver developers should subclass this abstract class `BranchAndBoundAlgorithm` for their own implementations. For example, the branch & cut algorithm described in Section 5 uses the branch & bound classes described in this section to implement branch & cut. The classes and interfaces `BranchAndBoundAlgorithm` uses are:

- `Tree`: the list of branch & bound nodes, together with information such as the global best feasible solution and best possible objective.
- `Node`: contains node-specific `NodeStorage` and the node local feasible solution (primal solution) and best possible objective (lower objective bound), if any.
- `NodeStorage`: contains algorithm-specific data, e.g. the Section 5 branch & cut algorithm uses this storage to keep a pointer to the original user model, its linear relaxation, and a list of cuts generated at a node [71, 74].
- `BranchingStrategy`: decides on which variable and at which point to branch.
- `NodeSelectionStrategy`: decides which node to visit next.

4.1. Lower-level classes of `BranchAndBoundAlgorithm`

`Tree` is the main class responsible for keeping track of the state of the branch & bound algorithm. The tree is initialized by specifying the root node storage, branching strategy, and node selection strategy.

The `BranchingStrategy` interface only requires implementing the `branch(node, tree)` method. This method takes as input the current node and the branch & bound tree, and returns one (or more) `BranchingPoint`. A branching point contains a variable and a list of points where that variable will be branched. This implementation, which allows simultaneously branching on multiple variables and at multiple points, helps developers investigate wide (non binary) branching strategies [15, 56]. As of version 1.0.0, GALINI supports branching on variables only.

The `NodeSelectionStrategy` interface has three methods:

- `insert_node(node)`: add a node to the list of nodes to visit,
- `has_nodes()`: return true if the selection strategy has more nodes to be visited,
- `next_node()`: return the next node to visit and remove it from the list to visit.

4.2. `BranchAndBoundAlgorithm`

Solver developers should extend the abstract `BranchAndBoundAlgorithm` class and implement the following abstract methods:

- `find_initial_solution(model, tree, node)`: called at the beginning of the algorithm to try to find a feasible solution before visiting the root node.
- `branching_strategy`: this property returns an instance of a class that implements the `BranchingStrategy`.
- `node_selection_strategy`: this property contains the node selection strategy.
- `extra_config`: this static property should return the algorithm-specific configuration options.
- `init_node_storage(model)`: called to create and initialize the `NodeStorage` used at the root node.
- `solve_problem_at_node(tree, node)`: called at each branch & bound node (except at the root node), should return a primal feasible solution and the best possible objective (objective lower bound) for the node, if any.
- `solve_problem_at_root(tree, node)`: similar to `solve_problem_at_node`, but called only at the root. Developers can use it for algorithm-specific initialization.

Listing 1 contains the definition of the `BranchAndBoundAlgorithm` class together with a simplified version of the branch & bound algorithm to show how the seven abstract methods that solver developers need to implement are used by the algorithm.

`BranchAndBoundAlgorithm` starts by creating the root node and setting the tree state to have a lower objective bound of negative infinity and an upper objective bound of positive infinity. The tree also initiates a solution pool of the best feasible solutions found (`bab.solution_pool_size`, default = 5). Finally, it adds the root node to the tree. When a node is added to the tree, it is also added to the list of open nodes and passed to the `insert_node` method of the node selection strategy.

The algorithm first visits the root with the `solve_problem_at_root(tree, node)` method. For all other nodes, `BranchAndBoundAlgorithm` calls `solve_problem_at_node(tree, node)`. The concrete implementations of both of these methods must return a `NodeSolution`, a class with two properties:

- `lower_bound_solution`: This object represents the solution of the lower bounding problem, e.g. a MIP or LP relaxation. This solution object must contain the solver status, e.g. optimal, infeasible, unbounded, and the solution objective value. In the concrete branch & cut implementation presented in Section 5, GALINI uses the MIP dual value as the solution of the lower bounding problem.
- `upper_bound_solution`: This object represents a feasible solution to the original optimization problem. This solution object must contain the solver status, objective value, and solution point (as a dictionary that maps Pyomo variables to their solution value).

After visiting the node, the branch & bound algorithm updates the tree lower and upper bound, checks for convergence and, if it did not converge, branches using `branching_strategy`. Branching creates new children nodes and adds them to the branch and bound tree. The algorithm then enters the branch and bound loop until one of the termination conditions are met:

- **Timeout**: the solver run time is greater than the user-specified time limit.
- **Maximum Number of Nodes Visited**: the algorithm has visited more than the maximum number of user-specified nodes (`node_limit`, default = 10^8).
- **Convergence**: the relative difference, i.e. the relative gap, between the tree best possible solution and the tree best feasible solution is within the user-specified convergence tolerance (`relative_gap`, default = 10^{-6}), or the absolute difference between the two is within another user-specified tolerance (`absolute_gap`, default = 10^{-8}).

The `node_selection_strategy` choses the next node to visit in the branch & bound algorithm. The branch & bound `Tree` tracks the global state of the branch & bound algorithm and has the following properties:

- `root`: a reference to the root node,
- `solution_pool`: a priority queue of the best feasible solutions,
- `open_nodes`: a collection of the tree's nodes that have not been visited yet,
- `fathomed_nodes`: a collection of the tree's nodes that have been fathomed. Nodes are fathomed when: (i) the node relaxation is infeasible, (ii) the node lower objective bound is greater than the best known feasible solution, or (iii) the node best possible solution (lower bound) is within the relative or absolute tolerance of the tree global best feasible solution.
- `state`: the state of the tree contains the global best possible objective value and the best feasible objective value, together with the number of nodes visited.

After visiting each node, `BranchAndBoundAlgorithm` updates the tree global best possible objective value and best feasible objective value. The new best feasible objective value is the smallest of the existing best feasible objective values and the node solution feasible solution objective value. The algorithm also uses the `NodeSolution` feasible solution to update the solution pool. Then, the current node is removed from the list of open nodes, and the tree global best possible solution is updated to the best possible solution of all open nodes. An unvisited node best possible solution is its parent's best possible solution. GALINI 1.0.0 outputs the values of all the counters to the rich logging system. Section 6 describes the rich logging system in more detail.

```
class BranchAndBoundAlgorithm:
    @staticmethod
    def extra_config():
```

```
        """Returns algorithm-specific configuration options"""

    def find_initial_solution(self, model, tree, node):
        """Returns a feasible solution to model if any"""

    @property
    def branching_strategy(self):
        """Returns an instance of the algorithm branching strategy"""

    @property
    def node_selection_strategy(self):
        """Returns an instance of the algorithm node selection strategy"""

    def init_node_storage(self, model):
        """Returns the root node storage"""

    def solve_problem_at_node(self, tree, node):
        """Returns a node solution with the best possible objective
        and a feasible solution"""

    def solve_problem_at_root(self, tree, node):
        """Returns a node solution with the best possible objective
        and a feasible solution"""

    def solve(self, model):
        node_storage = sef.init_node_storage(model)
        tree = Tree(node_storage)
        node = tree.root
        self.find_initial_solution(model, tree, node)
        self.solve_problem_at_root(tree, node)
        while not self.has_converged(tree):
            node_children = self.branching_strategy.branch(node)
            for child in node_children:
                self.node_selection_strategy.insert_node(child)
            node = self.node_selection_strategy.next_node()
            self.solve_problem_at_node(tree, node)
```

Listing 1 Definition of the `BranchAndBoundAlgorithm` class with a simplified version of the algorithm. Highlighted in red, the method that solver developers extend to implement their new algorithms. The algorithm starts by initializing the algorithm-specific node storage, then it tries to find an initial feasible solution. The branch & bound algorithm proper starts by solving the root node problem, then it enters the loop. At each iteration of the loop, the algorithm branches on the previous iteration node and adds them to the queue of nodes to visit, then it picks the next node to visit and solves the problem at that node. This loop is repeated until one of the termination conditions is met.

5. Extensible Branch & Cut Algorithm

This section gives an high-level description of the GALINI branch & cut algorithm. The GALINI branch & cut algorithm implements the seven abstract methods from the beginning of Section 4. The branch & cut algorithm itself is extensible. Section 5.1 describes the implementation of GALINI extensibility. The GALINI branch & cut algorithm provides the following extension points:

- **Initial Feasible Solution Search**: this extension point is called at the beginning of the algorithm to find a feasible solution,
- **Cutting Planes**: mix & match different cutting planes classes,
- **Primal Heuristic**: finds a feasible solution. Called immediately after solving the relaxation,
- **Node Selection Strategy**: the branch & bound node selection strategy de-

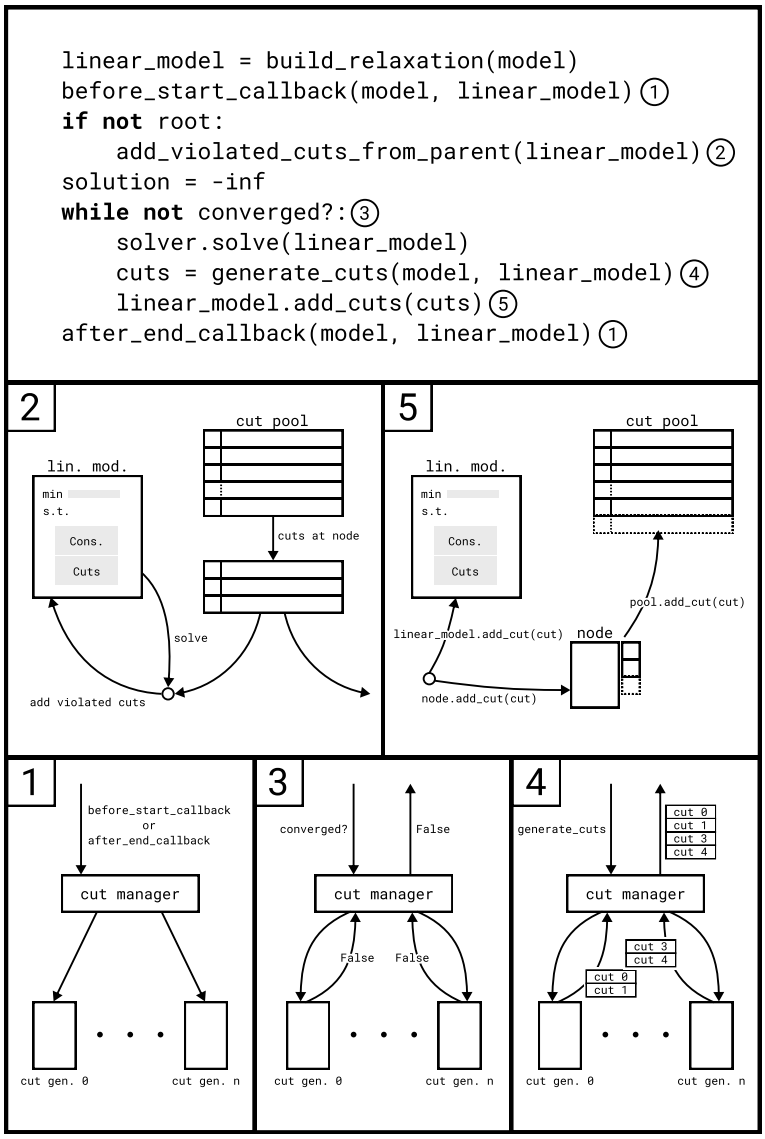

Figure 3. Overview of the GALINI cut loop at each node. 1) The callbacks are called at the beginning and end of each node visit, this callbacks can be used by cuts generators to build and cleanup internal data structures. 2) Check cuts that where inherited to see which ones are violated and add them to the problem. 3) Stop generating cuts when all cuts generators have converged or if the maximum number of iterations is reached. 4) The cut manager call each cut generator to generate new cuts, returning all generated cuts to the algorithm. 5) The returned cuts are added to the cut pool and the linear relaxation of the problem.

scribed in Section 4. This section describes the default implementation,
- **Branching Strategy**: the branch & bound node branching strategy described in Section 4. This section describes the default implementation,
- **Relaxation**: relaxes the original MINLP user model into a MIP model.

Unlike the extension points described in the branch & bound algorithm, these extension points are dynamic and can be extended without changing GALINI itself. Users can decide which code to run by updating the configuration file passed to the GALINI command line tool. Figure 3 shows a simplified version of the cut loop executed at each branch & bound node.

5.1. Extensibility

Extensibility is achieved through Python *entry points*, a mechanism for registering classes to a central registry handled by the user's Python installation. GALINI looks for classes registered at specific entry point keys, e.g. `galini.cuts_generators`, and loads them at runtime. Algorithm developers can develop their extensions in a project separate from the GALINI codebase. Advantages of this model are:

- Simplifies developing a new extension. Developers only need to understand the interfaces provided by GALINI and the GALINI problem representation.
- Extensions can be distributed independently of GALINI.
- Since the interfaces are stable, less work is required to maintain the extensions.

Appendix B describes in detail the interfaces that algorithm developers need to implement to build extensions for the branch & cut algorithm.

5.2. Branch & Cut Algorithm

The algorithm starts by finding an initial feasible solution. To find a feasible solution, GALINI starts by performing FBBT and then solves the NLP restriction obtained by randomly fixing integer variables. GALINI will use any user-provided starting point. If this initial primal heuristic returns a feasible solution, this solution is stored in the branch & bound tree at the root node. If the problem is detected as convex and the problem does not contain any integer variable, then the algorithm has converged and GALINI will return the solution to the user.

After this step, the branch & bound algorithm solves the problem at the root node by calling the `solve_problem_at_root(tree, node)` implemented by the branch & cut algorithm. When solving at the root node, the branch & cut algorithm starts by performing OBBT on the variables that participate in nonlinear terms. If GALINI has a feasible solution, the branch & cut algorithm uses the objective value as an upper bound on the objective in the OBBT problem.

Solving a branch & bound node requires to first update the problem relaxation to use the node's variables bounds and then perform FBBT to tighten the bounds. After this, the branch & cut algorithm relaxes the integrality constraints on integer variables to obtain a Linear Program (LP). GALINI solves the LP using a linear solver (`mip_solver.name`, default Cplex) and then enters a loop to add violated cuts from the node ancestors until there are no more violated cuts. GALINI then enters a cut loop to generate new cuts. The cut loop stops after a maximum number of iterations (`cuts.maxiter`, default 20) or if two consecutive cut rounds have an objective within

a configurable relative tolerance (`cuts.cut_tolerance`, default 10^{-5}). After the cut loop, GALINI reintroduces the integrality constraints on integer variables and solves the MIP. The MIP best possible solution (its lower or dual bound) is used as the node lower bound. Finally, GALINI invokes the primal heuristic to find a feasible solution to the original problem. After solving each node, GALINI checks whether the solution converged or not using the check described in Section 4. After solving the root node, and if the solver has not converged already, GALINI enters the branch & bound loop described in Section 4.

GALINI uses the same convergence tolerance for the global branch & bound algorithm, the LP in the cut loop, the MIP lower bounding problem, and the NLP upper bounding problem.

By default, the branch & cut algorithm branches on the variable with the highest *nonlinear infeasibility* $\Omega_i(\hat{\boldsymbol{x}})$ [5, 16]:

$$\Omega_i(\hat{\boldsymbol{x}}) = \mu_1 \sum_{j \in E(i)} U_j(\hat{\boldsymbol{x}}) + \mu_2 \max_{j \in E(i)} U_j(\hat{\boldsymbol{x}}) + \mu_3 \min_{j \in E(i)} U_j(\hat{\boldsymbol{x}}) \tag{2}$$

where $E(i) = \{x_j \in \boldsymbol{x} : \exists w_{ij} = x_i x_j\}$ and $U_j(\hat{\boldsymbol{x}}) = |\hat{w}_{ij} - \hat{x}_i \hat{x}_j|/(1 + \sqrt{\hat{x}_i^2 + \hat{x}_j^2})$. GALINI uses weights $\mu_1 = 1.0$, $\mu_2 = 0.0$ and $\mu_3 = 0.0$. GALINI branches at a convex combination of the variable midpoint $x_i^m = x_i^L + (x_i^U - x_i^L)/2$ and the solution at the MIP relaxation $\hat{x}_i$ $\lambda x_i^m + (1-\lambda)\hat{x}_i$, with $\lambda = 0.25$. If a variable is unbounded, GALINI branches at the solution point.

The default node selection strategy used by the branch & cut algorithm chooses the node with the lowest lower bound [4]. This is done by keeping the list of nodes to visit in a priority queue that uses the node lower bound for ordering.

5.3. Feasibility-based bounds tightening

GALINI limits the maximum number of FBBT iterations (`fbbt_maxiter`, default = 10) since FBBT converges only in the limit [17]. GALINI also stops FBBT iterations when neither bounds propagation nor tightening produce changes, i.e. when the relative difference (Equation 1) of each lower and upper DAG node bound changes by less than the math context ϵ. As a code optimization, SUSPECT stores the list of DAG vertices with changed bounds and propagate, i.e. tighten from equations to variables, only when the bounds change.

5.4. Relaxations

Because GALINI builds relaxations based on Coramin classes, many model interrogation tasks needed to develop plug-ins are drastically simplified. Coramin relaxes bilinear terms by replacing each bilinear term with an auxiliary variable, i.e. a variable not present in the original problem, and adds the McCormick envelope [52, 68] of the bilinear term to the relaxation.

The relaxation of the user model is stored, together with the original model, in the root node `NodeStorage`. All other nodes store the node variables bounds and provide a method to update the original model bounds and the relaxation. After updating the relaxation variable bounds, the branch & cut algorithm recomputes the McCormick relaxation to use the new bounds.

GALINI also registers a function with Coramin to relax quadratic expressions. When GALINI recognizes that an expression is convex, GALINI dissaggregates the convex quadratic expression and replaces each separable term with an auxiliary variable. GALINI still asks Coramin to relax the individual terms of the quadratic expression with auxiliary variables, but the additional auxiliary variable GALINI introduces can, for instance, develop outer approximation cutting planes. The disaggregation step increases the number of variables but it may also tighten the linear relaxation [59, 87]. GALINI starts by building a graph where nodes represent variables, then if the coefficients q_{ij} of variables x_i, x_j is non zero, it adds an edge between node i and node j. Then, for each connected component (that is a subgraph for which any two nodes are connected to each other by paths and which is not connected to any additional nodes in the subgraph), GALINI checks the convexity of the quadratic expression it represents. If the expression is convex the expression is added to the list of convex expressions GALINI replaces with an auxiliary variable. Otherwise, GALINI adds the expression to the list of expressions it will relax.

Example GALINI dissaggregates the expression $f(x_0, x_1, x_2) = x_0^2 + 2x_0x_1 + x_1^2 + x_2^2$ by replacing the convex expression $x_0^2 + 2x_0x_1 + x_1^2$ with an auxiliary variable w_0 and the other expression x_2^2 with auxiliary variable w_{22}. The expression then becomes $f(w_0, w_{22}) = w_0 + w_{22}$ and GALINI retains the convexity of w_0 for cutting planes. Coramin further relaxes the convex quadratic expression $w_0 = x_0^2 + 2x_0x_1 + x_1^2$ by replacing each bilinear term with an auxiliary variable, obtaining $w_0 = w_{00} + 2w_{01} + w_{11}$. Coramin generates the McCormick envelopes for $w_{00} = x_0^2$, $w_{01} = x_0x_1$, $w_{11} = x_1^2$ and $w_{22} = x_2^2$.

The linear relaxation replaces the objective function $f(\boldsymbol{x})$ with an auxiliary variable w_f and adds the constraint

$$\hat{f}(\boldsymbol{x}) - w_f \leq 0$$

where $\hat{f}(\boldsymbol{x})$ is the underestimator of $f(\boldsymbol{x})$. This is done because some types of cutting planes, for example outer approximation cuts, require adding cuts relative to the objective function.

5.5. Cutting Planes

GALINI is a branch & cut solver [9, 10, 11, 12, 73, 83, 84, 86], so cutting planes play an important role. Several classes and objects manage and create cutting planes.

Section 4 describes node storage, i.e. algorithm-specific storage. At the root node, the branch & cut algorithm uses node storage to add a cut pool to the branch & bound tree. This cut pool stores cuts in a shared cut list and identifies each cut by an increasing index. Local cuts are cuts that are accessible from a node and all its descendants. The other nodes store the indices of the local cuts together with a pointer to the parent local node storage. GALINI 1.0.0 implements all cuts as local cuts. GALINI provides facilities to iterate over all the cuts accessible at a node. Figure 4 shows how the cut pool storage is implemented.

The cut generator interface requires to implement six methods:

- `before_start_at_root(problem, relaxed_problem)`: called before entering the cut loop at the root node,

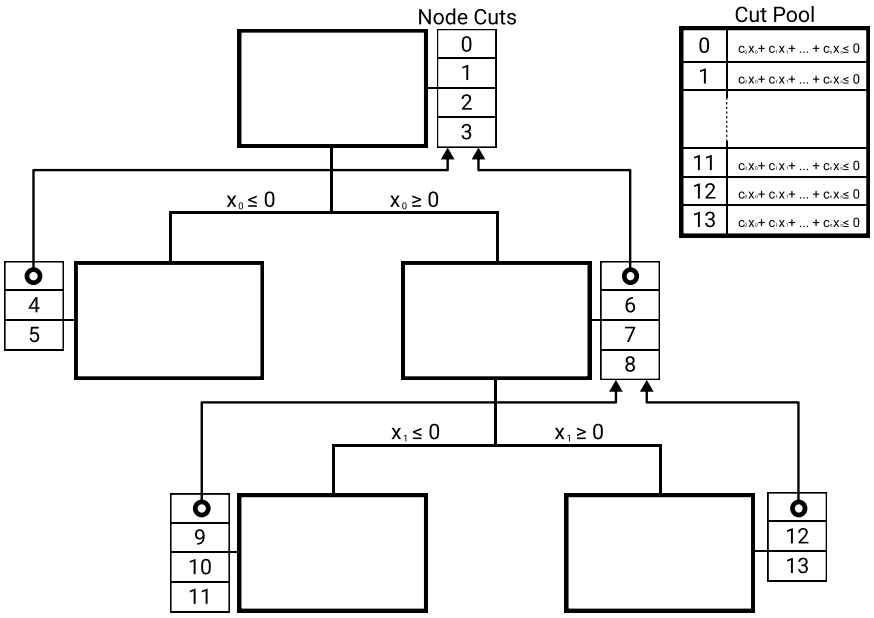

Figure 4. GALINI cuts are stored in a cut pool together with its index. Nodes store a list of the valid cuts indexes and a pointer to their parent cuts.

- `after_end_at_root(problem, relaxed_problem, solution)`: called after solving the MIP at the root node,
- `before_start_at_node(problem, relaxed_problem)`: called before entering the cut loop at non-root nodes,
- `after_end_at_node(problem, relaxed_problem, solution)`: called after solving the MIP at non-root nodes,
- `has_converged(state)`: returns a true value if the cut generator won't generate any more cuts at this node. For example, our outer approximation cuts return false for optimization instances without convex quadratic terms.
- `generate(problem, relaxed_problem, solution, tree, node)`: return a list of cuts to be added to `relaxed_problem`. You have access to the previous iteration `solution`.

The branch & cut algorithm interacts with the cuts generators using a *cut manager*. Users can configure the cut manager to use specific cuts by changing their configuration file, in this way they can experiment by combining different classes of cuts generators [82]. When the cut manager is initialized, it starts by loading the cut generators that the user has enabled in the configuration file. It also initializes each cut generator telemetry data to collect information about the number of cuts generated and the total time spent in the cut generator. The cut manager interface implements the same six methods as a cut generator. The cut manager works by forwarding each method to the cut generators and combining the results.

Cuts don't have to be linear (or convex): they can include quadratic expressions that GALINI will automatically linearize using the relaxation `relax_inequality(model, inequality, relaxation_side, ctx)` method before adding it to the linear problem. This implementation frees the cut developer from keeping track of which auxiliary variables map to which bilinear expression.

GALINI includes the following cutting planes:

- **Outer Approximation**: add cuts based on the first-order Taylor expansion [21, 34, 48]. Since GALINI disaggregates the convex expressions into separable expressions, it may add more than one cut per equation [29].
- **Triangle**: add cuts based on triangle inequalities [14, 20, 80].
- **SDP**: use Neural Networks to select the best cutting planes [8, 13, 81, 85].

There are parameters in each of the cutting planes classes. For example, GALINI generates an outer approximation cut only if the difference between the value of the convex nonlinear expression, obtained by evaluating the expression at the LP solution point, and its linearized expression is above a above a user-configurable threshold (`deviation_threshold`, default = 10^{-5}).

GALINI could be extended to include specialized cuts for nonconvex quadratic problems with separable constraints [32] or for pooling problems [27, 63].

6. Rich Logging & Counters

GALINI is a solver that can be easily extended, for this reason we include several functions that make debugging every aspect of the solver easier.

GALINI users can enable *Paranoid Mode*, when this flag is active the solver will perform extra checks to verify the correctness of every step. If any check fails, GALINI stops the execution to drop into a Python console inside the function that failed. Developers can use the console to inspect the state of their function to find the cause of the bug and even change the state of the running program to quickly test a fix. By default this flag is turned off since it can slow down the solving process considerably.

GALINI provides logging functions that can be used to not only log string content to the screen, but also log richer data structures. Example data that can be logged by GALINI:

- **Tensors**: tensors (multi-dimensional vectors and matrices) are written to an HDF5 file so that they can be stored and viewed at a later moment. This is extremely useful when debugging numerical issues,
- **Branch & Bound Operations**: the branch & bound algorithm logs every operation on the branch & bound tree. The algorithm logs when a new node is visited or fathomed,
- **Variables**: variables that change over time (for example the branch & bound tree best possible and best feasible objectives) can be logged. These variables can then be plotted to see how their value changes in relation to time or number of branch & bound nodes visited,
- **Time Spans**: sections of code can be annotated using the `telemetry(name)` context manager to log the cumulative amount of time spent in the section of code.

With this extra data, we built GALINI dashboard, which is similar to Tensorboard for Tensorflow [1] and provides insight into the solver. The GALINI dashboard[3] provides an interface to visualize: 1) the evolution of the branch & bound best possible and best feasible objectives, 2) the branch & bound tree, and 3) a timeline of where the solver spends its time. Figure 5 shows a screen of the GALINI dashboard.

GALINI developers have also access to *counters* and *gauges*. Counters are fields that represent an increasing sequence, for example the number of cuts generated or

[3] `https://github.com/cog-imperial/galini-dashboard`

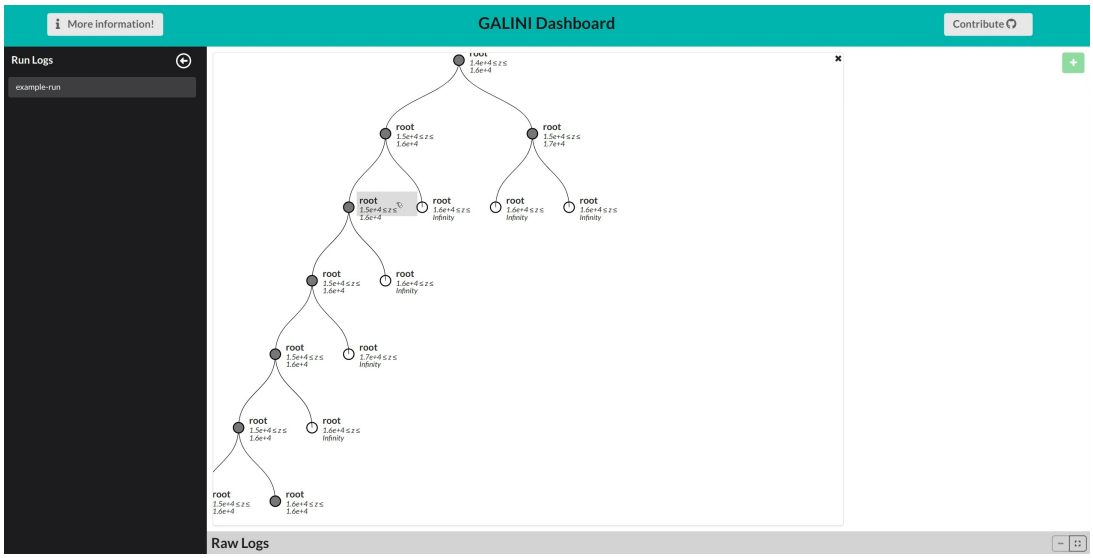

Figure 5. GALINI dashboard screen showing the interactive branch & bound tree. A short video demonstration of the dashboard is available at `https://youtu.be/3uWw_EcagNE`.

the number of nodes visited. Gauges are fields whose value can both increase and decrease, for example the number of open nodes. Developers can create counters and gauges, the branch & bound algorithm automatically logs these values at the end of each node visit.

7. Computational Results

We test GALINI on a Linux workstation with an Intel i7-6700 3.4 GHz CPU and 16 GB of RAM. We use Python 3.8.5, Cplex 12.8.0.0, Ipop 3.13.2 compiled against HSL [49] 2014.01.10.

We use GALINI version 1.1.0, Gurobi beta version 9.1 (a prerelease version downloaded on 2020-10-08), and the version of ANTIGONE, Baron, Couenne and SCIP included in GAMS 31.1.

We run the computational experiments on 761 MIQCQP problems from MINLPLib 2 [24, 88] (accessed October 2020) and 453 problems from QPLIB [41]. We augment the MINLPLib 2 OSiL [39] with the same starting point as the GAMS models.

The optimality gap $\text{gap} = \frac{z^U - z^L}{|z^U|}$ is set to 10^{-6} and the timelimit is 300 seconds. Problems where the solver overshot the timelimit by over 5% (15 seconds) are not considered solved.

In the first set of results, we run GALINI with all different combinations of cuts: triangle, outer approximation, and sdp cuts individually, then by combining all pairs of cuts, and finally with all three classes of cuts activated. Figure 8 shows the performance profile of different GALINI cuts configurations on the MINLPLib 2 dataset. We can see that activating all cuts generators is the best choice for perforamnce since it has no performance impact.

In the second set of results, we compare GALINI with both commercial and open source solvers. Figure 9 contains the performance profile, we can see that GALINI spends more time solving problems but the optimality gap after 5 minutes is comparable to other, more mature solvers. This is the result we expected since GALINI is

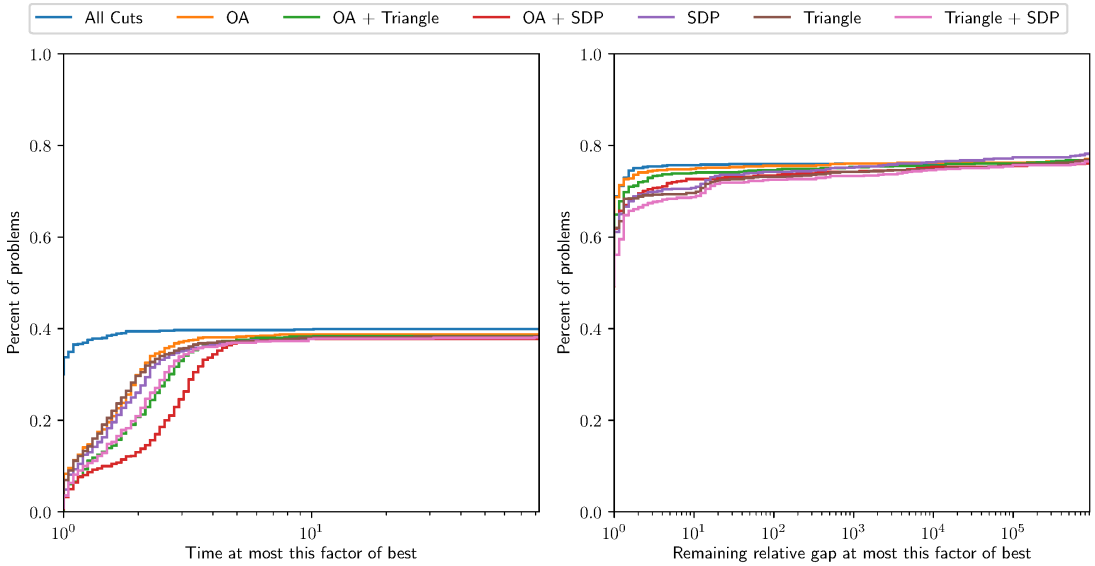

Figure 6. Performance profile of different GALINI cuts combinations on 761 instances form MINLPLib2. The figure on the left compares CPU time, the figure on the right the remaining gap after 300 seconds.

built to be easy to extend and does not focus on performance.

8. Conclusion

This manuscript introduces GALINI, an extensible MIQCQP solver written in Python. We show how GALINI can be used by solver developers to build new solvers and how algorithm developers can extend GALINI at runtime to change parts of the branch & cut algorithm. Our computational results show that the branch & cut algorithm performance is comparable to other open source solvers and for this reason we think GALINI is a great starting point for algorithm developers. GALINI could be extended to include specialized cuts for nonconvex quadratic problems with separable constraints [32] or for pooling problems [27, 63].

9. Acknowledgements

This work was funded by an Engineering & Physical Sciences Research Council Research Fellowship to RM [GrantNumber EP/P016871/1].

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DENA0003525.

Disclaimer: This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

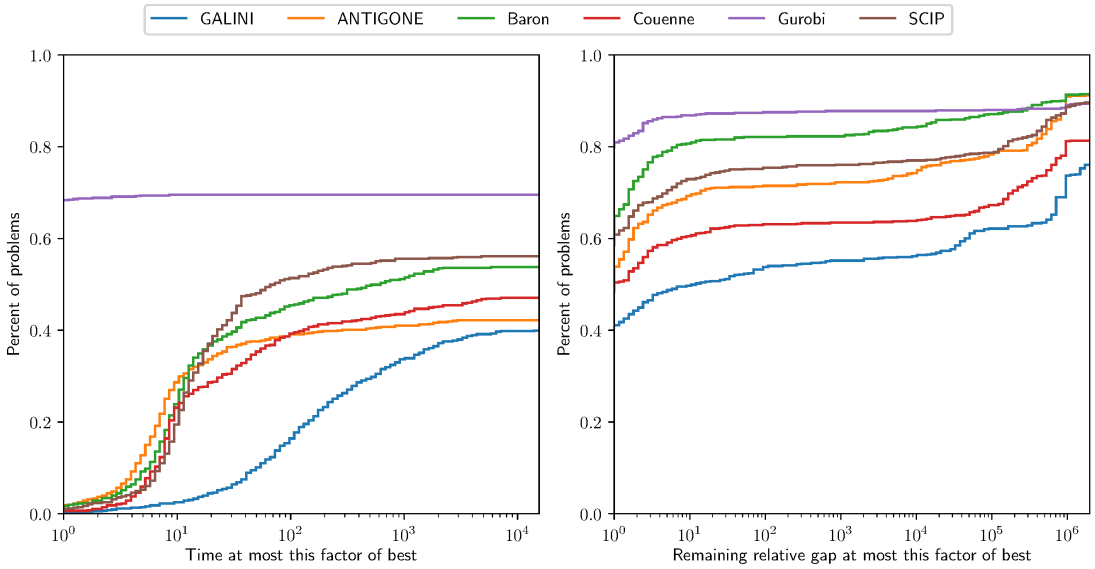

Figure 7. Performance profile comparing different solvers on 761 instances form MINLPLib2. The figure on the left compares CPU time, the figure on the right the remaining gap after 300 seconds.

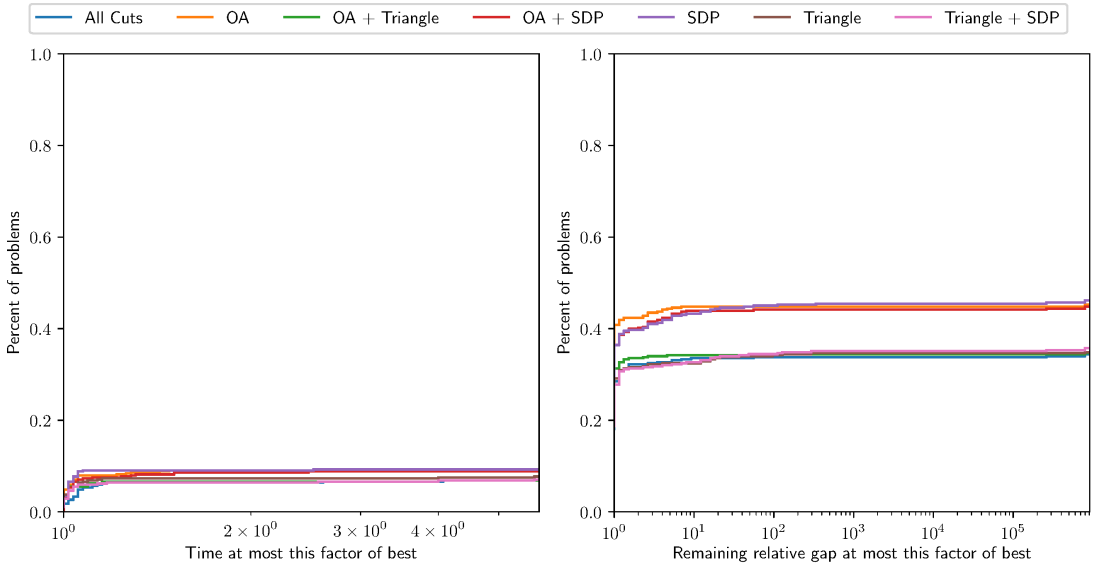

Figure 8. Performance profile of different GALINI cuts combinations on 453 instances form QPLIB. The figure on the left compares CPU time, the figure on the right the remaining gap after 300 seconds.

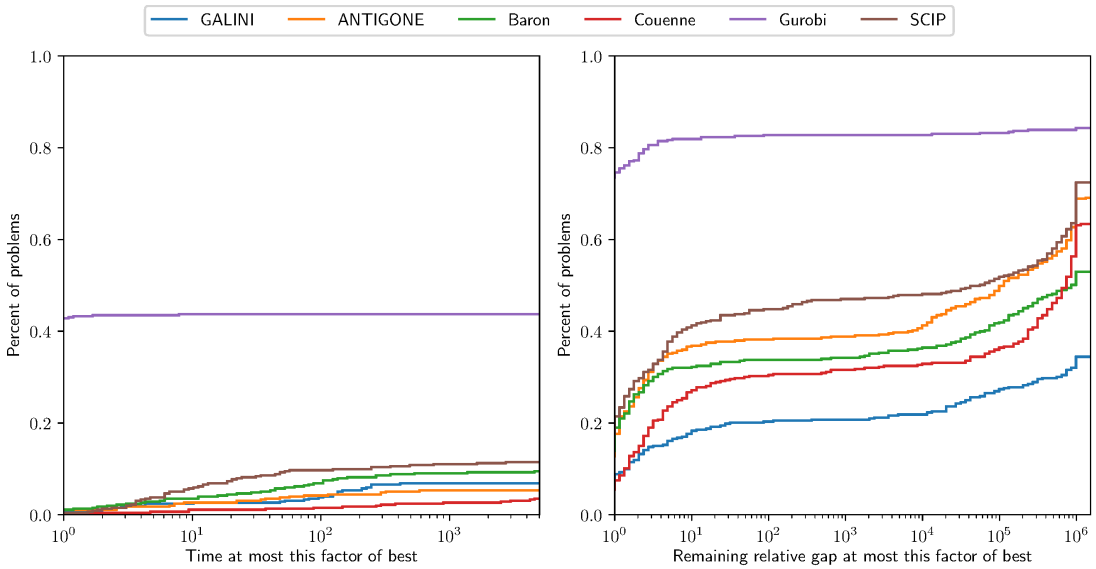

Figure 9. Performance profile comparing different solvers on 453 instances form QPLIB. The figure on the left compares CPU time, the figure on the right the remaining gap after 300 seconds.

References

[1] M. Abadi et al. "Tensorflow: A system for large-scale machine learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283.

[2] K. Abhishek, S. Leyffer, and J. Linderoth. "FilMINT: An Outer Approximation-Based Solver for Convex Mixed-Integer Nonlinear Programs". In: *INFORMS J Comput* 22.4 (2010), pp. 555–567.

[3] T. Achterberg. "SCIP: solving constraint integer programs". In: *Mathematical Programming Computation* 1.1 (2009), pp. 1–41.

[4] C. S. Adjiman, I. P. Androulakis, and C. A. Floudas. "Global optimization of mixed-integer nonlinear problems". In: *AIChE Journal* 46.9 (2000), pp. 1769–1797.

[5] C. S. Adjiman, I.P. Androulakis, and C.A. Floudas. "A global optimization method, αBB, for general twice-differentiable constrained NLPs-II. Implementation and computational results". In: *Computers & Chemical Engineering* 22.9 (1998), pp. 1159–1179.

[6] M. Alfaki and D. Haugland. "A multi-commodity flow formulation for the generalized pooling problem". In: *Journal of Global Optimization*. Vol. 56. 3. 2013, pp. 917–937.

[7] M. Alfaki and D. Haugland. "Strong formulations for the pooling problem". In: *Journal of Global Optimization* 56.3 (2013), pp. 897–916.

[8] K. M. Anstreicher. "Semidefinite programming versus the reformulation-linearization technique for nonconvex quadratically constrained quadratic programming". In: *Journal of Global Optimization* 43.2-3 (2009), pp. 471–484.

[9] C. Audet, P. Hansen, and F. Messine. "The small octagon with longest perimeter". In: *Journal of Combinatorial Theory. Series A* 114.1 (2007), pp. 135–150.

[10] C. Audet et al. “A branch and cut algorithm for nonconvex quadratically constrained quadratic programming”. In: *Mathematical Programming, Series B* 87.1 (2000), pp. 131–152.
[11] C. Audet et al. “Pooling problem: Alternate formulations and solution methods”. In: *Management Science* 50.6 (2004), pp. 761–776.
[12] C. Audet et al. “The largest small octagon”. In: *Journal of Combinatorial Theory. Series A* 98.1 (2002), pp. 46–59.
[13] R. Baltean-Lugojan et al. “Selecting cutting planes for quadratic semidefinite outer-approximation via trained neural networks”. In: (2018). URL: http://www.optimization-online.org/DB%7B%5C_%7DFILE/2018/11/6943.pdf.
[14] X. Bao, N. V. Sahinidis, and M. Tawarmalani. “Multiterm polyhedral relaxations for nonconvex, quadratically constrained quadratic programs”. In: *Optimization Methods and Software* 24.4-5 (2009), pp. 485–504.
[15] N. Beaumont. “An algorithm for disjunctive programs”. In: *European Journal of Operational Research* 48.3 (1990), pp. 362–371.
[16] P. Belotti et al. “Branching and bounds tightening techniques for non-convex MINLP”. In: *Optimization Methods and Software* 24.4-5 (2009), pp. 597–634.
[17] P. Belotti et al. “Feasibility-Based Bounds Tightening via Fixed Points”. In: *Combinatorial Optimization and Applications*. Springer, Berlin, Heidelberg, 2010, pp. 65–76.
[18] P. Belotti et al. “Mixed-integer nonlinear optimization”. In: *Acta Numerica* 22 (2013), pp. 1–131.
[19] H. Y. Benson. “Mixed integer nonlinear programming using interior-point methods”. In: *Optim Method Softw* 26.6 (2011), pp. 911–931.
[20] P. Bonami, O. Günlük, and J. Linderoth. “Globally solving nonconvex quadratic programming problems with box constraints via integer programming methods”. In: *Mathematical Programming Computation* 10.3 (2018), pp. 333–382.
[21] P. Bonami et al. “An algorithmic framework for convex mixed integer nonlinear programs”. In: *Discrete Optimization* 5.2 (2008), pp. 186–204.
[22] D. Bongartz et al. *MAiNGO: McCormick based Algorithm for mixed integer Nonlinear Global Optimization*. Tech. rep. Technical report, 2018.
[23] F. Boukouvala, R. Misener, and C. A. Floudas. “Global optimization advances in Mixed-Integer Nonlinear Programming, MINLP, and Constrained Derivative-Free Optimization, CDFO”. In: *European Journal of Operational Research* 252.3 (2016), pp. 701–727.
[24] M. R. Bussieck, A. S. Drud, and A. Meeraus. “MINLPLib - A collection of test models for mixed-integer nonlinear programming”. In: *INFORMS Journal on Computing* 15.1 (2003), pp. 114–119.
[25] R. H. Byrd, J. Nocedal, and R. A. Waltz. “Knitro: An Integrated Package for Nonlinear Optimization”. In: *Large-Scale Nonlinear Optimization*. Ed. by G. Di Pillo and M. Roma. Boston, MA: Springer US, 2006, pp. 35–59.
[26] P. M. Castro and I. E. Grossmann. “Global Optimal Scheduling of Crude Oil Blending Operations with RTN Continuous-time and Multiparametric Disaggregation”. In: *Industrial & Engineering Chemistry Research* 53.39 (2014), pp. 15127–15145.
[27] F. Ceccon, G. Kouyialis, and R. Misener. “Using functional programming to recognize named structure in an optimization problem: Application to pooling”. In: *AIChE Journal* 62.9 (2016), pp. 3085–3095.
[28] F. Ceccon, J. D. Siirola, and R. Misener. “SUSPECT: MINLP special structure detector for Pyomo”. In: *Optimization Letters* (2019), pp. 1–14.

[29] C. Coey, M. Lubin, and J. P. Vielma. "Outer approximation with conic certificates for mixed-integer convex problems". In: *Mathematical Programming Computation* (2020), pp. 1–45.

[30] *Coramin: A collection of tools (classes, functions, etc.) for developing MINLP algorithms*. `https://github.com/Coramin/Coramin`. Accessed: 2020-08-21.

[31] F. Domes and A. Neumaier. "Constraint propagation on quadratic constraints". In: *Constraints* 15.3 (2010), pp. 404–429.

[32] H. Dong. "Relaxing nonconvex quadratic functions by multiple adaptive diagonal perturbations". In: *SIAM Journal on Optimization* 26.3 (2016), pp. 1962–1985.

[33] H. Dong and N. Krislock. "Semidefinite approaches for MIQCP: Convex relaxations and practical methods". In: *Springer Proceedings in Mathematics and Statistics*. Vol. 147. Springer New York LLC, 2015, pp. 49–75.

[34] M. A. Duran and I. E. Grossmann. "An outer-approximation algorithm for a class of mixed-integer nonlinear programs". In: *Mathematical Programming* 36.3 (1986), pp. 307–339.

[35] D. C. Faria and M. J. Bagajewicz. "A new approach for global optimization of a class of MINLP problems with applications to water management and pooling problems". In: *AIChE Journal* 58.8 (2012), pp. 2320–2335.

[36] D. C. Faria and M. J. Bagajewicz. "Novel bound contraction procedure for global optimization of bilinear MINLP problems with applications to water management problems". In: *Computers & Chemical Engineering* 35.3 (2011), pp. 446–455.

[37] R. Fletcher and S. Leyffer. "Numerical experience with lower bounds for MIQP branch-and-bound". In: *SIAM J Optim* 8.2 (1998), pp. 604–616.

[38] C. A. Floudas and I. E. Grossmann. "Synthesis of flexible heat exchanger networks with uncertain flowrates and temperatures". In: *Computers & Chemical Engineering* 11.4 (1987), pp. 319–336.

[39] R. Fourer, J. Ma, and K. Martin. "OSiL: An instance language for optimization". In: *Computational optimization and applications* 45.1 (2010), pp. 181–203.

[40] R. Fourer and D. Orban. "DrAmpl: A meta solver for optimization problem analysis". In: *Computational Management Science* 7.4 (2010), pp. 437–463.

[41] F. Furini et al. "QPLIB: a library of quadratic programming instances". In: *Mathematical Programming Computation* 11.2 (2019), pp. 237–265.

[42] B. Galan and I. E. Grossmann. "Optimal design of distributed wastewater treatment networks". In: *Industrial & Engineering Chemistry Research* 37.10 (1998), pp. 4036–4048.

[43] C.-Y. Gau and L. E. Schrage. "Implementation and Testing of a Branch-and-Bound Based Method for Deterministic Global Optimization: Operations Research Applications". In: Springer, Boston, MA, 2004, pp. 145–164.

[44] I. E. Grossmann. *Global Optimization in Engineering Design*. Springer US, 1996, p. 387. ISBN: 9781475753318.

[45] I. E. Grossmann and Z. Kravanja. "Mixed-Integer Nonlinear Programming: A Survey of Algorithms and Applications". In: *Large-scale optimization with applications*. Springer, New York, NY, 1997, pp. 73–100.

[46] W. E. Hart, J.-P. Watson, and D. L. Woodruff. "Pyomo : modeling and solving mathematical programs in Python". In: *Mathematical Programming Computation* 3.3 (2011), pp. 219–260.

[47] W. E. Hart et al. *Pyomo–optimization modeling in python*. Second Edi. Springer Science & Business Media, 2017.

[48] H. Hijazi, P. Bonami, and A. Ouorou. "An outer-inner approximation for separable mixed-integer nonlinear programs". In: *INFORMS Journal on Computing* 26.1 (2014), pp. 31–44.

[49] HSL HSL. *A collection of Fortran codes for large scale scientific computation, 2002*. 2002.

[50] M. Hunting. "The AIMMS outer approximation algorithm for MINLP". In: *Paragon Decision Technology, Haarlem* (2011).

[51] R. Karuppiah and I. E. Grossmann. "Global optimization for the synthesis of integrated water systems in chemical processes". In: *Computers & Chemical Engineering* 30.4 (2006), pp. 650–673.

[52] F. A. Al-Khayyal and J. E. Falk. "Jointly Constrained Biconvex Programming". In: 8.2 (1983), pp. 273–286.

[53] O. Kröger et al. "Juniper: An Open-Source Nonlinear Branch-and-Bound Solver in Julia". In: *CPAIOR*. Ed. by W.-J. van Hoeve. Springer International Publishing, 2018, pp. 377–386.

[54] S. Le Digabel. "Algorithm 909: NOMAD: Nonlinear optimization with the MADS algorithm". In: *ACM Transactions on Mathematical Software (TOMS)* 37.4 (2011), p. 44.

[55] H. Lee et al. "Mixed-integer linear programming model for refinery short-term scheduling of crude oil unloading with inventory management". In: *Industrial & Engineering Chemistry Research* 35.5 (1996), pp. 1630–1641.

[56] S. Lee and I. E. Grossmann. "New algorithms for nonlinear generalized disjunctive programming". In: *Computers & Chemical Engineering* 24.9-10 (2000), pp. 2125–2141.

[57] J. Li, R. Misener, and C. A. Floudas. "Scheduling of crude oil operations under demand uncertainty: A robust optimization framework coupled with global optimization". In: *AIChE Journal* 58.8 (2012), pp. 2373–2396.

[58] J. Li et al. "Improving the robustness and efficiency of crude scheduling algorithms". In: *AIChE Journal* 53.10 (2007), pp. 2659–2680.

[59] L. Liberti and C. C. Pantelides. "An Exact Reformulation Algorithm for Large Nonconvex NLPs Involving Bilinear Terms". In: *Journal of Global Optimization* 36.2 (2006), pp. 161–189.

[60] Y. Lin and L. Schrage. "The global solver in the LINDO API". In: *Optimization Methods and Software* 24.4-5 (2009), pp. 657–668.

[61] J. Liu et al. "A multitree approach for global solution of ACOPF problems using piecewise outer approximations". In: *Computers & Chemical Engineering* 114 (2018), pp. 145–157.

[62] R. Lougee-Heimer. "The Common Optimization INterface for Operations Research: Promoting open-source software in the operations research community". In: *IBM Journal of Research and Development* 47.1 (2003), pp. 57–66.

[63] J. Luedtke et al. "Strong Convex Nonlinear Relaxations of the Pooling Problem". In: *SIAM Journal on Optimization* 30.2 (2020), pp. 1582–1609. eprint: 1803.02955.

[64] A. Lundell, J. Kronqvist, and T. Westerlund. "The Supporting Hyperplane Optimization Toolkit". In: *URL http://www.github.com/coin-or/shot* (2018).

[65] A. Mahajan et al. "Minotaur: A mixed-integer nonlinear optimization toolkit". In: *Optimization Online* 6275 (2017).

[66] E. Martelli et al. "MINLP model and two-stage algorithm for the simultaneous synthesis of heat exchanger networks, utility systems and heat recovery cycles". In: *Computers & Chemical Engineering* 106 (2017), pp. 663–689.

[67] E. Martelli et al. “Synthesis of Heat Exchanger Networks and Utility Systems: sequential initialization procedure and simultaneous MINLP algorithm”. In: *Computer Aided Chemical Engineering*. Vol. 38. Elsevier B.V., 2016, pp. 1449–1454.

[68] G. P. McCormick. “Computability of global solutions to factorable nonconvex programs: Part I - Convex underestimating problems”. In: *Mathematical Programming* 10.1 (1976), pp. 147–175.

[69] W. Melo, M. Fampa, and F. Raupp. “An overview of MINLP algorithms and their implementation in Muriqui Optimizer”. In: *Annals of Operations Research* (2018).

[70] R. Misener and C. A. Floudas. “Advances for the pooling problem: Modeling, global optimization, and computational studies”. In: *Applied and Computational Mathematics* 8.1 (2009), pp. 3–2.

[71] R. Misener and C. A. Floudas. “ANTIGONE: Algorithms for coNTinuous / Integer Global Optimization of Nonlinear Equations”. In: *Journal of Global Optimization* 59.2-3 (2014), pp. 503–526.

[72] R. Misener and C. A. Floudas. “Global Optimization of Large-Scale Generalized Pooling Problems: Quadratically Constrained MINLP Models”. In: *Industrial & Engineering Chemistry Research* 49.11 (2010), pp. 5424–5438.

[73] R. Misener and C. A. Floudas. “GloMIQO: Global mixed-integer quadratic optimizer”. In: *Journal of Global Optimization* 57.1 (2013), pp. 3–50.

[74] R. Misener, J. B. Smadbeck, and C. A. Floudas. “Dynamically generated cutting planes for mixed-integer quadratically constrained quadratic programs and their incorporation into GloMIQO 2”. In: *Optimization Methods and Software* 30.1 (2015), pp. 215–249.

[75] H. Nagarajan et al. “An adaptive, multivariate partitioning algorithm for global optimization of nonconvex programs”. In: *Journal of Global Optimization* (2019). ISSN: 1573-2916.

[76] H. Nagarajan et al. “Tightening McCormick relaxations for nonlinear programs via dynamic multivariate partitioning”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2016, pp. 369–387.

[77] J. Najman and A. Mitsos. “Tighter McCormick relaxations through subgradient propagation”. In: *Journal of Global Optimization* 75.3 (2019), pp. 565–593.

[78] Zorka Novak P. and Z. Kravanja. “A methodology for the synthesis of heat exchanger networks having large numbers of uncertain parameters”. In: *Energy* 92 (2015), pp. 373–382.

[79] I. Nowak, H. Alperin, and S. Vigerske. “LaGO – An Object Oriented Library for Solving MINLPs”. In: *Global Optimization and Constraint Satisfaction*. Ed. by C. Bliek, C. Jermann, and A. Neumaier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 32–42.

[80] M. Padberg. “The Boolean quadric polytope: Some characteristics, facets and relatives”. In: *Mathematical Programming* 45.1-3 (1989), pp. 139–172.

[81] A. Qualizza, P. Belotti, and F. Margot. “Linear Programming Relaxations of Quadratically Constrained Quadratic Programs”. In: *Mixed Integer Nonlinear Programming*. 2012, pp. 407–426.

[82] F. Rendl, G. Rinaldi, and A. Wiegele. “Solving Max-Cut to optimality by intersecting semidefinite and polyhedral relaxations”. In: *Mathematical Programming* 121.2 (2010), pp. 307–335.

[83] A. Saxena, P. Bonami, and J. Lee. “Convex relaxations of non-convex mixed integer quadratically constrained programs: Extended formulations”. In: *Mathematical Programming* 124.1-2 (July 2010), pp. 383–411.

[84] A. Saxena, P. Bonami, and J. Lee. "Convex relaxations of non-convex mixed integer quadratically constrained programs: Projected formulations". In: *Mathematical Programming* 130.2 (2011), pp. 359–413.

[85] H. D. Sherali, E. Dalkiran, and J. Desai. "Enhancing RLT-based relaxations for polynomial programming problems via a new class of v-semidefinite cuts". In: *Computational Optimization and Applications* 52.2 (2012), pp. 483–506.

[86] M. Tawarmalani and N. V. Sahinidis. "A polyhedral branch-and-cut approach to global optimization". In: *Mathematical Programming* 103.2 (2005), pp. 225–249.

[87] M. Tawarmalani and N. V. Sahinidis. *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming*. Vol. 65. Nonconvex Optimization and Its Applications. Boston, M.: Springer US, 2002.

[88] S. Vigerske. "MINLPLib 2". In: *Proceedings of the XII global optimization workshop MAGO*. Vol. 2014. 2014, pp. 137–140.

[89] S. Vigerske and A. Gleixner. "SCIP: global optimization of mixed-integer nonlinear programs in a branch-and-cut framework". In: *Optimization Methods and Software* 33.3 (2018), pp. 563–593.

[90] S. Vigerske et al. "Analyzing the computational impact of MIQCP solver components". In: *Numerical Algebra, Control and Optimization* 2.4 (2012), pp. 739–748.

[91] Stefan Vigerske. "Decomposition in Multistage Stochastic Programming and a Constraint Integer Programming Approach to Mixed-Integer Nonlinear Programming". PhD thesis. 2013. URL: `https://www.math.hu-berlin.de/%7B~%7Dstefan/diss.pdf`.

[92] J. Viswanathan and I.E. Grossmann. "A combined penalty function and outer-approximation method for MINLP optimization". In: *Computers and Chemical Engineering* 14.7 (1990), pp. 769–782.

[93] A. Wächter and L. T. Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". In: *Mathematical Programming* 106.1 (2006), pp. 25–57.

[94] T. Westerlund and R. Pörn. "Solving pseudo-convex mixed integer optimization problems by cutting plane techniques". In: *Optim Eng* 3.3 (2002), pp. 253–280.

[95] M. Wilhelm and M. D. Stuber. "Easy Advanced Global Optimization (EAGO): An Open-Source Platform for Robust and Global Optimization in Julia". In: *AIChE Annual Meeting*. 2017.

Appendix A. Coramin: Convex Relaxation Example

The following example demonstrates how Coramin can build and refine a convex relaxation of

$$\min \; x^2 + y^2 \tag{A1a}$$
$$\text{s.t.} \; y = (x-1)^2 \tag{A1b}$$
$$y \geq \exp(x) \tag{A1c}$$

```
import pyomo.environ as pe
import coramin

m = pe.ConcreteModel()
m.x = pe.Var(bounds=(-2,2))
m.y = pe.Var()
m.x_minus_1 = pe.Var(bounds=(-3, 1))
m.objective = pe.Objective(expr=m.x**2 + m.y**2)
m.x_minus_1_con = pe.Constraint(expr=m.x_minus_1 == m.x - 1)

# build the relaxation for y = x_minus_1**2
m.c1 = coramin.relaxations.PWXSquaredRelaxation()
m.c1.build(x=m.x_minus_1, aux_var=m.y)

# build the relaxation for y >= exp(x)
m.c2 = coramin.relaxations.PWUnivariateRelaxation()
m.c2.build(x=m.x,
           aux_var=m.y,
           shape=coramin.utils.FunctionShape.CONVEX,
           f_x_expr=pe.exp(m.x),
           relaxation_side=coramin.utils.RelaxationSide.UNDER)

# solve the relaxation
opt = pe.SolverFactory('gurobi_persistent')
opt.set_instance(m)
res = opt.solve()
pe.assert_optimal_termination(res)
print(m.y.value)  # 0.3987

# iterate over relaxations and
#   1. tell the relaxation to update the solver with any changes
#   2. add outer approximation cuts
for relaxation in coramin.relaxations.relaxation_data_objects(m):
    relaxation.add_persistent_solver(opt)
    relaxation.add_cut()
opt.solve()
print(m.y.value)  # 0.9981

# rebuild relaxation with updated bounds on x
m.x.setub(0.01)
for relaxation in coramin.relaxations.relaxation_data_objects(m):
    if m.x in relaxation.get_rhs_vars():
        relaxation.rebuild()
opt.solve()
print(m.y.value)  # 0.9990
```

GALINI is a quadratic optimization solver, so one of the most important Coramin relaxations is the McCormick convex hull. The McCormick relaxation replaces bilinear terms with auxiliary variables and adds the relevant McCormick envelopes to the problem. Coramin replaces the bilinear term $x_i x_j$ with auxiliary variable w_{ij} and the following constraints:

$$w_{ij} \geq x_i^L x_j + x_i x_j^L - x_i^L x_j^L \tag{A2}$$
$$w_{ij} \geq x_i^U x_j + x_i x_j^U - x_i^U x_j^U \tag{A3}$$
$$w_{ij} \leq x_i^U x_j + x_i x_j^L - x_i^U x_j^L \tag{A4}$$
$$w_{ij} \leq x_i^L x_j + x_i x_j^U - x_i^L x_j^U \tag{A5}$$

where $x_i \in [x_i^L, x_i^U]$ and $x_j \in [x_j^L, x_j^U]$. Coramin handles the case of a variable not having a lower or upper bound by not generating the constraints that would always be satisfied. The auxiliary variable bounds are computed using interval arithmetic, Coramin handles the case in which x_i and x_j coincide ($w_{ii} = x_i^2$) and use the correct bounds on w_{ii} and only add Equations (A2) to (A4). The auxiliary variable will also contain a reference to the bilinear term it's going to replace. When an auxiliary variable is created, it is also added to a special hash map in the context object that maps each bilinear term (as index of variables, with the smallest index first) to the auxiliary variable. This is important to not duplicate auxiliary variables. This map of auxiliary variables is shared between the different underestimators.

Appendix B. Branch & Cut Extensions Interfaces

The `InitialPrimalSearchStrategy` requires algorithm developers to implement the following method:

(1) `solve(model, tree, node)`: find a feasible solution for the model if any, or `None` if not possible.

The `PrimalHeuristic` interface:

(1) `solve(model, linear_model, mip_solution, tree, node)`: find a feasible solution for the model if any, or `None` if not possible. Unlike the method in `InitialPrimalSearchStrategy`, this method has access to the linear relaxation of the model and its solution from the cut loop.

The `Relaxation` interface:

(1) `relax(model, ctx)`: this method is called once at the root node to create the linear relaxation of the model, the parameter `ctx` is an opaque data structure that Coramin uses to keep track of relaxations data. This data is used, for example, to always replace the same bilinear terms with the same auxiliary variables,
(2) `relax_inequality(model, inequality, relaxation_side, ctx)`: this method should relax the nonlinear inequality into a linear one. This method is called by the algorithm when a cut generator returns a nonlinear cut.