SANDIA REPORT

SAND2021-6270
Printed May 2021

Sandia National Laboratories

PAO 1.0: A Python Library for Adversarial Optimization

William E. Hart, Anya Castillo, Emma S. Johnson, She'ifa Punla-Green

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

PAO 1.0: A Python Library for Adversarial Optimization

William E. Hart
Sandia National Laboratories
PO Box 5800, MS 1318
Albuquerque, NM 87185
wehart@sandia.gov

Anya Castillo
8767 Springwood Dr
Woodbury, MN 55125
anya.castillo@gmail.com

Emma S. Johnson
H. Milton Stewart School of Industrial and Systems Engineering
Georgia Institute of Technology
755 Ferst Drive NW
Atlanta, GA 30332
ejohnson335@gatech.edu

She'ifa Punla-Green
Department of Mathematical Sciences
Rensselaer Polytechnic Institute
110 8th St, Amos Eaton Hall
Troy, NY 12180
sheifazera@gmail.com

SAND2021-6270

ABSTRACT

PAO is a Python-based package for Adversarial Optimization. The goal of this package is to provide a general modeling and analysis capability for bilevel, trilevel and other multilevel optimization forms that express adversarial dynamics. PAO integrates two different modeling abstractions:

1. **Algebraic models** extend the modeling concepts in the Pyomo algebraic modeling language to express problems with an intuitive algebraic syntax. Thus, we expect that this modeling abstraction will commonly be used by PAO end-users.
2. **Compact models** express objective and constraints in a manner that is typically used to express the mathematical form of these problems (e.g. using vector and matrix data types). PAO defines custom *Multilevel Problem Representations* (MPRs) that simplify the implementation of solvers for bilevel, trilevel and other multilevel optimization problems.

Acknowledgment

We thank Richard Chen, John Siirola and Jean-Paul Watson for collaborating on the development of **pyomo.bilevel**. The PAO software is derived from ideas we prototyped in **pyomo.bilevel**. We also thank members of the SECURE Grand Challenge LDRD who provided input on design and application of PAO, especially Bryan Arguello, Jared Gearhart, Ali Pinar, and Cindy Phillips .

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 DE-NA0003525. 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.

1 Installation

PAO currently supports the following versions of Python:

- CPython: 3.7, 3.8, 3.9

1.1 Using PIP

The standard utility for installing Python packages is **pip**. You can install the latest release of PAO by executing the following:

```
pip install pao
```

You can also use **pip** to install from the PAO software repository. For example, the master branch can be installed as follows:

```
python -m pip install https://github.com/or-fusion/pao.git
```

1.2 Using CONDA

The **conda** utility can be used to install the latest release of PAO using the `conda-forge` channel:

```
conda install -c conda-forge pao
```

1.3 Using GIT

PAO can be installed by cloning the PAO software repostory and then directly installing the software. For example, the master branch can be installed as follows:

```
git clone https://github.com/or-fusion/pao.git
cd pao
python setup.py develop
```

1.4 Conditional Dependencies

Both **conda** and **pip** can be used to install the third-party packages that are needed to model problems with PAO. We recommend **conda** because it has better support for optimization solver packages.

PAO intrinsically depends on Pyomo, both for the representation of algebraic problems but also for interfaces to numerical optimizers used by PAO solvers. Pyomo is installed with PAO, but the Pyomo website [6] and GitHub site [7] provide additional resources for installing Pyomo and related software.

PAO and Pyomo have conditional dependencies on a variety of third-party packages, including Python packages like scipy, numpy and optimization solvers. Optimization solvers are particularly important, and a commercial optimizer may be needed to analyze complex, real-world applications.

The following optimizers are used to test the PAO solvers:

- glpk - An open-source mixed-integer linear programming solver
- cbc - An open-source mixed-integer linear programming solver
- ipopt - An open-source interior point optimizer for continuous problems

Additionally, PAO can interface with optimization solvers at NEOS.

2 Overview

Many planning situations involve the analysis of a hierarchy of decision-makers with competing objectives. For example, policy decisions are made at different levels of a government, each of which has a different objective and decision space. Similarly, robust planning against adversaries is often modeled with a 2-level hierarchy, where the defensive planner makes decisions that account for adversarial response. Multilevel optimization techniques partition control over decision variables amongst the levels. Decisions at each level of the hierarchy may be constrained by decisions at other levels, and the objectives for each level may account for decisions made at other levels. The PAO library is designed to express this structure in a manner that is intuitive to users, and which facilitates the application of appropriate optimization solvers. In particular, PAO extends the modeling concepts in the Pyomo algebraic modeling language to express problems with an intuitive algebraic syntax.

However, data structures used to represent algebraic models are often poorly suited for complex numerical algorithms. Consequently, PAO supports *distinct* representations for algebraic models (using Pyomo) and compact problem representations that express objective and constraints using vector and matrix data types. Currently, PAO includes several *Multilevel Problem Representations* (MPRs) that represent multilevel optimization problems with an explicit, compact representation that simplifies the implementation of solvers for bilevel, trilevel and other multilevel optimization problems.

For example, PAO includes a compact representation for linear bilevel problems, `LinearMultilevelProblem`. Several solvers have been developed for problems expressed as a `LinearMultilevelProblem`, including the big-M method proposed by Fortuny-Amat and McCarl [3]. PAO can similarly express linear bilevel problems in Pyomo using a `Submodel` component, which was previously introduced in the **pyomo.bilevel** package [5]. Further, these Pyomo models can be automatically converted to the compact `LinearMultilevelProblem` representation and optimized with solvers tailored for that representation.

The use of independent problem representations in this manner has several implications for PAO. First, this design facilitates the development of solvers for algebraic modeling languages like Pyomo that are intrinsically more robust. Compact representations like `LinearMultilevelProblem` enable the development of solvers using natural operations (e.g. matrix-vector multiplication). Thus, we expect these solvers to be more robust and easier to maintain when compared to solvers developed using Pyomo data structures (e.g. expression trees). Additionally, the conversion of a Pyomo representation to a compact representation provides a context for verifying that the Pyomo model represents the intended problem form.

Second, this design facilitates the development of different problem representations that may or may not be inter-operable. Although PAO is derived from initial efforts with **pyomo.bilevel**, it has evolved from an extension of Pyomo's modeling capability to be a library that is synergistic with Pyomo but not strictly dependent on it.

The following sections provide detailed descriptions of these representations that are illustrated with increasingly complex examples, including

- bilevel
- bilevel with multiple lower-levels
- trilevel

Additionally, the following sections describe the setup of linear and quadratic problems, and the transformations that can be applied to them in PAO.

Note: We do not restrict the description of PAO representations to models that PAO can solve. Rather, a goal of this documentation to illustrate the breadth of the adversarial optimization problems that can be expressed with PAO, thereby motivating the development of new solvers for new classes of problems.

Note: Although the modeling abstractions in PAO are suitable for both pessimistic and optimistic multilevel optimization formulations, PAO currently assumes that all problems are expressed and solved in an an optimistic form. Thus, when there multiple solutions in lower-level problems, the choice of the solution is determined by the follower not the leader. This convention reflects the fact that solution techniques for pessimistic forms are not well-developed.

---

3 Simple Examples

We illustrate the PAO algebraic and compact representations with a series of simple examples. Consider the following bilevel problem introduced by Bard [2] (example 5.1.1):

$$\begin{array}{lll}
\min_{x\geq 0} & x-4y & \\
\text{s.t.} & \min_{y\geq 0} & y \\
 & \text{s.t.} & -x-y\leq -3 \\
 & & -2x+y\leq 0 \\
 & & 2x+y\leq 12 \\
 & & 3x-2y\leq 4
\end{array}$$

This problem has linear upper- and lower-level problems with different objectives in each level. Thus, this problem can be represented in PAO using a Pyomo model representation or the `LinearMultilevelProblem` representation.

3.1 Using Pyomo

The following python script defines a bilevel problem in Pyomo:

```
>>> import pyomo.environ as pe
>>> from pao.pyomo import *

# Create a model object
>>> M = pe.ConcreteModel()

# Define decision variables
>>> M.x = pe.Var(bounds=(0,None))
>>> M.y = pe.Var(bounds=(0,None))

# Define the upper-level objective
>>> M.o = pe.Objective(expr=M.x - 4*M.y)

# Create a SubModel component to declare a lower-level problem
# The variable M.x is fixed in this lower-level problem
>>> M.L = SubModel(fixed=M.x)

# Define the lower-level objective
>>> M.L.o = pe.Objective(expr=M.y)

# Define lower-level constraints
>>> M.L.c1 = pe.Constraint(expr=   -M.x -   M.y <= -3)
>>> M.L.c2 = pe.Constraint(expr= -2*M.x +   M.y <=  0)
>>> M.L.c3 = pe.Constraint(expr=  2*M.x +   M.y <= 12)
>>> M.L.c4 = pe.Constraint(expr=  3*M.x - 2*M.y <=  4)
```

(continues on next page)

(continued from previous page)

```
# Create a solver and apply it
>>> with Solver('pao.pyomo.FA') as solver:
...     results = solver.solve(M)

# The final solution is loaded into the model
>>> print(M.x.value)
4.0
>>> print(M.y.value)
4.0
```

The `SubModel` component defines a Pyomo block object within which the lower-level problem is declared. The `fixed` option is used to specify the upper-level variables whose value is fixed in the lower-level problem.

The `pao.pyomo.FA` uses the method for solving linear bilevel programs with big-M relaxations described by Fortuny-Amat and McCarl [3]. In this example, the default big-M value is used, but in general a user may need to explore suitable values for their problem. By default, the final values of the upper- and lower-level variables are loaded back into the Pyomo model. The `results` object contains information about the problem, the solver and the solver status.

3.2 Using LinearMultilevelProblem

Bilevel linear problems can also be represented using the `LinearMultilevelProblem` class. This class provides a simple mechanism for organizing data for variables, objectives and linear constraints. The following examples illustrate the use of `LinearMultilevelProblem` for Bard's example 5.1.1 described above, but this representation can naturally be used to express multi-level problems as well as problems with multiple lower-levels.

Using Numpy and Scipy Data

The following python script defines a bilevel problem using `LinearMultilevelProblem` with numpy and scipy data:

```
>>> import numpy as np
>>> from scipy.sparse import coo_matrix
>>> from pao.mpr import *

# Create a model object
>>> M = LinearMultilevelProblem()

# Declare the upper- and lower-levels, including the number of␣
↪decision-variables
```

(continues on next page)

(continued from previous page)

```
#  nxR=1 means there will be 1 real-valued decision variable
>>> U = M.add_upper(nxR=1)
>>> L = U.add_lower(nxR=1)

# Declare the bounds on the decision variables
>>> U.x.lower_bounds = np.array([0])
>>> L.x.lower_bounds = np.array([0])

# Declare the upper-level objective
#   U.c[X] is the array of coefficients in the objective for variables␣
↪in level X
>>> U.c[U] = np.array([1])
>>> U.c[L] = np.array([-4])

# Declare the lower-level objective, which has no upper-level decision-
↪variables
>>> L.c[L] = np.array([1])

# Declare the lower-level constraints
#   L.A[X] is the matrix coefficients in the constraints for variables␣
↪in level X
>>> L.A[U] = coo_matrix((np.array([-1, -2, 2, 3]),  # Coefficients
...                     (np.array([0, 1, 2, 3]),      # Row indices of␣
↪matrix entries
...                      np.array([0, 0, 0, 0]))))    # Column indices␣
↪of matrix entries
>>> L.A[L] = coo_matrix((np.array([-1, 1, 1, -2]),
...                     (np.array([0, 1, 2, 3]),
...                      np.array([0, 0, 0, 0]))))

# Declare the constraint right-hand-side
#   By default, constraints are inequalities, so these are upper-bounds
>>> L.b = np.array([-3, 0, 12, 4])

# Create a solver and apply it
>>> with Solver('pao.mpr.FA') as solver:
...     results = solver.solve(M)

# The final solution is loaded into the model
>>> print(U.x.values[0])
4.0
>>> print(L.x.values[0])
4.0
```

The `U` and `L` objects represent the upper- and lower-level respectively. When declaring these objects, the user specifies the number of real, integer and binary variables. The remaining declarations assume that these variables are used in that order. Thus, there is a single declaration for the objective coefficients, `c`, which is an array with values for each of the declared variables. However, the upper- and lower-level objective coefficients are separately declared for the upper- and lower-level variables by indexing `c` with `U` and `L` respectively. This example includes declarations for the upper- and lower-level variable bounds and objective coefficients. There are no upper-level constraints, so only the lower-level constriants are declared.

Note that the syntax for specifying solvers is analogous to that used with Pyomo models. The same solver options are available. The principle difference is the specification of the solver name that indicates the expected type of the model that will be solved.

Using Python Lists and Dictionaries

Although the constraint matrices are dense in this example, the `coo_matrix` is used to illustrate the general support for sparse data. The `LinearMultilevelProblem` class also supports a simpler syntax where dense arrays can be specified and Python lists and sparse matrices can be specified with Python tuple and dictionary objects:

```
>>> from pao.mpr import *

>>> M = LinearMultilevelProblem()

>>> U = M.add_upper(nxR=1)
>>> L = U.add_lower(nxR=1)

>>> U.x.lower_bounds = [0]
>>> L.x.lower_bounds = [0]

>>> U.c[U] = [1]
>>> U.c[L] = [-4]
>>> L.c[L] = [1]

>>> L.A[U] = (4,1), {(0,0):-1, (1,0):-2, (2,0):2, (3,0): 3}
>>> L.A[L] = (4,1), {(0,0):-1, (1,0): 1, (2,0):1, (3,0):-2}

>>> L.b = [-3, 0, 12, 4]

>>> with Solver('pao.mpr.FA') as solver:
...     results = solver.solve(M)

>>> print(U.x.values[0])
4.0
>>> print(L.x.values[0])
4.0
```

When specifying a sparse matrix, a tuple is provided (e.g. for `L.A[U]`). The first element is a

2-tuple that defines the shape of the matrix, and the second element is a dictionary that defines the non-zero values in the sparse matrix.

Similarly, a list-of-lists syntax can be used to specify dense matrices:

```
>>> from pao.mpr import *

>>> M = LinearMultilevelProblem()

>>> U = M.add_upper(nxR=1)
>>> L = U.add_lower(nxR=1)

>>> U.x.lower_bounds = [0]
>>> L.x.lower_bounds = [0]

>>> U.c[U] = [1]
>>> U.c[L] = [-4]
>>> L.c[L] = [1]

>>> L.A[U] = [[-1], [-2], [2], [3]]
>>> L.A[L] = [[-1], [1], [1], [-2]]
>>> L.b = [-3, 0, 12, 4]

>>> with Solver('pao.mpr.FA') as solver:
...     results = solver.solve(M)

>>> print(U.x.values[0])
4.0
>>> print(L.x.values[0])
4.0
```

When native Python data values are used to initialize a `LinearMultilevelProblem`, they are converted into numpy and scipy data types. This facilitates the use of `LinearMultilevelProblem` objects for defining numerical solvers using a consistent, convenient API for numerical operations (e.g. matrix-vector multiplication).

4 Pyomo Models

PAO can be used to express linear and quadratic problems in Pyomo using a *`SubModel`* component, which was previously introduced in the **pyomo.bilevel** package [5]. Pyomo represents optimization models using an model objects that are annotated with modeling component objects. Thus, *`SubModel`* component is a simple extension of the modeling concepts in Pyomo.

Hint: Advanced Pyomo users will realize that the PAO *`SubModel`* component is a special case of the Pyomo `Block` component, which is used to structure the expression of Pyomo models.

A *`SubModel`* component creates a context for expressing the objective and constraints in a lower-level model. Pyomo models can include nested and parallel *`SubModel`* components to express complex multilevel problems.

4.1 Bilevel Examples

Consider the following bilevel problem:

$$\textbf{Model PAO1} \qquad \begin{array}{ll} \min_{x\in[2,6],y} & x+3z \\ \text{s.t.} & x+y=10 \\ & \begin{array}{lll} \max_{z\geq 0} & z & \\ \text{s.t.} & x+z & \leq 8 \\ & x+4z & \geq 8 \\ & x+2z & \leq 13 \end{array} \end{array}$$

This problem has linear upper- and lower-level problems with different objectives in each level.

```
>>> M = pe.ConcreteModel()

>>> M.x = pe.Var(bounds=(2,6))
>>> M.y = pe.Var()

>>> M.L = pao.pyomo.SubModel(fixed=[M.x,M.y])
>>> M.L.z = pe.Var(bounds=(0,None))

>>> M.o = pe.Objective(expr=M.x + 3*M.L.z, sense=pe.minimize)
>>> M.c = pe.Constraint(expr= M.x + M.y == 10)

>>> M.L.o = pe.Objective(expr=M.L.z, sense=pe.maximize)
>>> M.L.c1 = pe.Constraint(expr= M.x + M.L.z <= 8)
>>> M.L.c2 = pe.Constraint(expr= M.x + 4*M.L.z >= 8)
>>> M.L.c3 = pe.Constraint(expr= M.x + 2*M.L.z <= 13)

>>> opt = pao.Solver("pao.pyomo.FA")
>>> results = opt.solve(M)
>>> print(M.x.value, M.y.value, M.L.z.value)
6.0 4.0 2.0
```

This example illustrates the flexibility of Pyomo representations in PAO:

- Each level can express different objectives with different senses
- Variables can be bounded or unbounded
- Equality and inequality constraints can be expressed

The `SubModel` component is used to define a logically separate optimization model that includes variables that are dynamically fixed by upper-level problems. All of the Pyomo objective and constraint declarations contained in the `SubModel` declaration are included in the sub-problem that it defines, even if they are nested in Pyomo `Block` components. The `SubModel` component also declares which variables are fixed in a lower-level problem. The value of the *fixed* argument is a Pyomo variable or a list of variables. For example, the following model expresses the upper-level variables with a single variable, *M.x*, which is fixed in the `SubModel` declaration:

```
>>> M = pe.ConcreteModel()

>>> M.x = pe.Var([0,1])
>>> M.x[0].setlb(2)
>>> M.x[0].setub(6)

>>> M.L = pao.pyomo.SubModel(fixed=M.x)
>>> M.L.z = pe.Var(bounds=(0,None))

>>> M.o = pe.Objective(expr=M.x[0] + 3*M.L.z, sense=pe.minimize)
>>> M.c = pe.Constraint(expr= M.x[0] + M.x[1] == 10)

>>> M.L.o = pe.Objective(expr=M.L.z, sense=pe.maximize)
>>> M.L.c1 = pe.Constraint(expr= M.x[0] + M.L.z <= 8)
>>> M.L.c2 = pe.Constraint(expr= M.x[0] + 4*M.L.z >= 8)
>>> M.L.c3 = pe.Constraint(expr= M.x[0] + 2*M.L.z <= 13)

>>> opt = pao.Solver("pao.pyomo.FA")
>>> results = opt.solve(M)
>>> print(M.x[0].value, M.x[1].value, M.L.z.value)
6.0 4.0 2.0
```

Although a lower-level problem is logically a separate optimization model, you cannot use a `SubModel` that is defined with a separate Pyomo model object. Pyomo implicitly requires that all variables used in objective and constraint expressions are attributes of the same Pyomo model. However, the location of variable declarations in a Pyomo model does *not* denote their use in upper- or lower-level problems. For example, consider the following model that re-expresses the previous problem:

```
>>> M = pe.ConcreteModel()

>>> M.x = pe.Var(bounds=(2,6))
>>> M.y = pe.Var()
>>> M.z = pe.Var(bounds=(0,None))

>>> M.o = pe.Objective(expr=M.x + 3*M.z, sense=pe.minimize)
>>> M.c = pe.Constraint(expr= M.x + M.y == 10)
```

(continues on next page)

(continued from previous page)

```
>>> M.L = pao.pyomo.SubModel(fixed=[M.x,M.y])
>>> M.L.o = pe.Objective(expr=M.z, sense=pe.maximize)
>>> M.L.c1 = pe.Constraint(expr= M.x + M.z <= 8)
>>> M.L.c2 = pe.Constraint(expr= M.x + 4*M.z >= 8)
>>> M.L.c3 = pe.Constraint(expr= M.x + 2*M.z <= 13)

>>> opt = pao.Solver("pao.pyomo.FA")
>>> results = opt.solve(M)
>>> print(M.x.value, M.y.value, M.z.value)
6.0 4.0 2.0
```

Note that *all* of the decision variables are declared outside of the `SubModel` component, even though the variable `M.z` is a lower-level variable. The declarations of `SubModel` components defines the mathematical role of all decision variables in a Pyomo model. As this example illustrates, the specification of a bilevel problem can be simplified if all variables are expressed at once.

Finally, we observe that PAO's Pyomo representation only works with a subset of the many different modeling components that are supported in Pyomo:

- `Set` - Set declarations
- `Param` - Parameter declarations
- `Var` - Variable declarations
- `Block` - Defines a subset of a model
- `Objective` - Define a model objective
- `Constraint` - Define model constraints

Additional Pyomo modeling components will be added to PAO as motivating applications arise and as suitable solvers become available.

4.2 Multilevel Examples

Multilevel problems can be easily expressed with Pyomo using multiple declarations of `SubModel`.

Multiple Lower Levels

Consider the following bilevel problem that extends the **PAO1** model to include two equivalent lower-levels:

Model PAO2

$$
\begin{array}{lllll}
\min_{x\in[2,6],y} & x+3z_1+3z_2 & & & \\
\text{s.t.} & x+y=10 & & & \\
 & \max_{z_1\geq 0} & z_1 & & \\
 & \text{s.t.} & x+z_1 & \leq 8 & \\
 & & x+4z_1 & \geq 8 & \\
 & & x+2z_1 & \leq 13 & \\
 & \max_{z_2\geq 0} & z_2 & & \\
 & \text{s.t.} & y+z_2 & \leq 8 & \\
 & & y+4z_2 & \geq 8 & \\
 & & y+2z_2 & \leq 13 &
\end{array}
$$

The **PAO2** model can be expressed in Pyomo as follows:

```
>>> M = pe.ConcreteModel()

>>> M.x = pe.Var(bounds=(2,6))
>>> M.y = pe.Var()
>>> M.z = pe.Var([1,2], bounds=(0,None))

>>> M.o = pe.Objective(expr=M.x + 3*M.z[1]+3*M.z[2], sense=pe.minimize)
>>> M.c = pe.Constraint(expr= M.x + M.y == 10)

>>> M.L1 = pao.pyomo.SubModel(fixed=[M.x])
>>> M.L1.o = pe.Objective(expr=M.z[1], sense=pe.maximize)
>>> M.L1.c1 = pe.Constraint(expr= M.x + M.z[1] <= 8)
>>> M.L1.c2 = pe.Constraint(expr= M.x + 4*M.z[1] >= 8)
>>> M.L1.c3 = pe.Constraint(expr= M.x + 2*M.z[1] <= 13)

>>> M.L2 = pao.pyomo.SubModel(fixed=[M.y])
>>> M.L2.o = pe.Objective(expr=M.z[2], sense=pe.maximize)
>>> M.L2.c1 = pe.Constraint(expr= M.y + M.z[2] <= 8)
>>> M.L2.c2 = pe.Constraint(expr= M.y + 4*M.z[2] >= 8)
>>> M.L2.c3 = pe.Constraint(expr= M.y + 2*M.z[2] <= 13)

>>> opt = pao.Solver("pao.pyomo.FA")
>>> results = opt.solve(M)
```

(continues on next page)

(continued from previous page)

```
>>> print(M.x.value, M.y.value, M.z[1].value, M.z[2].value)
2.0 8.0 5.5 0.0
```

Trilevel Problems

Trilevel problems can be described with nested declarations of *SubModel* components. Consider the following trilevel continuous linear problem described by Anadalingam [1]:

Model Anadalingam1988

$$\begin{array}{llll}
\min_{x_1 \geq 0} & -7x_1 - 3x_2 + 4x_3 & & \\
\text{s.t.} & \min_{x_2 \geq 0} & -x_2 & \\
& \text{s.t.} & \min_{x_3 \in [0,0.5]} & -x_3 \\
& & \text{s.t.} & x_1 + x_2 + x_3 \leq 3 \\
& & & x_1 + x_2 - x_3 \leq 1 \\
& & & x_1 + x_2 + x_3 \geq 1 \\
& & & -x_1 + x_2 + x_3 \leq 1
\end{array}$$

This model can be expressed in Pyomo as follows:

```
>>> M = pe.ConcreteModel()
>>> M.x1 = pe.Var(bounds=(0,None))
>>> M.x2 = pe.Var(bounds=(0,None))
>>> M.x3 = pe.Var(bounds=(0,0.5))

>>> M.L = pao.pyomo.SubModel(fixed=M.x1)

>>> M.L.B = pao.pyomo.SubModel(fixed=M.x2)

>>> M.o = pe.Objective(expr=-7*M.x1 - 3*M.x2 + 4*M.x3)

>>> M.L.o = pe.Objective(expr=-M.x2)
>>> M.L.B.o = pe.Objective(expr=-M.x3)

>>> M.L.B.c1 = pe.Constraint(expr=   M.x1 + M.x2 + M.x3 <= 3)
>>> M.L.B.c2 = pe.Constraint(expr=   M.x1 + M.x2 - M.x3 <= 1)
>>> M.L.B.c3 = pe.Constraint(expr=   M.x1 + M.x2 + M.x3 >= 1)
>>> M.L.B.c4 = pe.Constraint(expr= - M.x1 + M.x2 + M.x3 <= 1)
```

Note: PAO solvers cannot currently solve trilevel problems like this, but an issue has been submitted to add this functionality.

Bilinear Problems

PAO models using Pyomo represent general quadratic problems with quadratic terms in the objective and constraints at each level. The special case where bilinear terms arise with an upper-level binary variable multiplied with a lower-level variable is common in many applications. For this case, the PAO solvers for Pyomo models include an option to linearize these bilinear terms.

The following models considers a variation of the **PAO1** model where binary variables control the expression of lower-level constraints:

Model PAO3

$$
\begin{array}{lll}
\min_{x\in[2,6],y,w_1,w_2} & x+3z+5w_1 & \\
\text{s.t.} & x+y=10 & \\
& w_1+w_2 \geq 1 & \\
& w_1,w_2 \in \{0,1\} & \\
& \quad \max_{z\geq 0} \quad z & \\
& \quad \text{s.t.} \quad x+w_1 z & \leq 8 \\
& \quad\quad\quad\; x+4z & \geq 8 \\
& \quad\quad\quad\; x+2w_2 z & \leq 13
\end{array}
$$

The **PAO3** model can be expressed in Pyomo as follows:

```
>>> M = pe.ConcreteModel()

>>> M.w = pe.Var([1,2], within=pe.Binary)
>>> M.x = pe.Var(bounds=(2,6))
>>> M.y = pe.Var()
>>> M.z = pe.Var(bounds=(0,None))

>>> M.o = pe.Objective(expr=M.x + 3*M.z+5*M.w[1], sense=pe.minimize)
>>> M.c1 = pe.Constraint(expr= M.x + M.y == 10)
>>> M.c2 = pe.Constraint(expr= M.w[1] + M.w[2] >= 1)

>>> M.L = pao.pyomo.SubModel(fixed=[M.x,M.y,M.w])
>>> M.L.o = pe.Objective(expr=M.z, sense=pe.maximize)
>>> M.L.c1 = pe.Constraint(expr= M.x + M.w[1]*M.z <= 8)
>>> M.L.c2 = pe.Constraint(expr= M.x + 4*M.z >= 8)
>>> M.L.c3 = pe.Constraint(expr= M.x + 2*M.w[2]*M.z <= 13)

>>> opt = pao.Solver("pao.pyomo.FA", linearize_bigm=100)
>>> results = opt.solve(M)
>>> print(M.x.value, M.y.value, M.z.value, M.w[1].value, M.w[2].value)
6.0 4.0 3.5 0 1
```

In general, it may be difficult to determine a valid value of M. Thus, this transformation may result in a restriction or relaxation of the original problem (depending on where the big-M values are introduced).

5 Multilevel Representations

PAO includes several *Multilevel Problem Representations* (MPRs) that represent multilevel optimization problems with an explicit, compact representation that simplifies the implementation of solvers for bilevel, trilevel and other multilevel optimization problems. These MPRs express objective and constraints using vector and matrix data types. However, they organize these data types to provide a semantically clear organization of multilevel problems. Additionally, the MPRs provide checks to ensure the consistency of the data within and across levels.

The classes `LinearMultilevelProblem` and `QuadraticMultilevelProblem` respectively represent linear and quadratic multilevel problems. Although `QuadraticMultilevelProblem` is a generalization of the representation in `LinearMultilevelProblem`, the use of tailored representations for different classes of problems clarifies the semantic context when using them. For example, this allows for simple error checking to confirm that a problem is linear.

Currently, all PAO solvers for MPRs support only linear problems, so the following sections focus on `LinearMultilevelProblem`. However, we conclude with an example of model transformations that enable the solution of quadratic problems using `QuadraticMultilevelProblem`.

Note: We do not expect many users to directly employ a MPR data representation for their applications. Perhaps this would be desirable if their problem was already represented with matrix and vector data. In general, the algebraic representation supported by Pyomo will be more convenient for large, complex applications.

We expect this representation to be more useful for researchers developing multilevel solvers, since the MPR representations provide structure that simplifies the expression of necessary mathematical operations for these problems.

5.1 Linear Bilevel Examples

We consider again the bilevel problem PAO1 (4.1). This problem has linear upper- and lower-level problems with different objectives in each level.

```
>>> M = pao.mpr.LinearMultilevelProblem()

>>> U = M.add_upper(nxR=2)
>>> L = U.add_lower(nxR=1)

>>> U.x.lower_bounds = [2, np.NINF]
>>> U.x.upper_bounds = [6, np.PINF]
>>> L.x.lower_bounds = [0]
```

(continues on next page)

(continued from previous page)

```
>>> L.x.upper_bounds = [np.PINF]

>>> U.c[U] = [1, 0]
>>> U.c[L] = [3]

>>> L.c[L] = [1]
>>> L.maximize = True

>>> U.equalities = True
>>> U.A[U] = [[1, 1]]
>>> U.b = [10]

>>> L.A[U] = [[ 1, 0],
...           [-1, 0],
...           [ 1, 0]]
>>> L.A[L] = [[ 1],
...           [-4],
...           [ 2]]
>>> L.b = [8, -8, 13]

>>> M.check()

>>> opt = pao.Solver("pao.mpr.FA")
>>> results = opt.solve(M)
>>> print(M.U.x.values)
[6.0, 4.0]
>>> print(M.U.LL.x.values)
[2.0]
```

The example illustrates both the flexibility of the MPR represenetions in PAO but also the structure they enforce on the multilevel problem representation. The upper-level problem is created by calling the `add_upper()` method, which takes arguments that specify the variables at that level:

- `nxR` - The number of real variables (Default: 0)
- `nxZ` - The number of general integer variables (Default: 0)
- `nxB` - The number of binary variables (Default: 0)

In each level, the variables are represented as a vector of values, ordered in this manner.

Similarly, the `add_lower()` method is used to generate a lower-level problem from a given level. Note that this allows for the specification of arbitrary nesting of levels, since a lower-level can be defined relative to any other level in the model. Additionally, multiple lower-levels can be specified for relative to a single level (see below).

The `add_upper()` and `add_lower()` methods return the corresponding level object, which is used to specify data in the model later.

For a given level object, `Z`, the data `Z.x` contains information about the decision variables. In particular, the values `Z.x.lower_bounds` and `Z.x.upper_bounds` can be set with arrays of numeric values to specify lower- and upper-bounds on the decision variables. Note that missing lower- and upper-bounds are specified with `numpy.NINF` and `numpy.PINF` respectively.

The `Z.c` data specifies coefficients of the objective function for this level. This data is indexed by a level object `B` to indicate the data associated with the variables in `B`. In the example above:

- `U.c[U]` is the array of coefficients of the upper-level objective for the variables in the upper-level,
- `U.c[L]` is the array of coefficients of the upper-level objective for the variables in the lower-level, and
- `L.c[L]` is the array of coefficients of the lower-level objective for the variables in the lower-level.

Since `L.c[U]` is not specified, it has a value `None` that indicates that no upper-level variables have non-zero coefficients in the lower-level objective. The `Z.A` data specifies the matrix coefficients for the constraints using a similar indexing notation and semantics.

The values `Z.minimize` and `Z.maximize` can be set to `True` to indicate whether the objective in `Z` minimizes or maximizes. (The default is minimize.) Similarly the value `Z.inequalities` and `Z.equalities` can be set to `True` to indicate whether the constraints in `Z` are inequalities or equalities. (The default is inequalities.) Finally, the value `Z.b` defines the array of constraint right-hand-side values.

The :meth:`check` method provides a convenient sanity check that the data is defined consistently within each level and between levels.

Note that PAO supports a consistent interface for creating a solver interface and for applying solvers. In fact, the user should be aware that Pyomo and MPR solvers are named in a consistent fashion. For example, the Pyomo solver **pao.pyomo.FA** calls the MPR solver **pao.mpr.FA** after automatically converting the Pyomo representation to a :class:`LinearMultilevelProblem` representation. This example illustrates that values `Z.x.values` contains the values of each level `Z` after optimization.

5.2 Multilevel Examples

Multilevel problems can be easily expressed using the same MPR data representation.

Multiple Lower Levels

We consider again the bilevel problem PAO2 (4.2), which has has multiple lower-level problems. The **PAO2** model can be expressed as a linear multilevel problem as follows:

```
>>> M = pao.mpr.LinearMultilevelProblem()

>>> U = M.add_upper(nxR=2)
>>> L1 = U.add_lower(nxR=1)
>>> L2 = U.add_lower(nxR=1)

>>> U.x.lower_bounds = [2, np.NINF]
>>> U.x.upper_bounds = [6, np.PINF]
>>> U.c[U] = [1, 0]
>>> U.c[L1] = [3]
>>> U.c[L2] = [3]
>>> U.equalities = True
>>> U.A[U] = [[1, 1]]
>>> U.b = [10]

>>> L1.x.lower_bounds = [0]
>>> L1.x.upper_bounds = [np.PINF]
>>> L1.c[L1] = [1]
>>> L1.maximize = True
>>> L1.A[U] = [[ 1, 0],
...            [-1, 0],
...            [ 1, 0]]
>>> L1.A[L1] = [[ 1],
...             [-4],
...             [ 2]]
>>> L1.b = [8, -8, 13]

>>> L2.x.lower_bounds = [0]
>>> L2.x.upper_bounds = [np.PINF]
>>> L2.c[L2] = [1]
>>> L2.maximize = True
>>> L2.A[U] = [[0,  1],
...            [0, -1],
...            [0,  1]]
>>> L2.A[L2] = [[ 1],
...             [-4],
...             [ 2]]
>>> L2.b = [8, -8, 13]

>>> opt = pao.Solver("pao.mpr.FA")
>>> results = opt.solve(M)
```

(continues on next page)

(continued from previous page)

```
>>> print(U.x.values)
[2.0, 8.0]
>>> print(L1.x.values)
[5.5]
>>> print(L2.x.values)
[0.0]
```

The declarataion of the two lower level problems is naturally contained within the data of the `L1` and `L2` objects. Further, the cross-level interactions are intuitively represented using the index notation for the objective and constraint data objects.

Note that this more explicit representation clarifies some ambiguity in the expression of lower-levels in the Pyomo representation. The Pyomo representation of PAO2 only specifies the fixed variables that are **used** in each of the two lower-level problems. PAO analyzes the use of decision variables in Pyomo models, and treats *unused* variables as fixed. Thus, the Pyomo and MPR representations generate a consistent interpretation of the variable specifications. However, the MPR representation is more explicit in this regard.

Trilevel Problems

We consider again the trilevel problem described by Anadalingam (4.2), which can be expressed as a trilevel linear problem as follows:

```
>>> M = pao.mpr.LinearMultilevelProblem()

>>> U = M.add_upper(nxR=1)
>>> U.x.lower_bounds = [0]

>>> L = U.add_lower(nxR=1)
>>> L.x.lower_bounds = [0]

>>> B = L.add_lower(nxR=1)
>>> B.x.lower_bounds = [0]
>>> B.x.upper_bounds = [0.5]

>>> U.minimize = True
>>> U.c[U] = [-7]
>>> U.c[L] = [-3]
>>> U.c[B] = [4]

>>> L.minimize = True
>>> L.c[L] = [-1]

>>> B.minimize = True
```

(continues on next page)

(continued from previous page)

```
>>> B.c[B] = [-1]
>>> B.inequalities = True
>>> B.A[U] = [[1], [ 1], [-1], [-1]]
>>> B.A[L] = [[1], [ 1], [-1], [ 1]]
>>> B.A[B] = [[1], [-1], [-1], [ 1]]
>>> B.b = [3,1,-1,1]
```

Bilinear Problems

The *QuadraticMultilevelProblem* class can represent general quadratic problems with quadratic terms in the objective and constraints at each level. The special case where bilinear terms arise with an upper-level binary variable multiplied with a lower-level variable is common in many applications. For this case, PAO provides a function to linearize these bilinear terms.

We consider again the bilevel problem PAO3 (4.2), which is represented and solved as follows:

```
>>> M = pao.mpr.QuadraticMultilevelProblem()

>>> U = M.add_upper(nxR=2, nxB=2)
>>> L = U.add_lower(nxR=1)

>>> U.x.lower_bounds = [2, np.NINF, 0, 0]
>>> U.x.upper_bounds = [6, np.PINF, 1, 1]
>>> L.x.lower_bounds = [0]
>>> L.x.upper_bounds = [np.PINF]

>>> U.c[U] = [1, 0, 5, 0]
>>> U.c[L] = [3]

>>> L.c[L] = [1]
>>> L.maximize = True

>>> U.A[U] = [[ 1,  1,  0,  0],
...           [-1, -1,  0,  0],
...           [ 0,  0, -1, -1]
...           ]
>>> U.b = [10, -10, -1]

>>> L.A[U] = [[ 1, 0, 0, 0],
...           [-1, 0, 0, 0],
...           [ 1, 0, 0, 0]]
>>> L.A[L] = [[ 0],
...           [-4],
...           [ 0]]
```

(continues on next page)

(continued from previous page)

```
>>> L.Q[U,L] = (3,4,1), {(0,2,0):1, (2,3,0):2}

>>> L.b = [8, -8, 13]

>>> lmr, soln = pao.mpr.linearize_bilinear_terms(M, 100)
>>> opt = pao.Solver("pao.mpr.FA")
>>> results = opt.solve(lmr)
>>> soln.copy(From=lmr, To=M)
>>> print(U.x.values)
[6.0, 4.0, 0, 1]
>>> print(L.x.values)
[3.5]
```

The data `L.Q[U,L]` specifies the bilinear terms multiplying variables from level `U` with variables from level `L`, which are included in the constraints in level `L`. Note that `Q` is a tensor, which is indexed over the constraints, upper-level variables and lower-level variables. A similar syntax is used to define bilinear terms in objectives, `P`, though that is represented as a sparse matrix. Quadratic terms can be specified simply by using the same levels to index `Q` or `P`.

Model transformations like :func:`.linearize_bilinear_terms` are described in further detail in the next section. Note that this function returns both the transformed model as well as a helper class that maps solutions back to the original model. This logic facilitates the automation of model transformations within PAO.

6 Solvers

After formulating a multilevel problem, PAO users will generally need to (1) transform the model to a standard form, and (2) apply an optimizer to solve the problem. The examples in the previous sections illustrate that step (1) is often optional; PAO automates the applications of several model transformations, particularly for problems formulated with Pyomo. The following section summarizes the solvers available in PAO, and describes how PAO manages solvers. Section transformations describes model transformations in PAO.

6.1 Summary of PAO Solvers

The following summarizes the current solvers available in PAO:

- pao.mpr.FA, pao.pyomo.FA

  PAO solver for Multilevel Problem Representations that define linear bilevel problems. Solver uses big-M relaxations discussed by Fortuny- Amat and McCarl (1981).

- pao.mpr.MIBS, pao.pyomo.MIBS

PAO solver for Multilevel Problem Representations using the COIN-OR MibS solver by Tahernejad, Ralphs, and DeNegre (2020).

- pao.mpr.PCCG, pao.pyomo.PCCG

  PAO solver for Multilevel Problem Representations that define linear bilevel problems. Solver uses projected column constraint generation algorithm described by Yue et al. (2017).

- pao.mpr.REG, pao.pyomo.REG

  PAO solver for Multilevel Problem Representations that define linear bilevel problems. Solver uses regularization discussed by Scheel and Scholtes (2000) and Ralph and Wright (2004).

The following table summarize key features of the problems these solvers can be applied to:

		Solver			
Problem Feature		*FA*	*REG*	*PCCG*	*MibS*
Equation Structure	Linear	Y	Y	Y	Y
	Bilinear	Y	Y	Y	Y
	Nonlinear				
Upper-Level Variables	Integer	Y		Y	Y
	Real	Y	Y	Y	Y
Lower-Level Variables	Integer			Y	Y
	Real	Y	Y	Y	Y
Multilevel Representation	Bilevel	Y	Y	Y	Y
	Trilevel				
	k-Bilevel	Y	Y		

Note: The iterface to MibS is a prototype that has not been well-tested. This interface will be documented and finalized in an upcoming release of PAO.

6.2 The Solver Interface

The `Solver` object provides a single interface for setting up an interface to optimizers in PAO. This includes *both* PAO solvers for multilevel optimization problems, but also interfaces to conventional numerical solvers that are used by PAO solvers. We illustrate this distinction with the following example, which optimizes the PAO1 (4.1) example:

```
>>> # Create an interface to the PAO FA solver
>>> opt = pao.Solver("pao.pyomo.FA")
```

(continues on next page)

(continued from previous page)

```
>>> # Optimize the model
>>> # By default, FA uses the glpk MIP solver
>>> results = opt.solve(M)
>>> print(M.x.value, M.y.value, M.L.z.value)
6.0 4.0 2.0


>>> # Create an interface to the PAO FA solver, using cbc
>>> opt = pao.Solver("pao.pyomo.FA", mip_solver="cbc")

>>> # Optimize the model using cbc
>>> results = opt.solve(M)
>>> print(M.x.value, M.y.value, M.L.z.value)
6.0 4.0 2.0
```

The `Solver` object is initialized using the solver name followed by solver-specific options. In this case, the FA algorithm accepts the `mip_solver` option that specifies the mixed-integer programming (MIP) solver that is used to solve the MIP that is generated by FA after reformulating the bilevel problem. The value of `mip_solver` is itself an optimizer. As illustrated here, this option can simply be the string name of the MIP solver that will be used. However, the `Solver` object can be used to define a MIP solver interface as well:

```
>>> # Create an interface to the cbc MIP solver
>>> mip = pao.Solver("cbc")
>>> # Create an interface to the PAO FA solver, using cbc
>>> opt = pao.Solver("pao.pyomo.FA", mip_solver=mip)

>>> # Optimize the model using cbc
>>> results = opt.solve(M)
>>> print(M.x.value, M.y.value, M.L.z.value)
6.0 4.0 2.0
```

This enables the customization of the MIP solver used by FA. Note that the `solve()` method accepts the same options as `Solve`. This allows for more dynamic specification of solver options:

```
>>> # Create an interface to the cbc MIP solver
>>> cbc = pao.Solver("cbc")
>>> # Create an interface to the glpk MIP solver
>>> glpk = pao.Solver("glpk")

>>> # Create an interface to the PAO FA solver
>>> opt = pao.Solver("pao.pyomo.FA")

>>> # Optimize the model using cbc
```

(continues on next page)

(continued from previous page)

```
>>> results = opt.solve(M, mip_solver=cbc)
>>> print(M.x.value, M.y.value, M.L.z.value)
6.0 4.0 2.0

>>> # Optimize the model using glpk
>>> results = opt.solve(M, mip_solver=glpk)
>>> print(M.x.value, M.y.value, M.L.z.value)
6.0 4.0 2.0
```

Warning: The `solve()` current passes unknown keyword arguments to the optimizer used by PAO solvers, but this feature will be disabled.

PAO Solvers

Solvers developed in PAO have names that begin with `pao.`. The current set of available PAO solvers can be queried using the `Solver` object:

```
>>> for name in pao.Solver:
...     print(name)
pao.mpr.FA
pao.mpr.MIBS
pao.mpr.PCCG
pao.mpr.REG
pao.pyomo.FA
pao.pyomo.MIBS
pao.pyomo.PCCG
pao.pyomo.REG

>>> pao.Solver.summary()
pao.mpr.FA
    PAO solver for Multilevel Problem Representations that define␣
↪linear
    bilevel problems.  Solver uses big-M relaxations discussed by␣
↪Fortuny-
    Amat and McCarl (1981).

pao.mpr.MIBS
    PAO solver for Multilevel Problem Representations using the COIN-OR
    MibS solver by Tahernejad, Ralphs, and DeNegre (2020).

pao.mpr.PCCG
    PAO solver for Multilevel Problem Representations that define␣
↪linear
```

(continues on next page)

(continued from previous page)

```
    bilevel problems. Solver uses projected column constraint␣
↪generation
    algorithm described by Yue et al. (2017).

pao.mpr.REG
    PAO solver for Multilevel Problem Representations that define␣
↪linear
    bilevel problems.  Solver uses regularization discussed by Scheel␣
↪and
    Scholtes (2000) and Ralph and Wright (2004).

pao.pyomo.FA
    PAO solver for Pyomo models that define linear and bilinear bilevel
    problems.  Solver uses big-M relaxations discussed by Fortuny-Amat␣
↪and
    McCarl (1981).

pao.pyomo.MIBS
    PAO solver for Multilevel Problem Representations using the COIN-OR
    MibS solver by Tahernejad, Ralphs, and DeNegre (2020).

pao.pyomo.PCCG
    PAO solver for Pyomo models that define linear and bilinear bilevel
    problems.  Solver uses projected column constraint generation
    algorithm described by Yue et al. (2017)

pao.pyomo.REG
    PAO solver for Pyomo models that define linear and bilinear bilevel
    problems.  Solver uses regularization discussed by Scheel and␣
↪Scholtes
     (2000) and Ralph and Wright (2004).
```

The `solve()` method includes documentation describing the keyword arguments for a specific solver. For example:

```
>>> opt = pao.Solver("pao.pyomo.FA")
>>> help(opt.solve)
Help on method solve in module pao.pyomo.solvers.mpr_solvers:

solve(model, **options) method of pao.pyomo.solvers.mpr_solvers.
↪PyomoSubmodelSolver_FA instance
    Executes the solver and loads the solution into the model.

    Parameters
    ----------
```

(continues on next page)

(continued from previous page)

```
    model
        The model that is being optimized.
    options
        Keyword options that are used to configure the solver.

    Keyword Arguments
    -----------------
    tee
      If True, then solver output is streamed to stdout. (default is␣
↪False)
    load_solutions
      If True, then the finale solution is loaded into the model.␣
↪(default is True)
    linearize_bigm
      The name of the big-M value used to linearize bilinear terms. ␣
↪If this is not specified, then the solver will throw an error if␣
↪bilinear terms exist in the model.
    mip_solver
      The MIP solver used by FA.  (default is glpk)

    Returns
    -------
    Results
        A summary of the optimization results.
```

The `solve()` method returns a results object that contains data about the optimization process. In particular, this object contains information about the termination conditions for the solver. The `check_optimal_termination()` method can be used confirm that the termination condition indicates that an optimal solution was found. For example:

```
>>> nlp = pao.Solver('ipopt', print_level=3)
>>> opt = pao.Solver('pao.pyomo.REG', nlp_solver=nlp)
>>> results = opt.solve(M)
>>> print(results.solver.termination_condition)
TerminationCondition.optimal
>>> results.check_optimal_termination()
True
```

Pyomo Solvers

The `Solver` object also provides a convenient interface to conventional numerical solvers. Currently, solver objects constructed by `Solver` are simple wrappers around Pyomo optimization solver objects. This interface supports two types of solver interfaces: (1) solvers that execute locally, and (2) solvers that execute on remote servers.

When optimizating a **Pyomo** model, solver parameters can be setup both when the solver interface is created and when a model is optimized. For example:

```
>>> # This is a nonlinear toy problem modeled with Pyomo
>>> NLP = pe.ConcreteModel()
>>> A = list(range(10))
>>> NLP.x = pe.Var(A, bounds=(0,None), initialize=1)
>>> NLP.o = pe.Objective(expr=sum(pe.sin((i+1)*NLP.x[i]) for i in A))
>>> NLP.c = pe.Constraint(expr=sum(NLP.x[i] for i in A) >= 1)

>>> nlp = pao.Solver('ipopt', print_level=3)
>>> # Apply ipopt with print level 3
>>> results = nlp.solve(NLP)
>>> # Override the default print level to using 5
>>> results = nlp.solve(NLP, print_level=5)
```

However, PAO users will typically setup solver parameters when the Pyomo solver is initially created:

```
>>> nlp = pao.Solver('ipopt', print_level=3)
>>> opt = pao.Solver('pao.pyomo.REG', nlp_solver=nlp)
>>> results = opt.solve(M)
```

When executing locally, the `executable` option can be used to explicitly specify the path to the executable that is used by this solver. This is helpful in contexts where Pyomo is not automatically finding the *correct* optimizer executable in a user's shell environment.

When executing on a remote server, the `server` is used to specify the server that is used. Currently, only the `neos` server is supported, which allows the user to perform optimization at NEOS [4]. The NEOS server requires a user to specify a valid email address:

```
>>> nlp = pao.Solver('ipopt', server='neos', email='pao@gmail.com')
>>> opt = pao.Solver('pao.pyomo.REG', nlp_solver=nlp)
>>> results = opt.solve(M)
```

Warning: There is no common reference for solver-specific parameters for the solvers available in Pyomo. These are generally documented with solver documentation, and users should expect to contact solver developers to learn about these.

7 Model Transformations

PAO includes a variety of functions that transform models, which generally are applied as follows:

```
>>> new_model, soln = transform_function(old_model)
```

Here, the function `transform_func` generates the model `new_model` from the model `old_model`. The object `soln` is used to map a solution back to the old model:

```
>>> soln.copy(From=new_model, To=old_model)
```

The following transformation functions are documented and suitable for use by end-users:

- *`pao.pyomo.convert.convert_pyomo2MultilevelProblem()`*

  This function generates a *`LinearMultilevelProblem`* or *`QuadraticMultilevelProblem`* from a Pyomo model. By default, all constraints in the MPR representation are inequalities.

- *`pao.mpr.convert_repn.linearize_bilinear_terms()`*

  This function generates a *`LinearMultilevelProblem`* from a *`QuadraticMultilevelProblem`* that only contains bilinear terms. This transformation currently is limited to MPRs that only contain inequality constraints.

- *`pao.mpr.convert_repn.convert_to_standard_form()`*

  This function generates an equivalent linear multilevel representation for which all variables are non-negative and all constraints have the same form (inequalities or equalities). This simplifies the implementation of solvers, which typically assume a standard form for subproblems.

8 Library Reference

The following classes and functions represent the core functionality in PAO:

8.1 Solver API

class SolverAPI

The base class for all PAO solvers.

The SolverAPI class defines a consistent API for optimization solvers.

__bool__()

Raises an error because this class cannot be interpreted with a boolean.

Raises **RuntimeError** – Casting a solver to bool is not allowed.

__enter__()

Setup a solver and return it within a context manager.

Returns Return a reference to **self**.

Return type *SolverAPI*

__exit__(*t*, *v*, *traceback*)

Cleanup the solver at the end of a context manager.

available()

Returns a bool indicating if the solver can be executed.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to confirm that a solver license is available).

Returns This method returns True if the solver can be executed.

Return type bool

is_persistent()

Returns True if the solver is persistent.

Persistent solvers maintain the model representation in memory, which enables performance optimization when a problem is resolved after changing initial conditions or tweaking model parameters.

The default is to return False, but this method can be overloaded in a subclass to support solver-specific logic.

Returns This method returns True if the solver is persistent.

Return type bool

abstract solve(*model*, ***options*)

Executes the solver and loads the solution into the model.

Parameters

- **model** – The model that is being optimized.

- **`options`** – Keyword options that are used to configure the solver.

Keyword Arguments

- **`tee`** – If True, then solver output is streamed to stdout. (default is False)
- **`load_solutions`** – If True, then the finale solution is loaded into the model. (default is True)

Returns A summary of the optimization results.

Return type *Results*

`valid_license``()`

Returns a bool indicating if the solver has a valid license.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to check the solver license).

Returns This method returns True if the solver license is valid.

Return type bool

`version``()`

Returns a tuple that describes the solver version.

The return value is a tuple of strings. A typical format is (major, minor, patch), but this is not required. The default behavior is to return an empty tuple.

Returns The solver version.

Return type tuple

`class SolverFactory`

A class that manages a registry of solvers.

A solver factory manages a registry that enables solvers to be created by name.

`__call__``(`*name*, ***options*`)`

Constructs the specified solver.

This method creates a class instance for the solver that is specified.

Parameters

- **`name`** (`str`) – The name of a solver
- **`options`** – Keyword options that are used to configure the solver.

Returns A solver class instance for the solver that is specified.

Return type *SolverAPI*

`__iter__``()`

Yields *string* – Yields the next solver name that has been registered

description(*name*)
Returns the description of the specified solver.

Parameters **name** (*str*) – The name of a solver

Returns A short description of the specified solver

Return type str

register(*cls=None*, ***, *name=None*, *doc=None*)
Register a solver with the specified name.

Parameters

- **cls** – Class type for the solver
- **name** (*str*) – Unique name of the solver
- **doc** (*str*) – Short description of the solver

Returns If the **cls** parameter is None, then a class decorator function is returned that can be used to register a solver.

Return type decorator

summary()
Print a summary of all solvers.

class Results(*found_feasible_solution=None*)
The results object.

__str__()
Generate a string summary of this results object.

Returns A string summarizing the data in this object.

Return type str

check_optimal_termination()
This function returns True if the termination condition for the solver is 'optimal', 'locallyOptimal', or 'globallyOptimal', and the status is 'ok'

Returns

Return type bool

found_feasible_solution()

Returns True if at least one feasible solution was found. False otherwise.

Return type bool

load_from(*model*)
Load solution from the model.

When completed, this results object contains only one solution, which corresponds to the solution in the model.

`store_to`(*model*, *i=0*)
Store the solution in this object into the given model.

A results object may contain one or more solutions. This method copies the **i**-th solution into the model.

Parameters **`i`** (*`int`*) – The index of the solution copied into the model.

`class TerminationCondition`(*value*)
The TerminationCondition class defines a enumeration of optimization termination conditions.

`error = 11`
The solver exited due to an error

`globallyOptimal = 7`
The solver exited with a locally optimal solution

`infeasible = 9`
The solver exited because the problem is infeasible

`infeasibleOrUnbounded = 10`
The solver exited because the problem is either infeasible or unbounded

`interrupted = 12`
The solver exited because it was interrupted

`licensingProblems = 13`
The solver exited due to licensing problems

`locallyOptimal = 6`
The solver exited with a locally optimal solution

`maxIterations = 2`
The solver exited due to an iteration limit

`maxTimeLimit = 1`
The solver exited due to a time limit

`minStepLength = 4`
The solver exited due to a minimum step length

`objectiveLimit = 3`
The solver exited due to an objective limit

`optimal = 5`
The solver exited with an optimal solution

`unbounded = 8`
The solver exited because the problem is unbounded

unknown = 0

unknown serves as both a default value, and it is used when no other enum member makes sense

8.2 Pyomo Models

Pyomo Representation

class SubModel(**args*, ***kwds*)

This Pyomo model component defines a sub-model in a multi-level problem.

Pyomo models can include nested and parallel SubModel components to express complex multi-level problems.

__init__(**args*, ***kwargs*)

The constructor for SubModel components.

Keyword Arguments **fixed** – A Pyomo variable or a list of Pyomo variables that are optimized by upper-levels in the model.

Parameters

- ***args** – Arguments passed to the SimpleBlock base class.
- ****kwargs** – Other keyword arguments passed to the SimpleBlock base class.

Model Transformations

convert_pyomo2MultilevelProblem(*model*, ***, *determinism=1*, *inequalities=True*, *linear=None*)

Traverse the model an generate a LinearMultilevelProblem or QuadraticMultilevelProblem. Generate errors if this problem cannot be represented in this form.

Parameters

- **model** – A Pyomo model object.
- **determinism** (*int, Default: 1*) – Indicates whether the traversal of **model** is ordered. Valid values are:
  - 0 - Unordered traversal of **model**
  - 1 - Ordered traversal of component indices in **model**
  - 2 - Ordered traversal of components by name in **model**
- **linear** (*bool*) – A flag that indicates whether the expected model representation is linear (True) or quadratic (False). If not specified, then

no error checking is done to confirm whether the model is linear or quadratic.

- **`inequalities`** (*`bool, Default: True`*) – If True, then the multilevel problem object represents all constraints as less-than-or-equal inequalities. Otherwise, the multilevel problem represents all constraints as equalities.

Returns This object corresponds to the problem in **model**.

Return type *LinearMultilevelProblem* or *QuadraticMultilevelProblem*

PAO Solvers

`class PyomoSubmodelSolver_FA`(***kwds*)

PAO FA solver for Pyomo models: pao.pyomo.FA

This solver converts the Pyomo model to a LinearBilevelProblem and calls the pao.mpr.FA solver.

`available`()

Returns a bool indicating if the solver can be executed.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to confirm that a solver license is available).

Returns This method returns True if the solver can be executed.

Return type bool

`is_persistent`()

Returns True if the solver is persistent.

Persistent solvers maintain the model representation in memory, which enables performance optimization when a problem is resolved after changing initial conditions or tweaking model parameters.

The default is to return False, but this method can be overloaded in a subclass to support solver-specific logic.

Returns This method returns True if the solver is persistent.

Return type bool

`solve`(*model*, ***options*)

Executes the solver and loads the solution into the model.

Parameters

- **`model`** – The model that is being optimized.

- **`options`** – Keyword options that are used to configure the solver.

Keyword Arguments

- **`tee`** – If True, then solver output is streamed to stdout. (default is False)
- **`load_solutions`** – If True, then the finale solution is loaded into the model. (default is True)
- **`linearize_bigm`** – The name of the big-M value used to linearize bilinear terms. If this is not specified, then the solver will throw an error if bilinear terms exist in the model.
- **`mip_solver`** – The MIP solver used by FA. (default is glpk)

Returns A summary of the optimization results.

Return type *Results*

`valid_license``()`

Returns a bool indicating if the solver has a valid license.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to check the solver license).

Returns This method returns True if the solver license is valid.

Return type bool

`version``()`

Returns a tuple that describes the solver version.

The return value is a tuple of strings. A typical format is (major, minor, patch), but this is not required. The default behavior is to return an empty tuple.

Returns The solver version.

Return type tuple

`class PyomoSubmodelSolver_MIBS``(**`*kwds*`)`

PAO MibS solver for Pyomo models: pao.pyomo.MIBS

This solver converts the Pyomo model to a LinearBilevelProblem and calls the pao.mpr.MIBS solver.

`available``()`

Returns a bool indicating if the solver can be executed.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to confirm that a solver license is available).

Returns This method returns True if the solver can be executed.

Return type bool

`is_persistent`()

Returns True if the solver is persistent.

Persistent solvers maintain the model representation in memory, which enables performance optimization when a problem is resolved after changing initial conditions or tweaking model parameters.

The default is to return False, but this method can be overloaded in a subclass to support solver-specific logic.

Returns This method returns True if the solver is persistent.

Return type bool

`solve`(*model*, ***options*)

Executes the solver and loads the solution into the model.

Parameters

- **`model`** – The model that is being optimized.
- **`options`** – Keyword options that are used to configure the solver.

Keyword Arguments

- **`tee`** – If True, then solver output is streamed to stdout. (default is False)
- **`load_solutions`** – If True, then the finale solution is loaded into the model. (default is True)
- **`linearize_bigm`** – The name of the big-M value used to linearize bilinear terms. If this is not specified, then the solver will throw an error if bilinear terms exist in the model.

Returns A summary of the optimization results.

Return type *Results*

`valid_license`()

Returns a bool indicating if the solver has a valid license.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to check the solver license).

Returns This method returns True if the solver license is valid.

Return type bool

`version`()

Returns a tuple that describes the solver version.

The return value is a tuple of strings. A typical format is (major, minor, patch), but this is not required. The default behavior is to return an empty tuple.

Returns The solver version.

Return type tuple

`class PyomoSubmodelSolver_PCCG`(***kwds*)

PAO PCCG solver for Pyomo models: pao.pyomo.PCCG

This solver converts the Pyomo model to a LinearBilevelProblem and calls the pao.mpr.PCCG solver.

`available`()

Returns a bool indicating if the solver can be executed.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to confirm that a solver license is available).

Returns This method returns True if the solver can be executed.

Return type bool

`is_persistent`()

Returns True if the solver is persistent.

Persistent solvers maintain the model representation in memory, which enables performance optimization when a problem is resolved after changing initial conditions or tweaking model parameters.

The default is to return False, but this method can be overloaded in a subclass to support solver-specific logic.

Returns This method returns True if the solver is persistent.

Return type bool

`solve`(*model*, ***options*)

Executes the solver and loads the solution into the model.

Parameters

- **`model`** – The model that is being optimized.
- **`options`** – Keyword options that are used to configure the solver.

Keyword Arguments

- **`tee`** – If True, then solver output is streamed to stdout. (default is False)
- **`load_solutions`** – If True, then the finale solution is loaded into the model. (default is True)

- **`linearize_bigm`** – The name of the big-M value used to linearize bilinear terms. If this is not specified, then the solver will throw an error if bilinear terms exist in the model.
- **`mip_solver`** – The MIP solver used by PCCG. (default is cbc)

Returns A summary of the optimization results.

Return type *Results*

`valid_license``()`

Returns a bool indicating if the solver has a valid license.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to check the solver license).

Returns This method returns True if the solver license is valid.

Return type bool

`version``()`

Returns a tuple that describes the solver version.

The return value is a tuple of strings. A typical format is (major, minor, patch), but this is not required. The default behavior is to return an empty tuple.

Returns The solver version.

Return type tuple

`class PyomoSubmodelSolver_REG``(**kwds)`

PAO REG solver for Pyomo models: pao.pyomo.REG

This solver converts the Pyomo model to a LinearBilevelProblem and calls the pao.mpr.REG solver.

`available``()`

Returns a bool indicating if the solver can be executed.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to confirm that a solver license is available).

Returns This method returns True if the solver can be executed.

Return type bool

`is_persistent``()`

Returns True if the solver is persistent.

Persistent solvers maintain the model representation in memory, which enables performance optimization when a problem is resolved after changing initial conditions or tweaking model parameters.

The default is to return False, but this method can be overloaded in a subclass to support solver-specific logic.

Returns This method returns True if the solver is persistent.

Return type bool

`solve`(*model*, ***options*)

Executes the solver and loads the solution into the model.

Parameters

- **`model`** – The model that is being optimized.
- **`options`** – Keyword options that are used to configure the solver.

Keyword Arguments

- **`tee`** – If True, then solver output is streamed to stdout. (default is False)
- **`load_solutions`** – If True, then the finale solution is loaded into the model. (default is True)
- **`linearize_bigm`** – The name of the big-M value used to linearize bilinear terms. If this is not specified, then the solver will throw an error if bilinear terms exist in the model.
- **`nlp_solver`** – The NLP solver used by REG. (default is ipopt)

Returns A summary of the optimization results.

Return type *Results*

`valid_license`()

Returns a bool indicating if the solver has a valid license.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to check the solver license).

Returns This method returns True if the solver license is valid.

Return type bool

`version`()

Returns a tuple that describes the solver version.

The return value is a tuple of strings. A typical format is (major, minor, patch), but this is not required. The default behavior is to return an empty tuple.

Returns The solver version.

Return type tuple

8.3 Compact Models

LinearMultilevelProblem Representation

class LinearMultilevelProblem(*name=None*)

```
For bilevel problems, let:

  U   = LinearMultilevelProblem.U
  L   = U.L
  x   = [U.x, L.x]'       # dense column vector
  U.c = [U.c[U], U.c[L]]' # dense column vector
  L.c = [L.c[U], L.c[L]]' # dense column vector
  U.A = [U.A[U], U.A[L]]  # sparse matrix
  L.A = [L.A[U], L.A[L]]  # sparse matrix

Then we have:

  min_{U.x}   U.c' * x + U.d
  s.t.        U.A  * x       <= U.b                  # Or ==

          where L.x satisifies

              min_{L.x}   L.c' * x + L.d
              s.t.        L.A  * x       <= L.b      # Or ==
```

class QuadraticMultilevelProblem(*name=None, bilinear=False*)

```
For bilevel problems, let:

  U   = QuadraticMultilevelProblem.U
  L   = U.L
  x   = [U.x, L.x]'           # dense column vector
  U.c = [U.c[U], U.c[L]]'     # dense column vector
  U.P = [U.P[U,U], U.P[U,L]]  # sparse matrix
        [0,        U.P[L,L]]
  U.A = [U.A[U], U.A[L]]      # sparse matrix
  U.Q = [U.Q[U,U], U.Q[U,L]]  # sparse matrix
        [0,        U.Q[L,L]]
  L.c = [L.c[U], L.c[L]]'     # dense column vector
  L.P = [L.P[U,U], L.P[U,L]]  # sparse matrix
        [0,        L.P[L,L]]
  L.A = [L.A[U], L.A[L]]      # sparse matrix
  L.Q = [L.Q[U,U], L.Q[U,L]]  # sparse matrix
```

(continues on next page)

(continued from previous page)

```
        [0,         L.Q[L,L]]

Then we have:

  min_{U.x}    U.c' * x + x' * U.P * x + U.d
  s.t.         U.A  * x + x' * U.Q * x       <= U.b
↪# Or ==

          where L.x satisifies

              min_{L.x}    L.c' * x + x' * L.P * x + L.d
              s.t.         L.A  * x + x' * L.Q * x       <= L.b
↪# Or ==
```

Model Transformations

convert_to_standard_form(*M*, *inequalities=False*)

Normalize the LinearMultilevelProblem into a standard form.

This function copies the multilevel problem, **M**, and transforms the problem such that

1. Each real variable x is nonnegative (x >= 0)
2. Constraints have the specified form (e.g. all equalities or all inequalities)
3. Each level is a minimization problem

Parameters

- **M** (LinearMultilevelProblem) – The model that is being normalized
- **inequalities** (*bool, Default: False*) – If this is True, then the normalized form has inequality constraints. Otherwise, the normalized form has equality constraints.

Returns A normalized version of the input model

Return type *LinearMultilevelProblem*

linearize_bilinear_terms(*M*, *bigM*)

Generate a linear multilevel problem from a quadratic multilevel problem by linearizing bilinear terms.

This function copies the linear terms in the multilevel problem, **M**, and replaces bilinear terms with a new variable. This transformation only applies when at least one of the variables in each bilinear term is binary or integer.

Parameters

- **`M`** (`QuadraticMultilevelProblem`) – The model that is being linearized
- **`bigM`** (*`float`*) – The big-M value used to linearize nonlinear terms

Returns A linearized version of the input model

Return type *LinearMultilevelProblem*

PAO Solvers

class `LinearMultilevelSolver_FA`(***kwds*)

PAO FA solver for linear MPRs: pao.mpr.FA

This solver replaces lower-level problems using the KKT conditions and calls a MIP solver to solve the reformulated problem.

`available`()

Returns a bool indicating if the solver can be executed.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to confirm that a solver license is available).

Returns This method returns True if the solver can be executed.

Return type bool

`is_persistent`()

Returns True if the solver is persistent.

Persistent solvers maintain the model representation in memory, which enables performance optimization when a problem is resolved after changing initial conditions or tweaking model parameters.

The default is to return False, but this method can be overloaded in a subclass to support solver-specific logic.

Returns This method returns True if the solver is persistent.

Return type bool

`solve`(*model*, ***options*)

Executes the solver and loads the solution into the model.

Parameters

- **`model`** – The model that is being optimized.
- **`options`** – Keyword options that are used to configure the solver.

Keyword Arguments

- **`tee`** – If True, then solver output is streamed to stdout. (default is False)
- **`load_solutions`** – If True, then the finale solution is loaded into the model. (default is True)
- **`mip_solver`** – The MIP solver used by FA. (default is glpk)
- **`bigm`** – The big-M value used to enforce complementarity conditions. (default is 1e5)

Returns A summary of the optimization results.

Return type *Results*

`valid_license``()`

Returns a bool indicating if the solver has a valid license.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to check the solver license).

Returns This method returns True if the solver license is valid.

Return type bool

`version``()`

Returns a tuple that describes the solver version.

The return value is a tuple of strings. A typical format is (major, minor, patch), but this is not required. The default behavior is to return an empty tuple.

Returns The solver version.

Return type tuple

`class LinearMultilevelSolver_MIBS``(`***kwds*`)`

PAO MibS solver for linear MPRs: pao.mpr.MIBS

`available``()`

Returns a bool indicating if the solver can be executed.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to confirm that a solver license is available).

Returns This method returns True if the solver can be executed.

Return type bool

`create_mibs_model``(`*repn*, *mps_filename*, *aux_filename*`)`

TODO - Document this transformation

is_persistent()

Returns True if the solver is persistent.

Persistent solvers maintain the model representation in memory, which enables performance optimization when a problem is resolved after changing initial conditions or tweaking model parameters.

The default is to return False, but this method can be overloaded in a subclass to support solver-specific logic.

Returns This method returns True if the solver is persistent.

Return type bool

solve(*model*, ***options*)

Executes the solver and loads the solution into the model.

Parameters

- **model** – The model that is being optimized.
- **options** – Keyword options that are used to configure the solver.

Keyword Arguments

- **tee** – If True, then solver output is streamed to stdout. (default is False)
- **load_solutions** – If True, then the finale solution is loaded into the model. (default is True)
- **executable** – The executable used for MibS. (default is mibs)
- **param_file** – The parameter file used to configure MibS. (default is None)

Returns A summary of the optimization results.

Return type *Results*

valid_license()

Returns a bool indicating if the solver has a valid license.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to check the solver license).

Returns This method returns True if the solver license is valid.

Return type bool

version()

Returns a tuple that describes the solver version.

The return value is a tuple of strings. A typical format is (major, minor, patch), but this is not required. The default behavior is to return an empty tuple.

Returns The solver version.

Return type tuple

class LinearMultilevelSolver_PCCG(***kwds*)

PAO PCCG solver for linear MPRs: pao.mpr.PCCG

This solver iteratively adds constraints to tighten a relaxation of the lower-level problem.

available()

Returns a bool indicating if the solver can be executed.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to confirm that a solver license is available).

Returns This method returns True if the solver can be executed.

Return type bool

is_persistent()

Returns True if the solver is persistent.

Persistent solvers maintain the model representation in memory, which enables performance optimization when a problem is resolved after changing initial conditions or tweaking model parameters.

The default is to return False, but this method can be overloaded in a subclass to support solver-specific logic.

Returns This method returns True if the solver is persistent.

Return type bool

solve(*mpr*, *options=None*, ***config_options*)

Executes the solver and loads the solution into the model.

Parameters

- **model** – The model that is being optimized.
- **options** – Keyword options that are used to configure the solver.

Keyword Arguments

- **tee** – If True, then solver output is streamed to stdout. (default is False)
- **load_solutions** – If True, then the finale solution is loaded into the model. (default is True)
- **mip_solver** – The MIP solver used by PCCG. (default is cbc)
- **bigm** – The big-M value used to enforce complementarity conditions. (default is 1e6)

- **epsilon** – Parameter used in disjunction approximation. (default is 1e-4)
- **atol** – Convergence tolerance for |UB-LB|. (default is 1e-8)
- **rtol** – Convergence tolerance for |UB-LB|. (default is 1e-8)
- **maxit** – Maximum number of iterations. (default is None)
- **quiet** – If False, then enable verbose solver output. (default is True)

Returns A summary of the optimization results.

Return type *Results*

valid_license()

Returns a bool indicating if the solver has a valid license.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to check the solver license).

Returns This method returns True if the solver license is valid.

Return type bool

version()

Returns a tuple that describes the solver version.

The return value is a tuple of strings. A typical format is (major, minor, patch), but this is not required. The default behavior is to return an empty tuple.

Returns The solver version.

Return type tuple

class LinearMultilevelSolver_REG(***kwds*)

PAO REG solver for linear MPRs: pao.mpr.REG

This solver replaces lower-level problems using the KKT conditions and calls a NLP solver to solve the reformulated problem.

available()

Returns a bool indicating if the solver can be executed.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to confirm that a solver license is available).

Returns This method returns True if the solver can be executed.

Return type bool

is_persistent()

Returns True if the solver is persistent.

Persistent solvers maintain the model representation in memory, which enables performance optimization when a problem is resolved after changing initial conditions or tweaking model parameters.

The default is to return False, but this method can be overloaded in a subclass to support solver-specific logic.

Returns This method returns True if the solver is persistent.

Return type bool

`solve`(*model*, ***options*)

Executes the solver and loads the solution into the model.

Parameters

- **`model`** – The model that is being optimized.
- **`options`** – Keyword options that are used to configure the solver.

Keyword Arguments

- **`tee`** – If True, then solver output is streamed to stdout. (default is False)
- **`load_solutions`** – If True, then the finale solution is loaded into the model. (default is True)
- **`nlp_solver`** – The NLP solver used by REG. (default is ipopt)
- **`rho`** – The tolerance for constraints that enforce complementarity conditions. (default is 1e-7)

Returns A summary of the optimization results.

Return type *Results*

`valid_license`()

Returns a bool indicating if the solver has a valid license.

The default behavior is to always return *True*, but this method can be overloaded in a subclass to support solver-specific logic (e.g. to check the solver license).

Returns This method returns True if the solver license is valid.

Return type bool

`version`()

Returns a tuple that describes the solver version.

The return value is a tuple of strings. A typical format is (major, minor, patch), but this is not required. The default behavior is to return an empty tuple.

Returns The solver version.

Return type tuple

Warning: The logic in `pao.duality` is currently disabled. There are known errors in this code that will be resolved by re-implementing it using the logic in `pao.mpr`.

References

[1] G. Anandalingam, A mathematical programming model of decentralized multi-level systems. J. Oper.Res. Soc.39(11), 1021–1033 (1988)

[2] J. F. Bard, Practical bilevel optimization: Algorithms and applications. Kluwer Academic Publishers, 1998.

[3] J. Fortuny-Amat and B. McCarl, A representation and economic interpretation of a two-level programming problem, The Journal of the Operations Research Society. 32(9), 783-792, 1981.

[4] https://neos-server.org/neos/

[5] W. E. Hart, C. D. Laird, J.-P. Watson, D. L. Woodruff, G. A. Hackebeil, B. L. Nicholson, J. D. Siirola. Pyomo - Optimization Modeling in Python, 2nd Edition. Springer Optimization and Its Applications, Vol 67. Springer, 2017.

[6] http://www.pyomo.org/

[7] https://github.com/Pyomo/pyomo

DISTRIBUTION

Email—Internal (encrypt for OUO)

Name	Org.	Sandia Email Address
Technical Library	01911	libref@sandia.gov